

Simple Polymorphic Usage Analysis

Simple Polymorphic Usage Analysis

Keith Wansbrough
Clare Hall



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

A dissertation submitted for the degree
of Doctor of Philosophy.

28 March, 2002

Publication history:

Submitted 28 March, 2002.

Examined 13 September, 2002.

Required corrections made 18 November, 2002.

Revised 31 March, 2003.

Indexed 10 March, 2005.

Also appears as University of Cambridge Computer Laboratory
Technical Report UCAM-CL-TR-623, March 2005. Subtract 20
from each page number in the technical report to obtain the
corresponding page number in this dissertation.

Copyright © 2002, 2003 Keith Wansbrough.

All rights reserved.

web: <http://www.lochan.org/keith/>

email: keith@lochan.org

Soli deo gloria

Abstract

Simple Polymorphic Usage Analysis

Ph.D. Thesis

Keith Wansbrough

Computer Laboratory
University of Cambridge
Cambridge, England

28 March, 2002.

Implementations of lazy functional languages ensure that computations are performed only when they are needed, and save the results so that they are not repeated. This frees the programmer to describe solutions at a high level, leaving details of control flow to the compiler.

This freedom however places a heavy burden on the compiler; measurements show that over 70% of these saved results are never used again. A usage analysis that could statically detect values used at most once would enable these wasted updates to be avoided, and would be of great benefit. However, existing usage analyses either give poor results or have been applied only to prototype compilers or toy languages.

This thesis presents a sound, practical, type-based usage analysis that copes with all the language features of a modern functional language, including type polymorphism and user-defined algebraic data types, and addresses a range of problems that have caused difficulty for previous analyses, including poisoning, mutual recursion, separate compilation, and partial application and usage dependencies. In addition to well-typing rules, an inference algorithm is developed, with proofs of soundness and a complexity analysis.

In the process, the thesis develops simple polymorphism, a novel approach to polymorphism in the presence of subtyping that attempts to strike a balance between pragmatic concerns and expressive power. This thesis may be considered an extended experiment into this approach, worked out in some detail but not yet conclusive.

The analysis described was designed in parallel with a full implementation in the Glasgow Haskell Compiler, leading to informed design choices, thorough coverage of language features, and accurate measurements of its potential and effectiveness when used on real code. The latter demonstrate that the analysis yields moderate benefit in practice.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation does not exceed 80 000 words, excluding appendices, bibliography, and figures.

Acknowledgements

When I first arrived in Glasgow to commence my PhD, I had little idea of what lay in store for me. Now I am at the end of that process, living in another country and looking back on the last few years of ups and downs with a sense of accomplishment. I have struggled and succeeded, I have been stretched and I have grown. And I have completed the thesis!

A work of this size is never performed in isolation, and I have been privileged to have the support of many others. To my New Zealand friends around the world an especial thank you, for you have stuck by me despite my poor track record of emails and telephone calls. Your wisdom, companionship, and occasional visits have been invaluable. To my flatmates, thank you for making the places I've lived in feel like home, for late-night conversations and friendly faces, and for rubbing off some of my sharper edges! To my church friends, and to the Tuesday night group, thank you for welcoming me to two new cities, for showing me around and providing me with a social network, and for being the international family of God to me. To the Scottish crowd, thank you for welcoming me with open arms and including me in the group. To my officemates, thank you for technical and non-technical discussions, for books and advice, and for accepting my taste in music. And finally to James, Gayle, Richard, and Allie, friends on the spot, thank you for all the prayers and diversion and food and fun.

This entire enterprise would not have been possible without the generosity of the Commonwealth Scholarship Commission, for which I am very grateful. I am also indebted to the EPSRC, GR/M/04716 for covering some research costs and GR/N/24872/01 for funding my current research position and thereby involuntarily contributing to this present work (thank you Peter for your forbearance; I've finished now!). My Cambridge college, Clare Hall, has provided me with accommodation and society. The Universities of Glasgow and Cambridge have provided ample facilities for my use, and I particularly wish to thank the technical support teams of the Department of Computing Science and the Computer Laboratory respectively for their timely and knowledgeable assistance (especially Piete Brooks, who has solved innumerable problems out-of-hours). I am also grateful to Microsoft Research, Cambridge, where I spent several months working on the implementation with the GHC team and drinking real hot chocolate.

For criticism, suggestion, and encouragement, I am grateful to the many people who have read and discussed my work over the years, as colleagues, visitors, referees, conference participants, and correspondents. There are too many to list them all, but I am particularly grateful to David N. Turner and Clem Baker-Finch for early discussions, to those who read and commented on the note that eventually became Chapter 4 and are listed in the acknowledgements to [WPJ00], to Andy Moran for discussions on operational semantics, and to those who took considerable time to read drafts of this thesis, providing me with rafts of corrections: Mick Francis, Jörgen Gustavsson, Joshua Lawrence, Andrei Serjantov, Peter Sewell, and Martin Sulzmann.

I have had the privilege of two supervisors in this work. At Glasgow, I began my research with Simon Peyton Jones. When Simon moved to join Microsoft Research, he arranged a transfer for me to the University of Cambridge. Simon remained my unofficial primary supervisor, but my official supervisor became Andrew Pitts. I have been grateful for the assistance of both: Simon for his unquenchable enthusiasm and his experience of compiler implementation, and Andy for his knowledge of theory and semantics. Both have guided my work, made crucial suggestions, read and corrected drafts, and supported me personally.

Finally, I thank my family. My parents, Norman and Tony, started me on this road a long time ago and have never stopped believing in me or loving me. My sister Heather understands me and shares my love of knowledge. Doug and Pat McEwan have welcomed me into their home and into their lives, and have become my second family. But last and greatest, my wonderful wife Carolyn has been an incredible support, in sickness and in health, in thesis-writing mode and in holiday, blessing me with her companionship, kindness, and unfailing love.

Thanks to all of you.

– Keith Wansbrough, Cambridge, 28 March, 2002.

Brief Contents

Abstract	v
Declaration	vii
Acknowledgements	ix
1 Introduction	1
2 Formal Framework	25
3 Monomorphic Usage Types	41
4 Polymorphic Usage Types	73
5 Covering the Full Language	127
6 Implementation	167
7 Conclusion and Future Work	223
A Index of Notation	231
B The Full Type System	237
C Extending the Lattice	251
D Proofs	275
Bibliography	313
Index	331

Contents

Abstract	v
Declaration	vii
Acknowledgements	ix
1 Introduction	1
1.1 Contributions of this thesis	1
1.2 The context: Lazy functional languages and optimisation	3
1.2.1 Strongly-typed lazy functional languages	3
1.2.2 Types	3
1.2.3 Laziness	4
1.2.4 The need for optimisation	5
1.2.5 The Glasgow Haskell Compiler	6
1.3 Usage	8
1.3.1 Thunks and updates	8
1.3.2 Update avoidance: the opportunity	9
1.3.3 Usage analysis	10
1.3.4 Optimisations enabled by usage information	11
1.3.5 Usage analyses: related work	12
1.4 Program analysis	15
1.4.1 Type-based analysis	16
1.4.2 Limitations of program analysis	17
1.5 Designing a type-based usage analysis	17
1.5.1 Structure	17
1.5.2 Practical considerations	18
1.5.3 Performance measurement	19
1.5.4 Proof techniques	20
1.5.5 Soft typing	20
1.5.6 Dealing with separate compilation	21
1.6 How To Read This Thesis	22
2 Formal Framework	25
2.1 Introduction	25
2.2 The source language	27
2.2.1 Source typing	28

2.3	The executable language	29
2.3.1	The language	29
2.3.2	Erasure	31
2.3.3	The trivial translation	31
2.4	Operational semantics	31
2.4.1	Statics	33
2.4.2	Dynamics	33
2.4.3	Evaluation and termination	34
2.4.4	The trivial translation	35
2.4.5	Copying and using abstractions	36
2.5	Proof of soundness	36
2.5.1	Definitions	37
2.5.2	Proof outline	37
2.6	Related work	39
3	Monomorphic Usage Types	41
3.1	Introduction: a usage analysis	41
3.2	A language with usage types	42
3.2.1	The type language	43
3.2.2	The term language	44
3.3	Usage well-typing rules	47
3.3.1	Update flags	47
3.3.2	Basic uses	47
3.3.3	The syntactic occurrence function	48
3.3.4	Occurrences in a closure	48
3.3.5	Subsumption	48
3.3.6	Demands and recursive binding groups	50
3.4	The goodness ordering	51
3.4.1	Intuition	52
3.4.2	Existence	52
3.4.3	Covariance of \sqsubseteq	52
3.5	Usage inference	53
3.5.1	Constraints	55
3.5.2	Inference algorithm phase 1	55
3.5.3	Pessimisation	56
3.5.4	Inference algorithm phase 2	57
3.6	Proofs	59
3.6.1	Well-typing rules	59
3.6.2	Inference phase 1	61
3.6.3	Inference phase 2	61
3.6.4	Overall results	63
3.7	Discussion	64
3.7.1	Curried functions and separate compilation	64
3.7.2	No principal types?	66
3.8	Separate compilation	67
3.9	Related work	68

3.9.1	Gustavsson	68
3.9.2	Uniqueness types	69
3.9.3	Subsumption	70
3.9.4	Type inference	71
3.9.5	The goodness ordering	71
3.9.6	Constraint solution	71
3.9.7	Occurrences and affine linear logic	71
4	Polymorphic Usage Types	73
4.1	Introduction	73
4.1.1	Problems with monomorphic usage analysis	74
4.1.2	An idea: polymorphism	74
4.1.3	A solution: Simple polymorphism	75
4.1.4	Examples	77
4.2	A language with polymorphic usage types	78
4.2.1	The type language	78
4.2.2	The term language	81
4.2.3	The operational semantics: type erasure without erasing types	81
4.3	Polymorphic usage well-typing rules	83
4.3.1	Update flags	83
4.3.2	Usage abstraction and application	85
4.3.3	Subtyping	86
4.4	Polymorphic usage inference	86
4.4.1	Inference phase 1	87
4.4.2	Generalisation	87
4.4.3	Pessimisation	92
4.4.4	Inference phase 2	92
4.5	Generalisation and simple polymorphism	92
4.5.1	Most-general types and maximal applicability	93
4.5.2	Which type is ‘best’?	93
4.5.3	Forbidden variables	99
4.5.4	The closure algorithm	101
4.5.5	Computing the transitive closure	105
4.6	Proofs	106
4.6.1	Well-typing rules	106
4.6.2	Inference phase 1	107
4.6.3	Inference phase 2	110
4.6.4	Overall results	110
4.7	Discussion	111
4.7.1	Application of simple polymorphism	111
4.7.2	Better generalisation	111
4.7.3	Usage analysis and generalisation	112
4.7.4	Beyond ML-style polymorphism	114
4.7.5	Implementation	115
4.8	Related work	115
4.8.1	Constrained polymorphism	116

4.8.1.1	Polymorphism and subtyping	116
4.8.1.2	Type inference and constraints	117
4.8.1.3	Constraint solution	117
4.8.1.4	Constraint simplification	118
4.8.1.5	Constraint approximation	119
4.8.1.6	New approaches	119
4.8.2	Pragmatic subtyping	120
4.8.3	Annotation polymorphism	121
4.8.4	Constraint approximation in HM(X)	123
5	Covering the Full Language	127
5.1	The full source language	127
5.1.1	Type polymorphism	128
5.1.2	User-defined algebraic data types	128
5.1.3	Language extensions	129
5.1.4	Operational semantics	131
5.2	Accommodating type polymorphism	134
5.2.1	Generalised types	134
5.2.2	Type variables	134
5.2.3	Typing	135
5.2.4	Subtyping	135
5.3	Accommodating data types	135
5.3.1	Separation between creation and use	138
5.3.2	Sharing	139
5.3.3	Typing	141
5.3.4	Subtyping	143
5.3.4.1	The intuition	143
5.3.4.2	A coinductive definition	144
5.3.4.3	An inductive definition	145
5.4	Annotation of data type declarations	147
5.4.1	Some typical data structures	147
5.4.2	Two extreme approaches	148
5.4.3	The issues	149
5.4.4	Some workable approaches	150
5.4.4.1	Equal	150
5.4.4.2	Full	151
5.4.4.3	Clean	151
5.4.4.4	Restricted	153
5.4.4.5	Discussion	153
5.4.5	Recursion	154
5.4.5.1	Direct self-recursion	154
5.4.5.2	Direct mutual recursion	156
5.4.5.3	Indirect recursion	157
5.4.5.4	Non-uniform recursion	158
5.4.5.5	Non-regular recursion	159
5.5	Usage inference in the full language	159

5.5.1	Inference phase 1	161
5.6	Proofs	161
5.6.1	Well-typing rules	161
5.6.2	Well-typing rules	162
5.6.3	Inference phase 1	163
5.6.4	Inference phase 2	163
5.6.5	Overall results	164
5.7	Related work	164
5.7.1	Gustavsson	164
5.7.2	Subtyping of data types	165
6	Implementation	167
6.1	The Glasgow Haskell Compiler	168
6.2	Implementing the analysis	168
6.2.1	A chronology	168
6.2.2	Usage projection types	171
6.2.3	Implementing the inference itself	172
6.3	Design choices	173
6.3.1	Locating the inference	173
6.3.2	Restricting generalisation	173
6.3.3	Usage specialisation	174
6.3.3.1	Call sites	174
6.3.3.2	Using specialisations	174
6.3.3.3	Interesting variables	175
6.3.3.4	Generating specialisations	175
6.3.3.5	Discussion and related work	176
6.4	Inference for Core constructs	177
6.4.1	Binds	177
6.4.2	Modules	177
6.4.3	Unsaturated constructors	177
6.4.4	Beyond A-normal form	177
6.4.5	Case expressions	178
6.4.6	Type coercions	179
6.4.7	Unboxed data types	180
6.4.8	Primops	180
6.4.9	Variance analysis	181
6.4.10	Unnecessary constructor usage arguments	181
6.4.11	Elaborating data type annotation schemes	182
6.4.12	Intermodule analysis	184
6.5	Inference for GHC-specific constructs	184
6.5.1	Substitution and Type invariants	185
6.5.2	Storing usage projection types	185
6.5.3	Rules	186
6.5.4	Existential constructors	187
6.6	Using the results of the analysis	187
6.6.1	Update avoidance	187

6.6.2	Informing the simplifier	187
6.6.3	Configuration	189
6.7	Measuring performance	190
6.7.1	The NoFib test suite	190
6.7.2	Ticky-ticky profiling	191
6.7.3	Timings	193
6.8	Results	193
6.8.1	Effectiveness of the analysis	193
6.8.2	Comparing the analyses	195
6.8.3	Run time and allocation	197
6.8.4	Examining the effect of one-shot lambda information	200
6.8.5	Examining the effect of generalisation strategy	204
6.8.6	Costs of analysis	206
6.9	Case studies	209
6.9.1	Queens: a case study	209
6.9.2	Boyer: another case study	210
6.10	Conclusion	212
6.10.1	The implementation	212
6.10.2	Results obtained	213
6.10.3	Easy future work	214
7	Conclusion and Future Work	223
7.1	The problem	223
7.2	The solution	224
7.3	The development	225
7.3.1	Usage analysis	225
7.3.2	Implementing a type-based analysis	226
7.4	Future work	228
7.5	Concluding remarks	230
A	Index of Notation	231
A.1	Basic notation	231
A.2	Languages	231
A.3	Alphabetic notation	232
A.4	Nonalphabetic notation	234
B	The Full Type System	237
C	Extending the Lattice	251
C.1	Absence, usage, and zero usage	252
C.2	Strictness and the demand/use distinction	253
C.3	An operational semantics	254
C.3.1	The language	254
C.3.2	The annotation domain and operations	255
C.3.3	The operational semantics	257
C.3.4	Example of behaviour	261

C.4	A type system	264
C.4.1	The type language	264
C.4.2	Lifting the annotation operations to types and environments .	266
C.4.3	Well-typing rules	267
C.4.4	Data structures	270
C.4.5	Weakening and contraction	270
C.4.6	Sample typing	271
C.4.7	Proof of soundness	271
C.5	Discussion	271
C.6	Related work	272
D	Proofs	275
D.1	Type soundness	275
D.2	Correctness	288
D.3	Constraints	290
D.4	Soundness of inference phase 1	291
D.5	Soundness and pseudo-completeness of inference phase 2	308
D.6	Extended system	310
	Bibliography	313
	Index	331

List of Figures

1.1	The architecture of the Glasgow Haskell Compiler.	7
1.2	The opportunity.	10
1.3	Thesis structure.	23
2.1	The source language L_0	26
2.2	Well-typing rules for the source language L_0	28
2.3	The executable language LIX_0 and configurations $LIXC_0$	30
2.4	The $LIXC_0$ operational semantics: \rightarrow and \rightarrow_δ	32
3.1	The usage-typed language LIX_1	43
3.2	Well-typing rules for LIX_1	46
3.3	The subtype (\preceq) and primitive (\leq) orderings over LIX_1	49
3.4	Type inference rules from L_0 to LIX_1	54
4.1	Some sample typings.	76
4.2	The polymorphically usage-typed language LIX_2	80
4.3	The operational semantics of usage polymorphism, $LIXC_2$	82
4.4	Well-typing rules for LIX_2	84
4.5	The subtype (\preceq) and primitive (\leq) orderings over LIX_2	86
4.6	Basic type inference rules from L_0 to LIX_2 , omitting (\blacktriangleright_2 -LETREC).	88
4.7	Type inference rule (\blacktriangleright_2 -LETREC).	90
4.8	Constraint and subtyping lattice of $idInt$	94
4.9	Constraint and subtyping lattice of $kInt$	96
4.10	Bowtie constraint and subtyping lattice.	97
4.11	Constraint and subtyping lattice of $twice$ and $plus3$	98
4.12	The closure operation.	102
4.13	Logical type rules for bounded usage quantification.	122
5.1	The full source language FL_0	129
5.2	Well-typing rules for the full source language FL_0	130
5.3	The executable language $FLIX_0$ and configurations $FLIXC_0$	132
5.4	The full operational semantics.	133
5.5	The polymorphically usage-typed language $FLIX_2$	136
5.6	Well-typing rules for $FLIX_2$	137
5.7	The subtype (\preceq) and primitive (\leq) orderings over $FLIX_2$	142
5.8	Positive and negative free occurrences.	146
5.9	Type inference rules from FL_0 to $FLIX_2$	160

6.1	Usage projection types for GHC.	170
6.2	The default projection – safely annotating with ω	170
6.3	Effectiveness of usage analysis.	194
6.4	Effectiveness versus opportunity.	195
6.5	Comparing the effectiveness of different usage analyses.	196
6.6	Effect of analysis on run time.	198
6.7	Allocations with and without the analysis.	199
6.8	Varying the analysis parameters: one-shot lambdas: allocations.	205
6.9	Varying the analysis parameters: one-shot lambdas: effectiveness.	206
6.10	Varying the analysis parameters: generalisation and specialisation: effectiveness.	207
6.11	Varying the analysis parameters: generalisation and specialisation: allocations.	208
6.12	Main loop of queens after optimisation.	209
6.13	Core terms.	216
6.14	Core types.	217
B.1	The full source language FL_0	238
B.2	Well-typing rules for the source language FL_0	239
B.3	The polymorphically usage-typed language $FLIX_2$	240
B.4	The executable language $FLIX_2$ and configurations $FLIXC_2$	241
B.5	The full operational semantics of $FLIXC_2$	242
B.6(a)	Well-typing rules for $FLIX_2$	243
B.6(b)	Well-typing rules for $FLIX_2$	244
B.7	Translation function and well-typing rule for $FLIXC_2$ contexts.	244
B.8	The subtype (\preceq) and primitive (\leq) orderings over $FLIX_2$	245
B.9	Positive and negative free occurrences.	246
B.10	Annotations.	246
B.11(a)	Type inference rules from FL_0 to $FLIX_2$	247
B.11(b)	Type inference rules from FL_0 to $FLIX_2$	248
B.12	The trivial non-closure operation.	249
B.13	The closure operation.	249
B.14	Stripping \natural and erasure \flat for $FLIX_2$	250
C.1	The extended-annotated executable language ELX	255
C.2	The annotation domain and operations.	256
C.3	Syntactic categories for the ELX operational semantics.	258
C.4	The operational semantics of ELX with auxiliary functions $use(S)$, $use(R)$	260
C.5	The typed language EL_1	265
C.6	Well-typing rules for EL_1	268
D.1	Contexts.	310

List of Tables

1.1	Languages in this thesis.	22
4.1	Run time improvement and effectiveness of usage analysis.	116
6.1	Opportunity and effectiveness of usage analysis (2002-03-18i). . .	200
6.2	Run time (2002-03-18h,i,j).	201
6.3	Bytes allocated (2002-03-18h,i,j).	201
6.4	Stability of opportunity percentage over analyses.	202
6.5	Stability of effectiveness percentage over usage analyses.	203
6.6	Binary size (2002-03-18h,i,j).	218
6.7	Mutator time (2002-03-18h,i,j).	218
6.8	Garbage collected (2002-03-18h,i,j).	219
6.9	Total size of modules (2002-03-18h,i,j).	219
6.10	Total compilation time (2002-03-18h,i,j).	220
6.11	Opportunity and effectiveness without usage analysis (2002-03-18h).	220
6.12	Opportunity and effectiveness of heavy usage analysis (2002-03-18j).	221

Chapter 1.



Introduction

Lazy functional languages ensure that computations are performed only when they are needed, and save the results so that computations are never repeated. This frees the programmer to describe solutions at a high level, leaving details of control flow to the compiler.

This freedom however places a heavy burden upon the compiler. In recent tests over a wide range of programs written in a lazy functional language and compiled using a modern optimising compiler, the programs were found to spend over 20% of their time needlessly updating values that were never used again – in fact, over 70% of updates were wasted (Section 1.3.2). A *usage analysis* that could statically detect values not used again would enable these wasted updates to be avoided, and would be of great benefit. However, existing usage analyses either give poor results or have been applied only to prototype compilers or toy languages.

In this thesis, I design a usage analysis that is applicable to full-scale languages, and implement it in a production compiler. In the process, I develop novel techniques to balance practicality with accuracy. The implementation guides the development, and enables quantitative measurements of its benefit in practice.

1.1 Contributions of this thesis

This thesis makes three major contributions:

- I design a *practical usage analysis* that copes with *all* the language features found in a modern functional language implementation, including type polymorphism (Section 5.2) and user-defined algebraic data types (Section 5.3), and addresses a range of problems that have caused difficulty for previous analyses, including poisoning (Section 3.3.5), mutual recursion (Section 3.3.6), separate compilation (Section 3.8), and partial application and usage depen-

dencies (Section 4.1.3). I give inference algorithms as well as type systems (Section 5.5), with proofs of soundness and complexity analyses (Section 5.6).

- In the process, I develop *simple polymorphism*, a type system for polymorphism in the presence of subtyping that attempts to strike a novel balance between pragmatic concerns and expressive power (Section 4.5). This thesis may be considered an extended experiment into this approach, worked out in some detail but not yet conclusive.
- The analysis described was designed in parallel with a *real implementation* in the Glasgow Haskell Compiler (Chapter 6), leading to informed design choices, thorough coverage of language features, and accurate measurements of its potential and effectiveness when used on real code. I show that the analysis yields moderate benefit in practice (Section 6.8).

In the process, I make the following technical contributions:

- I give an operational semantics for a language with Church-style explicit types and Girard–Reynolds polymorphism, which preserves the intended *type-erasure* semantics but *without erasing the types* (Section 4.2.3). This is a useful proof technique for demonstrating soundness results in polymorphic calculi.
- I present a coherent story of *subtyping for algebraic data types*, where subtyping is performed on the *arguments* of type constructors rather than merely on the constructors themselves, and derive an effective decision algorithm (Section 5.3.4).
- I design a notation, *annotation schemes*, that formally describes any of a wide range of alternatives which may be taken in assigning usage annotations to algebraic data types (Section 5.4). I parameterise my well-typing and inference rules over these descriptions, and present annotation schemes that encapsulate a variety of previous approaches as well as some novel ones. I consider approaches to mutual, nested, negative, and non-regular recursion in some detail.
- I design a novel *closure algorithm* which performs constraint *approximation* as required by simple polymorphism, rather than purely simplification as is usual (Section 4.5.4). The algorithm is explained in detail.
- I present a *constraint solution algorithm* which is used both alone (Section 3.5.4) and as part of the closure algorithm (Section 4.5.5). The algorithm processes the simple atomic constraints generated by my inference in essentially linear time, and I prove its correctness and complexity (Section 3.6.3).
- A few analyses in the past have used a *goodness ordering* on solutions that is *covariant* on arrow types, rather than contravariant as one might expect. I explain why this is justified (Section 3.4).
- I identify two kinds of usage, *demand* and *use*, and formally capture the distinction in both an operational semantics and a (sketched) type system (Appendix C). I explain the connection with strictness, absence, and linear logics.

Finally, I believe that the present thesis gives a *clearer exposition* of usage analysis than any previous work. I give examples and discuss difficult points, I distinguish clearly between the output of a usage analysis and the means by which it is obtained, I compare the existing literature, and wherever possible I expose a design space rather than merely selecting a single alternative. The analysis is developed in stages: Chapter 2 establishes the formal framework, Chapter 3 describes a straightforward monomorphic usage analysis for a toy language, Chapter 4 describes the simple-polymorphic usage analysis, and Chapter 5 extends this to a full-featured language. The thesis surveys and critically examines the related work on usage analysis, other program analyses, subtyping and polymorphism, polymorphic type inference, algebraic data types, and other areas. Useful references into the literature are given throughout.

The structure of the thesis is outlined and illustrated in Section 1.6.

1.2 The context: Lazy functional languages and optimisation

In the remainder of this chapter, we outline the approach taken by this thesis and place it within its wider context. We do not attempt a survey of all relevant literature; instead the state of the art will be referenced and discussed chapter by chapter.

1.2.1 Strongly-typed lazy functional languages

Functional languages are programming languages based more or less directly on Church's lambda calculus [Chu33]. The earliest was probably LISP [MAE⁺62], but pure functional languages did not appear until some time after a seminal series of papers by Landin describing the experimental language ISWIM [Lan64, Lan66].

The most popular functional languages in use today are ML [MTHM97] and Haskell [PJH⁺99]. ML is *impure* and *strict*, while Haskell is *pure* and *lazy*.¹ This thesis mainly concerns the latter class of languages, although similar analysis techniques may well be applicable to the former.

We do not attempt to argue here the benefits of functional languages generally or lazy languages in particular; instead the reader is referred to the excellent discussions of [Lan66, Hug89, HJ94, Wad98].

1.2.2 Types

An important characteristic of modern functional languages, and indeed many non-functional languages as well, is that they are *strongly typed*. This means that every expression is statically (at compile time) given a *type*, which describes its expected behaviour at runtime, and these are checked for compatibility. If there is a mismatch (a *type error*), then the program is not compiled; if on the other hand the program is

¹To be strictly correct, all existing Haskell *implementations* are lazy, but Haskell itself is defined only to be *non-strict*.

type correct, then since every function application is to a demonstrably compatible argument, it is certain that the program will never crash (or *go wrong* [Mil78]) due to an incompatible argument. This is a powerful aid to the programmer in debugging; in fact it is commonly said (and with a fair degree of truth) that once a program has been made to pass the type checker, it is correct! This surprising observation is probably due to the fact that types are able to express the *intentions* of the programmer in a form comprehensible to the compiler. The expressivity of the typing paradigm has also encouraged the use of higher-order abstractions such as maps and folds in functional programming to a much greater degree than in the procedural community (*pace* the recent popularity of Design Patterns [GHJV95]). Thus these languages are sometimes referred to as HOT languages, for Higher-Order and Typed [Wad97]. The benefits of strong typing are argued elsewhere, *e.g.*, [CW85, Pie02].

Types also have advantages for the compiler. Since it is certain that all function applications are to compatible arguments, for example, there is no need to perform runtime checking. Compilation via a typed intermediate language is also popular [SA95, HM95, MWCG99, PJS98a], both because the additional information available enables a wide variety of type-directed optimisations to be performed, and because the type system provides the compiler designer with the same type safety properties as the source language type system provides the programmer. The latter can be particularly useful, trapping many incorrectly-written transformations at an early stage [PJS98a, §11].

1.2.3 Laziness

The earliest functional languages were *strict*, or *call-by-value*. When evaluating the application of a function to an argument, the argument is first evaluated to a value, and then this value is substituted into the body of the function in place of the formal parameter. Call-by-value evaluation is familiar from procedural languages such as C or Pascal, and it is efficiently implementable. But its semantics is different from the usual semantics of the lambda calculus, and certain idioms are difficult to express, such as functions on infinite streams or control-flow abstractions like the *if* function. In call-by-value, the expression $(\lambda x . x + x) (1 + 2)$ reduces as follows

$$(\lambda x . x + x) (1 + 2) \mapsto_v (\lambda x . x + x) 3 \mapsto_v 3 + 3 \mapsto_v 6$$

in three steps.

An alternative to call-by-value is *call-by-name*. Under call-by-name, the argument to a function is substituted in directly, without first evaluating it. This avoids the problem of unnecessary computation or unforced nontermination: if a value is never required, it will never be evaluated. However, it introduces another problem: the argument to a function will be evaluated separately each time it occurs. Thus

$$(\lambda x . x + x) (1 + 2) \mapsto_n (1 + 2) + (1 + 2) \mapsto_n 3 + (1 + 2) \mapsto_n 3 + 3 \mapsto_n 6$$

in four steps.

To avoid this, implementors developed a third technique, *call-by-need* or *lazy*

evaluation [Wad71, AFM⁺95].² Lazy evaluation has the best of both worlds: it has the semantics of call-by-name, with the efficiency of call-by-value. Under call-by-need, the argument to a function is not evaluated until its value is actually needed, and the resulting value is stored so that if it is needed again the value may be given immediately without further evaluation. Thus

$$(\lambda x . x + x) (1 + 2) \mapsto_l \begin{array}{c} \xrightarrow{1+2} \\ \bullet + \bullet \end{array} \mapsto_l \begin{array}{c} \xrightarrow{3} \\ \bullet + \bullet \end{array} \mapsto_l 6$$

in three steps. Observe that we evaluate only expressions whose value is actually required, as in call-by-name, but we evaluate each expression only once, as in call-by-value.

An important consequence of lazy evaluation is that the program is evaluated in a much less sequential manner than call-by-value or procedural programs. Evaluation is *demand-driven*: an expression is evaluated only when (or if) its value is actually needed, and control flow is much less significant than *data flow*. This leads to elegant formulations of algorithms operating on large or potentially infinite data structures, such as streams or game trees. With attention drawn away from fine details of control flow, the programmer is free to focus on higher-level aspects of the problem's data structures and the relationships between them.

For example, Hughes [Hug89, §5] gives a simple example of a noughts-and-crosses (tic-tac-toe) program that uses alpha-beta pruning to select the best move. His algorithm is neatly expressed as a compositional pipeline:

$$\text{evaluate} = \text{max} \circ \text{maximise}' \circ \text{highfirst} \circ \text{maptree static} \circ \text{prune } 8 \circ \text{gametree}$$

where *gametree* generates the (potentially infinite) game tree, and subsequent stages prune the tree, perform static evaluation at each node, reorder branches at each node to select the best first, and select the best solution using alpha-beta pruning to avoid unnecessary computation. Control flow is left entirely implicit: even though the algorithm is presented directly as an operation on the infinite game tree, during execution the game tree is generated only *on demand*, and computations are performed only for those nodes that are actually required. In fact, the game tree itself may well be entirely eliminated by a subsequent optimisation known as *deforestation* [Wad90a, Gil96].

This freedom from fine details can be seen concretely in the commonly-observed statistic that a program written in Haskell has usually about a tenth of the number of lines as the same program written in C or C++, and may be indistinguishable to the untrained eye from the problem specification [HJ94, §7].

1.2.4 The need for optimisation

However, this freedom to express programs at a very high level does come at a cost. Control flow must be specified somewhere, and if it is not specified by the programmer it must be determined by the compiler. Further, the laziness of evaluation requires significant book-keeping internally, to keep track of delayed computations and

²This is not to be confused with the “lazy” lambda calculus of Abramsky [Abr90], which in fact has a call-by-name semantics. In this thesis, the term “lazy” refers to call-by-need.

results of previous computations. Both of these mean that a compiler for a lazy functional language has a significantly harder task than one for a procedural language, or even a strict functional one (folklore suggests a slowdown of two to ten times relative to C; see [HFA⁺96] for some concrete measurements). In consequence, *optimisation* for lazy functional languages has been a major area of research over the last twenty years or so.

Of the many different approaches to optimisation in these languages, we focus on the implementation of laziness. There are three main ways in which the costs associated with laziness may be reduced.

- First, we can design improved runtime systems that require less book-keeping, or do it faster or in less space.
- Second, we can identify places where call-by-need can be turned into call-by-value, evaluating the argument first and avoiding delaying the computation.
- Third, we can identify places where call-by-need can be turned into call-by-name, substituting the argument in directly and evaluating it along with the function body.

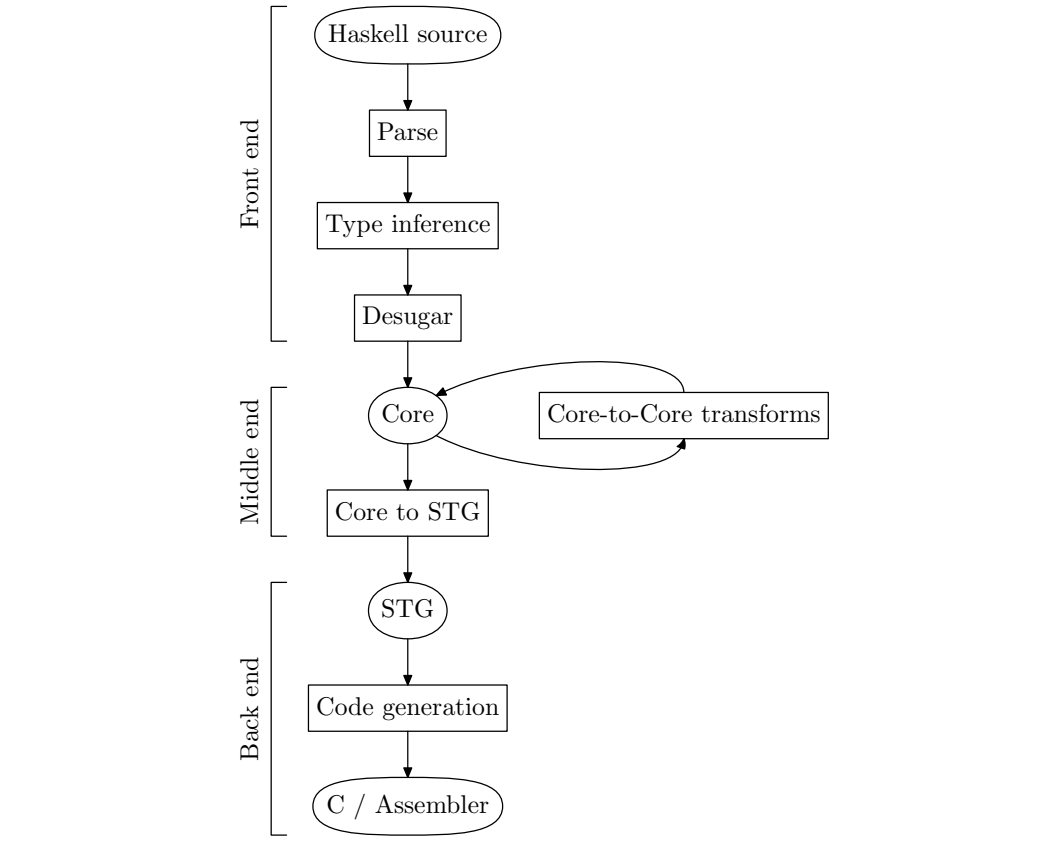
The first approach has led to the design and incremental improvement of a variety of abstract machines for lazy functional languages, some of which are cited in Section 1.3.1 below. The second approach is known as *strictness analysis*, and its large literature is briefly touched upon in Appendix C.6. The third approach is known as *usage analysis*, and is the subject of this thesis.

1.2.5 The Glasgow Haskell Compiler

At this point we take a moment to introduce the compiler which has motivated our research, the Glasgow Haskell Compiler (GHC)³ [PJS98a, PJ92]. GHC is a compiler for the lazy functional language Haskell [HW90, PJH⁺99]. It was developed by Peyton Jones and many others at the University of Glasgow, and more recently at Microsoft Research, Cambridge. It comprises around 70 000 lines of Haskell, written over a ten-year period, and is in active use by thousands of users around the world.

The design aims for GHC are twofold. First, it aims to be the best available compiler for Haskell: it accepts the entire standard Haskell 98 language [PJH⁺99] along with the commonly-accepted extensions, it generates efficient code, it is reasonably fast, it gives high-quality error messages, and it is actively maintained. Second, it aims to be a testbed for investigation of new optimisation techniques and language features: it is open-source, it is substantially commented and documented, and its design is modular. Both aims appear to have been achieved: GHC is the *de facto* standard Haskell compiler, and in recent years a host of optimisations and language features have been added, including deforestation [Gil96], multi-parameter type classes [PJJM97], functional dependencies [Jon99], implicit parameters [LSML00], restricted existential quantification, concurrency [PJGF96],

³Available from <http://www.haskell.org/ghc/>.

Figure 1.1 The architecture of the Glasgow Haskell Compiler.

asynchronous exceptions [MPJMR01], and more. Thus GHC was a natural target for our analysis and optimisations. Our actual experience is described in Chapter 6.

The architecture of GHC is depicted in Figure 1.1. Haskell source is first parsed, typechecked, and syntactic sugar is removed, converting it into the compiler’s intermediate language, *Core*.

Core is a small functional language, technically an explicitly-typed polymorphic lambda calculus in the style of Girard–Reynolds [Gir72, Rey74] with the addition of *letrec* (for binding, sharing, and recursion) and constructors and case (for algebraic data types). Crucially, *Core* has an operational semantics stating that first, *heap allocation is performed by letrec and only by letrec*, and second, *evaluation is performed by case and only by case*. These two facts greatly simplify the evaluator, but they also aid us in understanding the operational behaviour of a program. It is this operational interpretation of the intermediate language that allows us to design a usage analysis at the level of the intermediate language, without having to descend into the back end of the compiler.

At the *Core* level, GHC performs a large number of optimisations, each a source-to-source transformation. All transformations preserve types, and indeed many are type-directed. Some of the more important are specialisation, the worker/wrapper transformation that takes advantage of strictness and absence information, full lazy-

ness, and floating in, along with many local optimisations wired into the simplifier and performed repeatedly between each major transformation pass. This structure is described in more detail in [PJS98a].

Once Core optimisation is complete, the code is transformed into a low-level code called *STG* [PJ92], from which either C or assembly code is generated for execution.

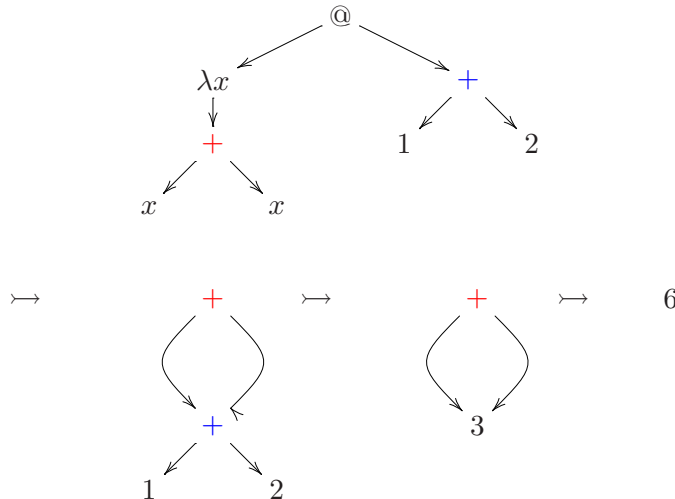
GHC is described in more detail in the references cited. Core is an intermediate language and for a long time had no standard textual representation, but recently Tolmach has developed a standard for *External Core* [TGT01], including both a syntax and an operational semantics.

1.3 Usage

Obviously the central concept in usage analysis is *usage*. But what exactly is usage? We answer this question below.

1.3.1 Thunks and updates

In Section 1.2.3 above we illustrated lazy evaluation by drawing shared expressions separately from the main expression, with blobs and arrows indicating sharing. This intuition is formalised by the notion of *graph reduction*, due to Wadsworth [Wad71]. An expression is represented by a *graph* in which edges denote pointers and nodes denote combinators (primitives, applications, and abstracted expressions). Sharing is now trivially indicated by having multiple pointers to a single node. *Reduction* rewrites the graph, repeatedly replacing a graphical redex with its contractum. For example, the reduction sequence of the expression $(\lambda x . x + x) (1 + 2)$ we saw earlier is as follows:



This model turns out to be quite inefficient, however, since the evaluator spends a lot of time walking up and down the graph locating the next redex and then copying a subgraph. A number of more efficient machines for lazy evaluation have been

developed, including the G-machine [Joh87, Aug87] and the Three-Instruction Machine [FW87] (see [PJL92] for a discussion and further references), culminating in the *STG-machine* model of Peyton Jones [PJ92], implemented in the Glasgow Haskell Compiler.

In this thesis we work with an abstract version of the STG-machine, described in detail in Chapter 2. The evaluator works with a *heap* which contains the graph, and a *stack* which contains the current context of evaluation. Delayed computations (such as $1 + 2$ in our example) are placed on the heap as *thunks* (also known as suspensions; the name “thunk” is due to [Ing61]),⁴ and only evaluated when demanded. Once a thunk has been evaluated, it is overwritten or *updated* with the resulting value, so that subsequent accesses will return this value immediately without further computation. Thus our example proceeds as follows (omitting many details):

<u>Stack</u>	<u>Heap</u>	
$(\lambda x . x + x) (1 + 2)$		
$\rightarrow x' + x'$	$x' : \boxed{1 + 2}$	build thunk
$\rightarrow \underline{1 + 2}, \quad x' + x'$	$x' : \boxed{1 + 2}$	demand x'
$\rightarrow \underline{3}, \quad x' + x'$	$x' : \boxed{1 + 2}$	evaluate
$\rightarrow \underline{3 + x'}$	$x' : \boxed{3}$	update thunk and return
$\rightarrow \underline{3 + 3}$	$x' : \boxed{3}$	lookup value
$\rightarrow \underline{6}$	$x' : \boxed{3}$	evaluate

It should be clear from this example how laziness is achieved: if a thunk is never demanded, it is never evaluated; if a thunk is demanded more than once, subsequent accesses yield the value computed by the initial demand.

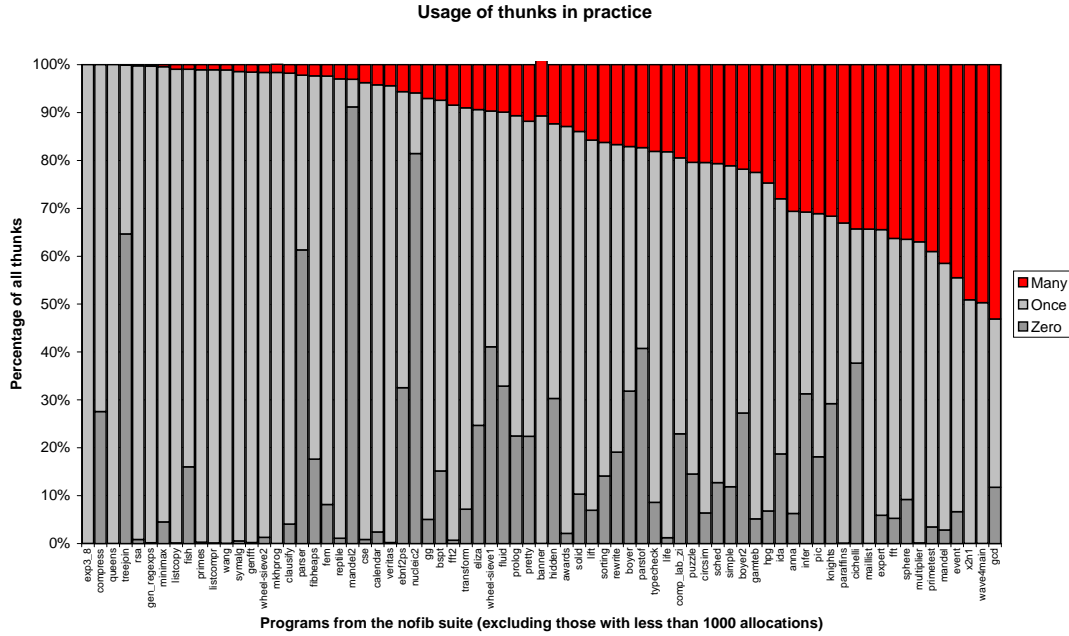
1.3.2 Update avoidance: the opportunity

This is all very well, and the STG machine is certainly an efficient implementation of lazy evaluation, but there is an important source of inefficiency: sometimes laziness is not necessary (Section 1.2.4). Consider the evaluation of a slightly different term, $(\lambda x . \lambda y . x + x + y) (1 + 2) (3 + 4)$. Here the value of x is demanded twice, but the value of y is demanded just once. Whereas the update of the thunk $1 + 2$ with the value 3 saves work the second time x is demanded, the update of the thunk $3 + 4$ with 7 is wasted. Is this a significant problem, and if so, can we avoid it by not performing unnecessary updates?

Marlow [Mar93] measured the number of updates performed and actually required during execution of some example programs compiled with GHC. He determined that on average, only 30% of all updates performed were necessary, and that avoiding the remaining 70% could result in roughly a 20% performance increase, due to a reduction in administration related to updates, avoiding the updates themselves, and benefits to the heap allocator and garbage collector.

To obtain more detailed results for ourselves we modified our Haskell compiler, GHC, so that it generated code to record the proportion of all allocated thunks that are demanded at most once (see Section 6.7.2). The necessary modifications are

⁴The amusing origin of the term is given in the eponymous entry of [Ray91].

Figure 1.2 The opportunity.

rather simple: we need only to track the total number of thunks allocated and, for each thunk, whether it is entered never, once, or more than once. We compiled and ran the entire NoFib suite, a large suite of around 50 benchmark programs ranging from tiny ones up to 10 000 line applications [Par93].

Figure 1.2 gives the results, which amply support the folklore and are consistent with the data of [Mar93]. Each bar is a program from NoFib, with thunks allocated divided into those used more than once, exactly once, and not at all during execution. In *every program but one*, the majority of thunks are demanded at most once, and hence do not need to be updated. In more than a third of the programs, over 95% of thunks are never entered more than once! There is clearly huge scope for optimisation here (although we discuss in Section 6.9.2 one reason such statistics may be misleading).

1.3.3 Usage analysis

In order to avoid unnecessarily updating a thunk, we must obtain information on the number of times it will be demanded. We refer to this as its *usage*. Since we have only two alternatives – to update the thunk, or not to update it – we need distinguish only two different usages: *used-at-most-once* and *possibly-used-many-times*, denoted 1 and ω respectively [TWM95a]. If a thunk is demanded just once, or not at all, then we may annotate it 1. If a thunk may be demanded twice, three times, or some unbounded number, or if we do not know how many times it will be demanded, we must annotate it ω . Then a thunk annotated 1 may safely not be updated, but a

thunk annotated ω must be updated. In the above example, we would like the thunk for $1 + 2$ to be given the annotation ω , but that for $3 + 4$ to be given 1. We sometimes refer to thunks that are not updated as *single-entry* thunks.

A usage analysis is simply an algorithm that infers such annotations statically for all thunks in a program (more precisely, for all thunk creation points); our quest in this thesis is to design a *good* usage analysis, which infers 1 for as many annotations as possible.

More generally, usage applies to values as well. A function is used each time it is applied to an argument, and a value is used each time it is passed to a primitive operation like $+$, or inspected and decomposed in a case statement. We refer to a function that is applied at most once as a *one-shot lambda*.

Other annotation domains are possible: it is sometimes useful to distinguish between used-once and not-used, for example, or one may distinguish all seven distinct nonempty subsets of $\{0, 1, > 1\}$. We consider these possibilities speculatively in Appendix C.

1.3.4 Optimisations enabled by usage information

Update avoidance is not the only optimisation that is enabled by usage information. Below we consider several others.

Inlining inside lambdas. Consider the expression

$$\text{let } x = e \text{ in } \lambda y . \text{ case } x \text{ of } \dots \quad (1.1a)$$

and suppose that x does not occur anywhere in the case alternatives. We could avoid the construction of the thunk for x entirely by inlining it at its (single) occurrence site, thus:

$$\lambda y . \text{ case } e \text{ of } \dots \quad (1.1b)$$

Now e is evaluated immediately by the case, instead of first being allocated as a thunk and then later evaluated by the case. In general this transformation is a disaster, because now e is evaluated as often as the lambda is applied, and that might be a great many times. If we could prove that the lambda was applied at most once, and hence that x 's thunk would be evaluated at most once, then we could safely perform the transformation [PJM99].

Floating in. Even when a thunk cannot be eliminated entirely, it may be made less expensive by floating its binding inwards, towards its use site. For example, consider:

$$\text{let } x = e \text{ in } \lambda y . \dots (f (g x)) \dots \quad (1.2a)$$

If the lambda is known to be one-shot (called at most once), it would be safe to float the binding for x inwards, thus:

$$\lambda y . \dots (f (\text{let } x = e \text{ in } g x)) \dots \quad (1.2b)$$

Now the thunk for x may never be constructed (in the case where f does not evaluate its argument); furthermore, if g is strict then we can evaluate e

immediately instead of constructing a thunk for it. This transformation is not a *guaranteed* win, because the size of closures can change, but on average it is very worthwhile [PJPS96].

Full laziness. In order to share their evaluation between successive applications, invariant subexpressions may be hoisted out of functions [PJL91]. This is exactly the opposite of the inlining and float-in transformations just discussed above; for example, (1.2b) would be transformed into (1.2a). As we have just seen, though, hoisting a subexpression out of a function that is called only once makes things (slightly) worse, not better. Information about the usage of functions can therefore be used to restrict the full laziness transform to cases where it is (more likely to be) of benefit.

The latter two transformations were discussed in detail in [PJPS96], along with measurements demonstrating their effectiveness. That paper pessimistically assumed that every lambda was called more than once. By identifying one-shot lambdas, usage information allows more thunks to be floated inwards and fewer to be hoisted outwards, thus improving the effectiveness of both transformations.

To summarise, we have strong reason to believe that accurate information on the usage of subexpressions can allow a compiler to generate significantly better code, primarily by relaxing pessimistic assumptions about lambdas. A great deal more background about transformations used in the compilation of lazy languages is given in [San95, Gil96, PJPS96, PJ96].

More formally, Gustavsson and Sands [GS00a] in their work on space behaviour of lazy functional languages define *work-safety* and *space-safety* of program transformations (work safety was introduced informally in [PJS98b, §4.1]). They define a *use-once-don't-drag* property (first named in [GS01a]), which is slightly stronger than used-at-most-once in that after its single use, a thunk having the property not only must not be used again, but must not be referred to in any live binding (even if this reference is never evaluated). They show that this property is sufficient to guarantee the work- and space-safety of the inlining transformation. They (correctly, Section 3.6.1) conjecture that our analysis has this property.⁵ Gustavsson argues in [Gus98, §9.1] that not dragging is also important in an update-avoiding implementation using garbage collection, because otherwise the collector might attempt to follow a dangling pointer; preventing this would require a process similar to black-holing (Section 2.4.2).

1.3.5 Usage analyses: related work

There has been little work on usage analyses until recently. The earliest work of which we are aware is that of Goldberg [Gol87], who used an abstract interpretation to derive information on the sharing of partial applications during call-by-need

⁵But note that the extended usage analysis of Appendix C, like the analyses of Mogensen [Ses91, Mar93, Mog97a] and others that address 0-usage, do not avoid dragging and thus do not guarantee space-safety of inlining. Whether they guarantee work-safety of inlining is an open problem, but Gustavsson and Sands conjecture that they do and that a proof along their lines would be straightforward [GS01a, §6].

evaluation of a program. This information can be used to optimise the generation of supercombinators, essentially implementing the full laziness transformation described above and in [PJL91].

Abstract interpretation and flow analysis have been used for a number of other usage or sharing analyses, notably Sestoft’s usage-interval analysis [Ses91, c.5] (discussed further in the appendix, Section C.6), Johnsson and Boquist’s sharing analysis for GRIN [BJ96, §5], and Marlow’s update avoidance analysis for GHC [Mar93].

Marlow addresses precisely the same problem as this thesis, that of identifying thunks used at most once during execution in programs compiled by GHC. His analysis is an abstract interpretation, with abstract values consisting of triples

$$AbVal = \mathbb{N}^{Var} \times \mathbb{N}^{Var} \times (AbVal \rightarrow AbVal)$$

The first component of the triple is a multiset of closures (variables) that will be entered (used) when the expression is evaluated. The second component is a multiset of closures that may possibly be referenced when the object is applied (if a function) or deconstructed (if a constructor). The third component is the abstract function from abstract argument to abstract result, relevant only if the object is a function. This information is sufficient to determine whether a given closure in a program should be marked updatable or non-updatable.

An abstract interpretation is given for a language equivalent to our full language FL_0 , and it was implemented in GHC. The results are very good, ranging from 7–72% of unnecessary updates avoided on a range of real programs from the NoFib test suite. However, the method chosen is extremely expensive for some programs, because the information stored for a given expression is unbounded. The treatment of recursion involves a fixpoint calculation in an infinite domain, which must be approximated by an arbitrary small number of iterations (Marlow performs just one iteration). The analysis also has no proof of soundness, although the success of the implemented analysis (which performs runtime checks to ensure non-updatable thunks are never re-entered) is a strong argument for soundness.

Turner *et al.* [TWM95a] point out that Marlow’s analysis is overly conservative, giving an example of a term in which their analysis (and hence our analysis of Chapter 3, which is strictly more powerful) is able to mark a thunk non-updatable where Marlow’s is not.

Gill [Gil96, pp. 72ff] presents a very simple set of “types” that are able to denote used-once arguments and non-shared application; “types” must be written by hand and justified intuitively, since no type rules or inference are given.

In 1987, Girard introduced *linear logic* [Gir87, Wad93b, Gir95]. Linear logic is a resource-conscious logic, in which each proposition may be used only once. Wadler [Wad90b, Wad91, Wad93a] proposed that linear logic could be used to define a type system for pure functional languages in which the usage of values is precisely controlled. Such a type system would allow (linear) values to be updated in place, avoiding copying and garbage collection.

Inspired by Wadler and others, a number of type-based analyses appeared capturing usage, single-threading, and related notions. Guzmán and Hudak [GH90] give a type-based analysis distinguishing seven combinations of mutability, sharing, and linearity properties for a call-by-need functional language with mutable

arrays; the type system is presented in classical style, augmented with a “liability” which tracks the usage of free variables. Launchbury *et al.* [LGH⁺92] give a linear-style type system which infers annotations from the set $\{Zero, Once, Many\}$, where *Once* is interpreted to mean at most once. They have no proof of correctness, and no inference. Bierman [Bie91, Bie92] proposed a type system taking annotations from a seven-point lattice subsuming linearity, call-by-name (absence / usage), call-by-value (strictness), and call-by-need (see the appendix, Section C.3.2 for further discussion of the Bierman lattice). Wright and Baker-Finch [WBF93] give a type-based analysis with intersection types for a call-by-name language, related to relevant logic [Dun86], with usage-count annotations taken from the natural numbers. Wright’s later analysis [Wri96], also linear-style and for call-by-name, is parameterised by an annotation algebra and may be instantiated as a linearity analysis, a strictness analysis, or one of three flavours of usage analysis (affine, affine-plus-linear, and linear with zero).

Maraist, Odersky, Turner, and Wadler observed [MOTW95] that lazy (call-by-need) evaluation corresponds to *affine* linear logic [Jac94]. Together with Mossin, the latter two authors proposed *Once Upon a Type* [TWM95a, TWM95b], the analysis upon which this thesis is based.

Once Upon a Type proposes a type-based usage analysis for a call-by-need language, with annotations taken from the set $\{1, \omega\}$. They present a natural (big-step) semantics for call-by-need based on [Lau93] but with rules that enforce correctness of usage annotations. Their well-typing rules are based on linear logic, but permit arbitrary weakening, and contraction only for variables annotated ω . The judgements are decorated with a set of constraints. The system is proven sound with respect to the natural semantics, and an inference is presented.

The key limitations of this work are an issue we term *poisoning* and address in Section 3.3.5, which leads to extremely poor results in practice, and the lack of treatment of type polymorphism or algebraic data structures other than lists, which we address in Chapter 5. The analysis is also usage-monomorphic. Nonetheless, this work formed the foundation of our own, and we owe much to the structure of its type system and soundness proof. The analysis of Chapter 3 is a slight but strict extension to *Once Upon a Type*, although there are differences in presentation.

A number of researchers other than ourselves were inspired by this research and set to work improving it: Faxén [Fax95, Fax97], Mogensen [Mog97a, Mog98], and Gustavsson [Gus98, Gus99, GS00b, Gus01a]. We refer to the work of these authors in detail where relevant throughout the thesis.

Faxén [Fax95, Fax97] gives a type-based flow analysis for a functional language with explicit evals and thunks. From the results of the flow analysis he is able to derive, *inter alia*, sharing information. The analysis uses subtyping to model flow, and constrained polymorphism to capture polyvariance. The second analysis [Fax97, c.5] performs generalisation at λ and instantiation at applications, rather than the standard approach due to Milner [Mil78] of generalisation at let and instantiation at variables followed in this thesis; undecidability is sidestepped by *ad hoc* approximation [Fax97, §5.2.4].

Mogensen [Mog98] extends the analysis of [TWM95a] to include more general recursive data types, and observes that to give good types to programs involving

selection functions such as *fst* and *snd* it is necessary to incorporate a zero annotation as well as 1 and ω . Furthermore, Mogensen incorporates subsumption, thus avoiding the problem of poisoning. The analysis does not possess a proof of correctness; indeed, we identified several errors in the published version of the system [Mog97a]. However, our use of subsumption was inspired by this work.

Gustavsson [Gus98, GS00b] has developed a series of usage analyses based on [TWM95a], but incorporating an additional optimisation, *update marker check avoidance*. Subsumption and constrained usage polymorphism are included, and the systems are all proven sound. More details can be found in the related work sections of Chapters 2, 3, 4, and 5.

Finally, Kobayashi [Kob99] addresses the problem of update-in-place or immediate garbage collection for call-by-value languages, and refines the linear approach by adding an annotation δ in addition to the usual 0, 1, ω . Kobayashi annotates heap allocation and access. A heap access (variable dereference) may be annotated δ if the variable is accessed but not held onto or returned; a variable may still be bound linearly (and annotated 1) even if it is referenced several times at δ and then finally at 1, since by this point there are no other live references to this variable. In other words, Kobayashi's type system takes the order of evaluation into account. In the type system, annotations appear on pairs, on functions, and on function arguments. Subtyping is permitted, and is implemented by means of a one-bit tag embedded within a heap pointer denoting whether the value pointed to is annotated 1 or ω . However, Kobayashi observes that curried functions present a problem [Kob99, §9].

The only *practical* experience above is by Marlow and Kobayashi; Johnsson and Boquist, Gustavsson, and Faxén have implemented prototypes and experimented with small programs. For this reason, the practical experience obtained in the present work (Chapter 6) is very important.

Other surveys of usage analysis can be found in [GS01a, §5] and [Ses91, §5.1.6].

1.4 Program analysis

So, we want to design a usage analysis. But what is an analysis, and what sort of analysis should we use? Nielson *et al.* define *program analysis* as follows:

Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at runtime when executing a program on a computer. [NNH99, p. 1]

They then identify four major analysis techniques: data flow analysis, constraint-based analysis, abstract interpretation, and type- and effect-based analysis. Briefly, these can be described as follows.

Data flow analysis. A data flow analysis considers the program as a graph, with nodes representing points of the program and edges representing control flow. It sets up a system of equations relating the state of the program at each point to its state at other points. The least solution of this system of equations yields the desired approximation to the state of the program at each point.

Constraint-based analysis. Constraint-based analysis extracts from a program a set of *constraints* describing the way in which the space of possible behaviours is constrained by each part of the program. The constraints are inequalities or set inclusions. Again, the least solution of this system of constraints yields the desired approximation to the behaviour of the program.

Abstract interpretation. An abstract interpretation defines an abstract domain, interprets the semantics of the program over this domain, and then abstractly executes the program in this domain. The abstract domain approximates the actual domain used at runtime in such a way as to make execution deterministic and the behaviour of the program decidable. The result of the abstract execution is related to the result of actual execution by an approximation relation or *Galois connection*.

Type- and effect-based analysis. A type- or effect-based analysis defines a system of *annotated types* which add additional information about the behaviour of the program to the existing types of the language. The analysis consists of a set of *well-typing rules* which allow the annotated type of a term to be derived. A well-typing may be obtained by conventional type inference, or by generating a set of constraints to be solved subsequently. The resulting types of the program and subterms yields the desired information.

There are other, more *ad hoc*, approaches to program analysis, but these are the main systematic ones.

1.4.1 Type-based analysis

The language we work with comes from the so-called HOT (Higher-Order, Typed) class of languages [Wad97]; specifically, we work with a typed intermediate language in a HOT language compiler. Of the four approaches listed above, type-based analysis best fits the type-based nature of these languages. A type-based usage analysis can be tightly integrated with the rest of the compiler, giving benefits in both directions: the analysis can readily make use of the existing type manipulation machinery for computing and maintaining its usage information, and the rest of the compiler can readily access the results of the analysis.

Furthermore, type-based analyses are extremely flexible. In consequence, they have become very popular, being applied to a vast range of problems, ranging from locating Y2K errors in legacy COBOL [EHM⁺99] to avoiding dynamic memory allocation and garbage collection in ML [TT94], amongst many others.

We note in passing that these apparently-distinct analysis techniques are often formally equivalent [NNH99, §1.1] Thus, for example, Jensen [Jen91] and others [PS96, PS92, Mon93] have shown that abstract interpretation is equivalent to type-based analysis with intersection types, and that polyvariance (or “splitting”) in data flow analysis is equivalent to polymorphism in type-based analysis [JW95]. Nevertheless, the techniques have differences of emphasis and flavour. We focus on a type-based analysis, but use a constraint-based approach to type inference.

1.4.2 Limitations of program analysis

Program analyses were defined at the beginning of this section as static techniques that yield safe and computable approximations. We might hope for better than this: why must we settle for approximations to the truth?

Unfortunately, no static analysis can be exact. The reason is that a static analysis must be an *algorithm* that *always terminates*, and so it can only compute *decidable* properties. Most interesting properties, however, relate to the dynamic behaviour of the program and are undecidable, as can easily be established by a reduction along the following lines (established here for usage analysis):⁶

Theorem 1.1 (Incompleteness for usage analyses)

No usage analysis can be both sound and complete.

Proof Assume (for the sake of contradiction) that we have a sound and complete usage analysis. Then let P be a program, and construct the program P' which binds x , uses it once, runs P , then uses x a second time. Ask the usage analysis what is the correct usage annotation for x ; if it says ω then the program halts, if it says 1 then the program does not halt. Thus we have solved the Halting Problem. This is a contradiction, and so no complete usage analysis can exist. \square

Thus we must accept an approximate analysis.

1.5 Designing a type-based usage analysis

Having elected to design a type-based analysis, we must now consider what is required. In the present section we set out the general structure of a type-based analysis, and then examine some specific considerations.

1.5.1 Structure

A type-based analysis begins with a *source language* in which are written the programs it analyses. This source language will be typed, and its underlying type system will form the framework from which we will hang the annotations used by our analysis.

We must also have a *semantics* that defines the property we are interested in. Without a formal semantics we can only appeal to the reader's intuition to justify our analysis. The semantics we give in this thesis is an *operational* one, reflecting the intensionality of our definition of usage (Section 1.3.3). It defines usage by allowing the program to be annotated with usage information, and *going wrong* [Mil78, WF94] (Section 1.2.2) if this information is incorrect.

⁶Rice's Theorem [Ric53] [DM58] [HU79, §8.4] establishes a similar result for properties of languages recognised by Turing machines, but despite the intuitive similarity it is not clear how this relates to intensional dynamic properties of program behaviour. For this reason, we give a direct proof.

The analysis is called type-based because information about the property we are interested in is carried by the *types*. Therefore, we must define the structure of these annotated types, based on the underlying types of the source language. It is convenient to work with an explicitly-typed language, and so the source term language must also be extended to use the new annotated types, yielding a *typed target language*.

The core of the analysis is a set of *well-typing rules* that extend the underlying well-typing rules of the source language so as to determine the correct values of the type annotations. The rules define the relationships between types and terms, and what is a valid typing for any given program; they encode our knowledge about the dynamic behaviour of the program in a way that allows some of it to be inferred statically.

In many cases the well-typing rules will admit more than one typing for a given program (for example, an approximating analysis may admit the typing “don’t know” for all terms). In this case, a *goodness ordering* must be provided that defines which is the *best* one. For well-behaved systems this will be the *principal* type or typing [Jim96], but for systems that lack principal types another ordering is required.

The well-typing rules are only able to tell us if a typing is correct or not; in general they do not directly provide us with a typing for an untyped (or only source-typed) source term. We must therefore provide an *inference algorithm* that computes the best valid typing for any given source program.

These six (source language, semantics, typed target language, well-typing rules, goodness ordering, and inference algorithm) together define a *type-based analysis*. However, a type-based analysis should also be justified by a number of proofs. Although the analysis cannot be complete (Theorem 1.1), we require that it be *sound*: analysis results must be safe approximations of the true values. We require several other properties: all source programs must have a valid typing (*i.e.*, no program may be rejected; see Section 1.5.5), the inference algorithm must compute a valid typing for all source terms that possess one (*i.e.*, the inference algorithm must be sound), and it must compute the best if there is more than one (a kind of inference completeness result). It is also useful to have a complexity bound for the analysis.

The analyses developed in this thesis, Chapters 3, 4, and 5, all fit within this framework; the semantics given in Chapter 2 is presented as a trivial usage analysis, and the speculative system of Appendix C lacks only the inference algorithm, goodness ordering, and proofs.

1.5.2 Practical considerations

Two important practical considerations in the design of a usage analysis are its efficiency of execution and its ease of implementation.

To be practical, a usage analysis intended to be added to an existing compiler should not increase compilation times significantly for programs of average size. While GHC (Section 1.2.5) is not an extremely rapid compiler, it is still fast enough for programmers to use the normal edit-compile-test-debug cycle: a module is typically compiled in a few seconds. An analysis that noticeably worsens this turnaround time will not be popular with users, and significant effort should be put into both al-

gorithmic optimisations and fine-tuning to minimise the cost of the analysis.

Less obviously, the analysis should be easily implementable as an extension to the existing compiler. While GHC was designed as a “motherboard” [PJ92, §11] into which analyses and optimisations can be plugged, not all extensions are created equal. Those that are predominantly *local*, dealing with particular fragments of code at particular phases during optimisation, are the easiest to implement. Those that act globally, collecting information about the whole module (or worse, the whole separately-compiled program, Section 1.5.6), are much harder.

Another potential source of difficulty in implementation is the level at which the analysis is embedded: there is a spectrum from recomputing information each time the program is inspected by the analysis, through adding information as explicit term or type annotations, all the way to extending the abstract syntax of terms and/or types to incorporate the new information. While the last alternative is theoretically the cleanest, most readily distributes the inferred information throughout the compiler, and provides the best guarantees that information remains consistent during optimisation, it is also by far the hardest to implement, requiring substantial and widely-distributed changes to the code base. We attempted a number of different embeddings, and our experiences are documented in Section 6.2.

For practical reasons, then, we must ensure that our usage analysis does not increase compilation times significantly, and that it is not too difficult to implement.

1.5.3 Performance measurement

Recall once more (Theorem 1.1) that our usage analysis will necessarily be incomplete. Since we cannot hope to identify *every* used-once thunk, we must surely hope to identify *as many as possible*. This is an experimental question, and so we must set up a scenario in which to test our analysis and some metrics by which we may measure its performance.

As a typical use scenario, we take the NoFib suite of benchmarks [Par93]. This is a suite of around 50 programs ranging from toy benchmarks of a few lines (*primes*, *nqueens*) through to substantial applications (*anna*, *veritas*, *HMMS*). Performance of GHC is regularly compared against this suite in regression testing, so the compiler should already perform well on them – this presents us with a hard target, since other analyses such as strictness may be competing with usage analysis to optimise the same part of a program.

Our initial metric is the most obvious, as we have seen already in Section 1.3.2. We measure the total number of thunk allocations performed during evaluation, and track the subsequent usage of each, breaking the numbers down into the numbers used more than once and at most once. We also measure the number of identified used-once thunks allocated. Comparing these gives us a measure of the effectiveness of the analysis: the proportion of used-once thunks that are identified.

This metric is not ideal, however, since we are not targeting only update avoidance. Section 1.3.4 listed a range of other optimisations we seek to enable; many of these are aimed at reducing the total number of allocations performed by the program, and hence affect our statistics. We compare allocations and average execution time to measure the effect of the other optimisations. Execution time is a particu-

larly unstable metric due to timing variations, swapping, and cache behaviour; the folklore suggests that total allocations is a prime indicator of running time, and this is easy to measure precisely.

During development, a number of informal metrics were used as a guide, in order to verify that the analysis was on the right track and to isolate problems. For example, the types shown in Figure 4.1 for some standard Haskell functions are the ones we would intuitively expect; it was clear from the types obtained for the same functions in the earlier analysis that something was not right, and this led to the development of simple polymorphism (Section 4.1). Another useful approach was to examine the annotated output from toy programs like `primes` or `nqueens` and compare the inferred results with a hand-simulation of their execution. This led to a number of advances in the analysis, notably the finer treatment of data structures (Section 5.4), and the use of specialisation (Section 6.3.3).

1.5.4 Proof techniques

Since we work with an operational semantics and seek to prove a soundness result, the natural approach is to use the syntactic proof technique developed by Wright and Felleisen [WF94]. The difficulties involved with traditional soundness proofs based on denotational semantics or structural (big-step) operational semantics are well argued in this paper, and we do not repeat them here. Wright and Felleisen's technique involves giving a *small-step* semantics for the language, such that each intermediate step in the evaluation of the program is itself a program, and showing that evaluating the program one step preserves its type. This result is called *subject reduction* in the paper, but we refer to it as *progress*, proving also that a well-typed non-value can always reduce.

1.5.5 Soft typing

In [Fag90, CF91], Fagan and Cartwright introduce *soft typing* as a way of bringing the benefits of static typing to dynamically-typed languages such as Scheme, without restricting their expressiveness. A soft type system can statically type *all* programs, by inserting explicit runtime checks around the arguments of applications not provably type-correct. The resulting program is statically well-typed, and the programmer can detect errors before execution by inspecting the inserted checks. They define two criteria for a soft type system to be effective and practical: first, the programmer must not be required to supply *any* type information, and second, few unnecessary checks should be inserted.

A type-based usage analysis must similarly be designed in such a way that *all* input programs can be typed, and that no type information need be provided by the user. In this sense, our analysis must be *soft*. However, rather than inserting runtime checks at each application site a soft usage analysis collects information from all sites and gives the function a type general enough to cover each. Thus we trade worsened precision of the inferred types for guaranteed freedom from runtime violations.

1.5.6 Dealing with separate compilation

Large programs are commonly broken up into multiple *modules*, which are each compiled separately and then linked together to form the final executable. This improves modularity (different parts of the program are in different modules) and compilation speed (only part of the program need be recompiled when one module is changed). However, separate compilation makes program analysis more difficult: the analysis can see only part of the program at any one time.

In the languages we consider in this thesis, a module is essentially a set of letrec bindings along with a *signature*, a set of exported variables with their types. When one module imports another, the signature variables of the imported module are added to the initial typing environment in which the present module is compiled.

Ideally, we would perform our usage analysis for the whole program at once, with all modules combined, generating code appropriate to the actual uses each part makes of the others. As is usual, however, we instead compile each module *separately* for speed and convenience.

Gustavsson [Gus01a, p. 4] observes that “Whether an argument is used at most once may depend on the entire program, so usage analyses are inherently global.” This global nature makes dealing with separate compilation rather more difficult for usage than for many other analyses. To perform usage analysis in conjunction with separate compilation, we must

- compute the usage annotations of the module being compiled in the absence of any information on the usage of exported variables,
- convey usage information from the analysis of one module to that of the next by means of usage types in signatures, and
- ensure that the usage types we provide for exported variables permit *any possible use* by modules that import the present one.

We call the last condition *maximal applicability*. These conditions ensure that all programs may be successfully analysed, and that the results of the analysis of individual modules, when combined, provide a sound analysis of the entire program.

Table 1.1 Languages in this thesis.

L_0	simplified source language	§2.2	Fig. 2.1
LIX_0	simplified instrumented executable language	§2.3	Fig. 2.3
LIX_1	monomorphically usage-typed language	§3.2	Fig. 3.1
LIX_2	polymorphically usage-typed language	§4.2	Fig. 4.2
FL_0	full source language	§5.1	Fig. 5.1
$FLIX_0$	full instrumented executable language	§5.1.4	Fig. 5.3
$FLIX_2$	full poly usage-typed language	§§5.2,5.3	Fig. 5.5
ELX	extended executable language	§C.3	Fig. C.1
EL_1	extended usage-typed language	§C.4	Fig. C.5

1.6 How To Read This Thesis

The thesis develops a simple-polymorphic usage analysis for a real language, proceeding in several stages.

Chapter 1 introduces the thesis, and establishes the context: lazy functional languages, usage, and type-based analysis.

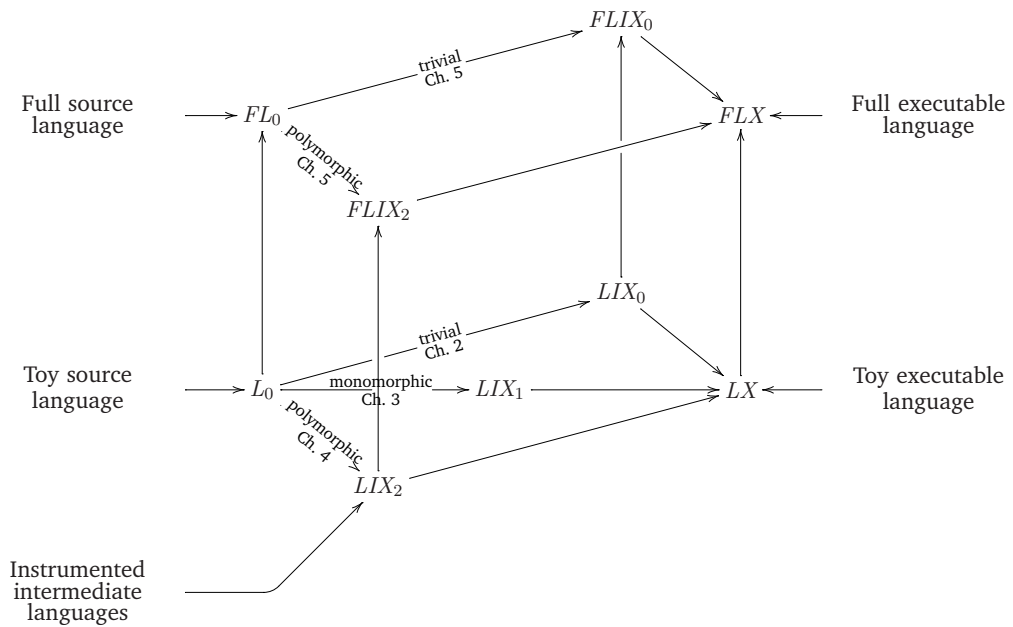
Chapter 2 introduces the formal framework for the rest of the thesis, giving an operational semantics (with a notion of usage) to a toy language, L_0 . This language serves as source language while we develop the basic components of a usage analysis in Chapter 3. That (monomorphic) analysis is insufficient in practice, and so in Chapter 4 we develop *simple polymorphism* in order to address the problems, still working with only the toy language so as to avoid unnecessary clutter.

Chapter 5 moves on to the full language, FL_0 , adding the key features of type polymorphism and user-defined algebraic data types. Since we are now in a position to conduct a trial by fire, Chapter 6 implements the analysis in GHC and measures the results. This ends the main body of the development, and we conclude in Chapter 7.

Appendix A gives a summary of the notations used in the thesis, and Appendix B collects all the definitions, well-typing rules, and inference rules for the full language. In Appendix C we consider a speculative future direction, investigating finer distinctions of usage than simply 1 vs. ω . Deferred proofs appear in Appendix D.

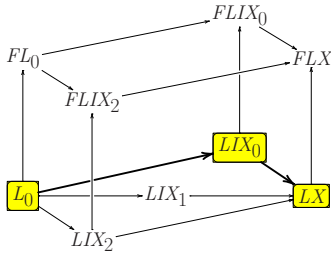
The structure of the thesis reflects the structure of the languages we develop, as illustrated in Figure 1.3. Chapter 2 lays the foundation, trivial analysis for the toy language. Chapter 3 contains the monomorphic analysis for the toy language. Chapter 4 presents the polymorphic analysis. Chapter 5 then extends these results to the full language. This illustration is repeated at the head of each chapter, appropriately highlighted to indicate the position of the chapter within the whole. The various languages are summarised in Table 1.1, each with the location of its definition.

Figure 1.3 Thesis structure.



Chapter 2.

Formal Framework



Our intuitive understanding of usage and usage analysis must be made formal in order to provide a framework in which we can define various usage analyses and examine their soundness and power. In this chapter we construct the formal machinery that will be required in the remainder of the thesis.

The presentation in this chapter is designed to generalise to the languages and analyses of later chapters; thus we occasionally use constructions that are slightly more complex than the reader may expect.

The development is mostly standard, and the eager reader may proceed to Chapter 3. In summary, this chapter introduces a simply-typed lambda calculus L_0 (with letrec and conditional) and translates it into its type erasure LX , giving the latter an abstract machine semantics based on that of Sestoft [Ses97] and proving a type soundness result. The novelty of LX is its use of *update flags* on bindings and lambda terms (Section 2.3.1), in order to control updates and copying, *i.e.*, usage.

2.1 Introduction

In this chapter, we define the following:

1. a *source language* (Section 2.2), in which the programs we analyse are written;
2. an *executable language* (Section 2.3), with a notion of usage information, on which the analysed programs will be executed;
3. a *trivial translation* (Section 2.3.3) which takes source programs into the exe-

Figure 2.1 The source language L_0 .

Terms	$M ::= A$	atom
	n	literal (integer)
	$\lambda x : t . M$	term abstraction
	$M A$	term application
	$M_1 + M_2$	primop (addition)
	$\text{if0 } M \text{ then } M_1 \text{ else } M_2$	zero-test conditional
	$\text{letrec } \overline{x_i : t_i = M_i} \text{ in } M$	recursive let binding
Atoms	$A ::= x$	term variable
t -types	$t ::= t_1 \rightarrow t_2$	function type
	Int	primitive type (integers)

cutable language, thus defining the meaning of a source program (this may be seen as an obviously-correct but vacuous usage analysis); and

4. an *operational semantics* (Section 2.4) for the executable language, with a notion of *correct* usage information, and by which we formalise our concept of usage.

These can be seen as four of the six parts of a usage analysis we defined in Section 1.5.1 (cf. Section 3.1). The supporting proofs are given in Section 2.5, and related work in Section 2.6.

The executable language, unlike the source language, contains *update flags*, which give information on the expected usage behaviour of the program. The operational semantics depends on the soundness of these flags for correct behaviour; if they are unsound then execution will get stuck. Thus the operational semantics encodes formally our intuitive concept of usage.

A usage analysis is simply a translation of the source language into the executable language; a sound usage analysis may infer any values for the update flags as long as they preserve the operational meaning of the program (even though in general the exact execution sequence will differ according to the flags). The aim of a usage analysis is to improve upon the trivial translation, choosing update flags that are more accurate but never unsound.

In our presentation of the executable language LX and its operational semantics – specifically in our use of shallow evaluation contexts, our various terminal sets of configurations, and our use of separate update flags independent of type annotations (against [TWM95a]) – we are greatly indebted to the presentation of Gustavsson [Gus98, §2]. We were inspired by Moran and Sands [MS99] to use Sestoft’s abstract machine style [Ses97] for our operational semantics, and we use some of their proof techniques.

2.2 The source language

Typed intermediate languages such as Core (Section 1.2.5) are generally extensions of the typed lambda calculus, usually containing (*inter alia*) type polymorphism and algebraic data types. At first, however, we treat a language *without* these features. We do this to avoid unnecessary clutter whilst we develop our theory; we will see in Chapter 5 that these features may be added in a largely orthogonal way. While the language described here is simplified, it still contains much of the technical complexity required to exercise usage type analysis.

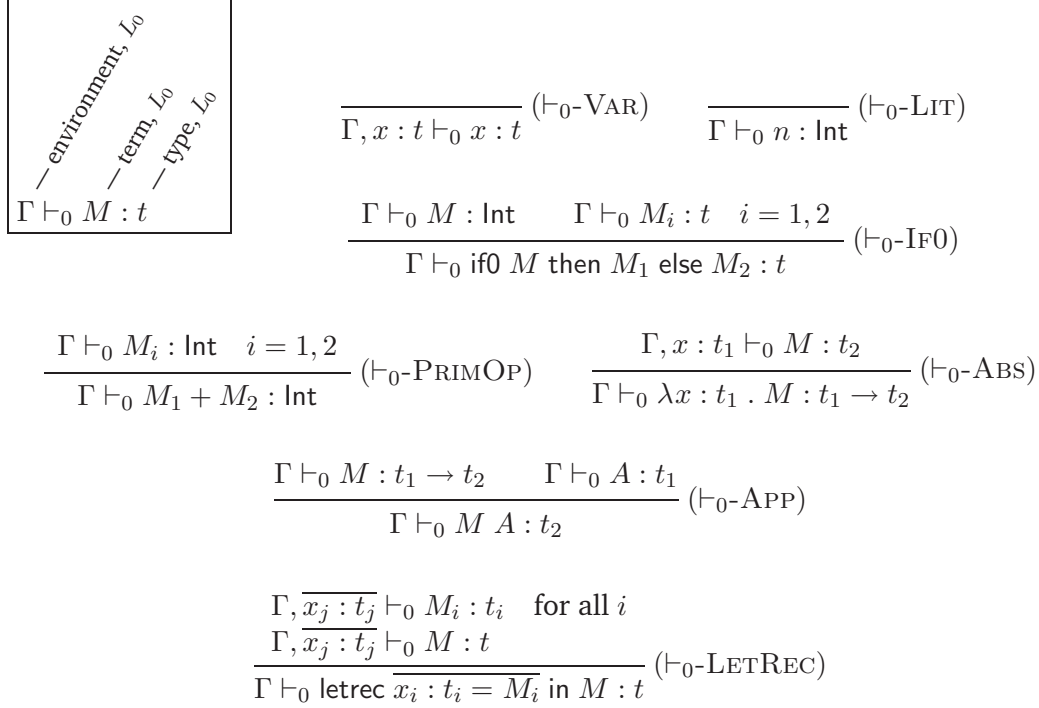
This language, L_0 , is presented in Figure 2.1. It is explicitly typed, with type annotations given to binding occurrences of variables in λ -abstractions and letrec bindings. The use of explicit types allows the type of any subterm to be determined locally, without performing type inference, and is a rather standard presentation for typed calculi. The user is not forced to provide all the type annotations herself; rather, they are inferred from context and optionally any annotations that are supplied by the user. Such inference is beyond the scope of this thesis. Throughout the thesis we omit type annotations from examples where they are uninteresting or clear from context.

L_0 terms are restricted syntactically to *A-normal form* [FSDF93]: function arguments are atoms (*i.e.*, variables) rather than general terms. This restriction forces all heap allocation and lazy evaluation to be represented explicitly by letrec bindings, and has become rather standard [Lau93, AF94, TMC⁺96, PJS98a, GS00b]. Most of the analyses in this thesis could be extended rather easily to unnormalised terms (we do this in the implementation, Section 6.4.4), but doing so here would unnecessarily complicate the presentation.

L_0 variables, literal constants, term abstractions, and applications are all standard. *Primops* are the primitive operations of the machine (arithmetic and logic, comparison, foreign functions, and so on). The arguments of a primop are of some primitive type, and are evaluated to values before applying the operation. Primops are always saturated (*i.e.*, given all their arguments). For simplicity, we consider only a single primop, addition; throughout the thesis the rules for other primops would differ only trivially from those for $+$, and could easily be added. Since we omit algebraic data types, we cannot include case in the simplified language; to introduce the potential for branching execution we include *if0*, which tests its (integer) first argument against zero, yielding its second argument if they are equal and its third if not. Recursion is introduced by letrec in the usual way: variables \overline{x}_i have the values of \overline{M}_i and are in scope over both the body M and all the right-hand sides \overline{M}_i . Note that we write \overline{M}_i to denote the vector M_1, M_2, \dots, M_n . There is an index of all notation in Appendix A. L_0 does not have a non-recursive let, because by appropriate α -conversion letrec can always be used instead.

Types are straightforward: functions $t_1 \rightarrow t_2$ from domain t_1 to codomain t_2 , and the primitive type *Int* (again, further primitive types could be added trivially and are omitted purely for simplicity of presentation).

As usual, terms are identified up to α -conversion; thus $\lambda x : \text{Int} . x$ and $\lambda y : \text{Int} . y$ are the same term. We follow the Barendregt Variable Convention [Bar81, §2.1.13] [Pit01], that all bound variables of a term are assumed to be distinct from each other

Figure 2.2 Well-typing rules for the source language L_0 .

and from the free variables of the term, and we implicitly perform α -conversion wherever necessary to preserve this property.

2.2.1 Source typing

The well-typing rules for L_0 are given in Figure 2.2, and are completely standard. This figure defines the relation $\Gamma \vdash_0 M : t$, which can be read as stating that “In type environment Γ , the L_0 term M can be given type t .” For example, rule $(\vdash_0\text{-IF0})$ states that in environment Γ , if M can be given type Int and M_1 and M_2 can both be given the same type t , then in the same environment the expression $\text{if0 } M \text{ then } M_1 \text{ else } M_2$ can be given the type t .

The type environment Γ is a finite set of pairs $x : t$ associating a variable x with a type t , with at most one pair $x : t$ for each variable x . Γ may be viewed as a partial function from variables x to types t , as in $\Gamma(x) = t$. The comma operator Γ, Γ' denotes union of two environments Γ and Γ' , subject to the above restriction. The set of variables bound by Γ is denoted $\text{dom}(\Gamma)$.

Since L_0 is explicitly typed, these rules are unambiguous and can be interpreted as an algorithm for computing the function $(\Gamma, M) \mapsto t$ in the usual way [LY98, Mil78, DM82].

A fully formal account would include judgements for well-formed types and environments, in a style similar to that of Cardelli [Car96]; we do not do this here.

2.3 The executable language

We will give meaning to the source language by providing it with an *operational semantics*. Specifically, we present a high-level *abstract machine* on which programs may be executed. In order to capture our intuitive understanding, we give to the abstract machine a notion of *usage*. That is, programs to be executed on it are annotated with (optional) usage information, and the machine makes use of this in its execution. Crucially, if the information provided to the abstract machine is incorrect, the execution fails. This gives us a way of verifying the correctness of usage information provided by an analysis: the information is correct if and only if the execution of the annotated program does not fail in this way.

In this section we define the annotated language as a separate language, the *executable language* LIX_0 , and give a *trivial translation* \mathcal{IT}_0 from L_0 into it, which simply embeds programs without adding any usage information.

2.3.1 The language

The terms and types of the executable language LIX_0 are defined in Figure 2.3 (the remaining syntactic categories are for the benefit of the operational semantics, below in Section 2.4). Evaluation contexts indicate where evaluation occurs in a term: in the function part of an application, in the first argument of a primop and then the second, and in the condition of a conditional.

LIX_0 contains usage information in the form of *update flags* on bindings and abstractions. Update flags χ are taken from the set $\{\bullet, !\}$, where \bullet denotes “not updatable/copyable” and $!$ denotes “updatable/copyable” or “not known” (see Section 1.3.3):

- for a *binding*, $\chi = !$ means the binding will be *updated* and $\chi = \bullet$ means it will not (*i.e.*, it is a single-entry thunk); and
- for an *abstraction*, $\chi = !$ means the abstraction may be *copied* (allowing multiple invocations) and $\chi = \bullet$ means it may not (*i.e.*, a one-shot lambda).

Literals may be freely copied, and so we do not give them an update flag.

LIX_0 also has an additional term form $\text{add}_n M$, which is a syntactic device used to handle the evaluation of arguments to a primop (an “additional expression form” [WF94, §7]). Primops must have all their arguments evaluated to a value before the result can be computed, but the operational semantics evaluates them one at a time. Once the first argument is evaluated, we reduce to a *partially-saturated* primop and place the next argument in evaluation position, proceeding in this manner until all arguments are evaluated. To accommodate this, we extend the syntactic category M of terms with a new term $\text{add}_n M$, denoting $n + M$, the primop $+$ where the left argument has been evaluated but the right has not.

In all other respects, L_0 and LIX_0 are identical.

We define an operation *stripping*, denoted \mathfrak{h} , which takes an LIX_0 term to its corresponding L_0 term. Stripping omits update flags, and translates $\text{add}_n M$ to $n + M$. Stripping \mathfrak{h} is not to be confused with erasure \mathfrak{b} , defined below; stripping omits update flags, leaving types, whereas erasure omits types, leaving update flags.

Figure 2.3 The executable language LIX_0 and configurations $LIXC_0$.

Terms	$M ::= R[M]$ $ \text{letrec } x_i : t_i =^{\chi_i} M_i \text{ in } M$ $ A$ $ V$	filled evaluation context recursive let binding atom value
Shallow evaluation contexts	$R ::= [\cdot] A$ $ [\cdot] + M$ $ \text{add}_n [\cdot]$ $ \text{if0 } [\cdot] \text{ then } M_1 \text{ else } M_2$	term application primop (addition) partially-saturated primop zero-test conditional
Atoms	$A ::= x$	term variable
Values	$V ::= n$ $ \lambda^{\chi} x : t . M$	literal (integer) term abstraction
Types	$t ::= t_1 \rightarrow t_2$ $ \text{Int}$	function type primitive type (integer)
Update flags	$\chi ::= \bullet$ $!$	not updatable/copyable updatable/copyable
Configurations	$C ::= \langle H; M; S \rangle$	where $\text{dom}(H) \not\subseteq \text{dom}(S)$
Heaps	$H ::= \emptyset$ $ H, x : t =^{\chi} M$	where $x \notin \text{dom}(H)$
Stacks	$S ::= \varepsilon$ $ R, S$ $ \#x : t, S$	where $x \notin \text{dom}(S)$

2.3.2 Erasure

The executable language LIX_0 bears type annotations just like L_0 , but we will see shortly that the operational semantics ignores these completely. This fact is important later (see Section 5.1.4): types are used in typechecking, analysis, and optimisation, but *types are ignored at runtime*. Thus we may *erase* the type annotations from LIX_0 and its operational semantics without changing its behaviour. Erasure is justified by an analogue of the famous theorem of Milner, that “well-typed programs do not go wrong” [Mil78, §3.7]; this result is proven in Section 2.5. Since a well-typed program never has a type error, types need not be checked during execution, and thus need not even be present. Implementations of Haskell, ML, and other Hindley–Milner-based languages therefore erase all type information prior to execution.

This means that the *actual* executable language is a language without types, which we call LX . Rather than present yet another language, we simply indicate the portions of LIX_0 that are not in LX , the *instrumentation*,¹ by **lowlighted text** in Figure 2.3 and subsequently. The instrumentation is very useful in expressing the analyses, but does not form a part of the analyses’ ultimate output. The instrumentation in LIX_0 consists of L_0 types; subsequent analyses in the thesis will augment LX with instrumentation from other languages: L_1 , L_2 , and so on. However, in all cases the underlying operational semantics is exactly that of LX ; in other words, although the instrumentation and its manipulation by the semantics may differ, the projection into LX and its manipulation are identical.

The operation of removing the instrumentation from LIX_0 we call *erasure*, and denote \flat ; it generalises to heaps, stacks, and contexts as well as terms. It is defined by a simple induction on the structure of its argument.

Type erasure is standard for strongly-typed languages [WF94, §1], and was common already when Milner wrote his paper [Mil78, p. 349].

2.3.3 The trivial translation

The trivial translation \mathcal{IT}_0 takes L_0 programs into LIX_0 , where they may be executed. The translation is *trivial* because it infers no usage information. We write $\mathcal{IT}_0 \llbracket M \rrbracket$ for the trivial translation of L_0 term M into LIX_0 .

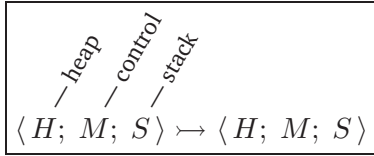
$\mathcal{IT}_0 \llbracket M \rrbracket$ embeds M into LIX_0 by marking all update flags as $!$, “not known” (i.e., “updatable/copyable”), and in all other respects leaving the term untouched. The definition is a simple induction on the structure of the source term and we omit the details. Note that (as with all usage analyses in this thesis) we have $(\flat \circ \mathcal{IT}_0) = id_{L_0}$, i.e., $(\mathcal{IT}_0 \llbracket M \rrbracket)^\flat = M$.

2.4 Operational semantics

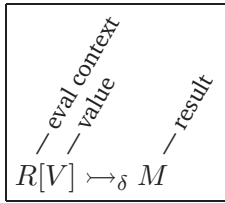
We define the operational semantics of LIX_0 by a high-level abstract machine, essentially the first lazy abstract machine of Sestoft [Ses97]. The semantics is given in

¹ LIX_0 stands for “eXecutable Language, Instrumented with types from L_0 ”.

Figure 2.4 The $LIXC_0$ operational semantics: \mapsto and \mapsto_δ .



$$\begin{array}{lll}
 \langle H; R[M]; S \rangle & \mapsto & \langle H; M; R, S \rangle & (\mapsto\text{-UNWIND}) \\
 \langle H; V; R, S \rangle & \mapsto & \langle H; M; S \rangle & (\mapsto\text{-REDUCE}) \\
 & & \text{if } R[V] \mapsto_\delta M \\
 \langle H; \text{letrec } \overline{x_i : t_i =^{x_i} M_i} \text{ in } M; S \rangle & & & (\mapsto\text{-LETREC}) \\
 & \mapsto & \langle H, y_i : \overline{t_i =^{x_i} M_i[\phi]}; M[\phi]; S \rangle & \\
 & & \text{where } \overline{y_i} \not\in \text{dom}(H) \cup \text{dom}(S) & \\
 & & \phi = [\overline{y_i}/\overline{x_i}] & \\
 \langle H, x : t =^\bullet M; x; S \rangle & \mapsto & \langle H; M; S \rangle & (\mapsto\text{-VAR-ONCE}) \\
 \langle H, x : t =^! M; x; S \rangle & \mapsto & \langle H; M; \#x : t, S \rangle & (\mapsto\text{-VAR-MANY}) \\
 \langle H; V; \#x : t, S \rangle & \mapsto & \langle H, x : t =^! V; V; S \rangle & (\mapsto\text{-UPDATE}) \\
 & & \text{where } |V| = \bullet \Rightarrow x \notin \text{fv}(H, V, S) &
 \end{array}$$



$$\begin{array}{ll}
 \text{where } |n| & = ! \\
 |\lambda^x x : t. M| & = \chi
 \end{array}$$

$$\begin{array}{lll}
 (\lambda^x x : t. M) A & \mapsto_\delta & M[A/x] & (\mapsto_\delta\text{-APP}) \\
 n + M & \mapsto_\delta & \text{add}_n M & (\mapsto_\delta\text{-PRIMOP-L}) \\
 \text{add}_{n_1} n_2 & \mapsto_\delta & n_3 & \text{if } n_3 = n_1 + n_2 & (\mapsto_\delta\text{-PRIMOP-R}) \\
 \text{if0 } n \text{ then } M_1 \text{ else } M_2 & \mapsto_\delta & M_i & \text{if } i = (n = 0 ? 1 : 2) & (\mapsto_\delta\text{-IF0})
 \end{array}$$

Figure 2.4.

2.4.1 Statics

The semantics is defined over abstract machine *configurations*, which are triples $\langle H; M; S \rangle$ consisting of a heap H , a term under evaluation M (traditionally called the *control* [Lan64]) and a stack S . The set of all well-formed LIX_0 configurations is denoted $LIXC_0$. A *heap* is an unordered set of quadruples $x : t =^x M$, at most one for any x , denoting a binding of variable x of type t to term M with update flag χ . The set of variables bound by H is denoted $\text{dom}(H)$. A *stack* is a list of shallow evaluation contexts R and update frames $\#x : t$, as explained below. Values bound by the heap must be disjoint from those bound by update frames on the stack; we write $\text{dom}(H) \not\downarrow \text{dom}(S)$ to abbreviate $\text{dom}(H) \cap \text{dom}(S) = \emptyset$.

During evaluation, the term is unwound into a list of nested *shallow evaluation contexts* according to the evaluation order, until a variable or value is reached. For example, in the term

$$M = (\text{if0 } 1 + 2 \text{ then } M_1 \text{ else } M_2) x$$

the function is evaluated before the argument, the condition is evaluated before the branches, and the left argument of the sum is evaluated before the right, and so M is unwound to

$$\langle \emptyset; 1; [\cdot] + 2, \text{if0 } [\cdot] \text{ then } M_1 \text{ else } M_2, [\cdot] x \rangle$$

i.e.,

$$[\text{if0 } [[1] + 2] \text{ then } M_1 \text{ else } M_2] x$$

and the value 1 is evaluated first. Each shallow evaluation context is a simple term with a hole in the evaluation position, such that the term in the hole must be evaluated before execution can proceed with the context.

Evaluation stops when a term is reduced to a *value* V , i.e., a literal or a function abstraction. Values cannot be reduced further.

To model laziness, when a heap binding is being evaluated an *update frame* recording the name of the variable whose value is being computed is pushed onto the stack. When the computation is complete, the update frame is popped off and the binding updated with the new value. At most one update frame may appear for any variable x , and the set of variables for which update frames appear in stack S is denoted $\text{dom}(S)$.

2.4.2 Dynamics

The semantics of Figure 2.4 defines a transition relation $C \mapsto C'$ from configurations to configurations. This transition relation unwinds terms and manipulates the heap, and uses a primitive transition relation $R[V] \mapsto_\delta M$ to perform the basic computations (function application, primops, and conditional).

The rule (\mapsto -UNWIND) unwinds a shallow evaluation context: if the control is a filled evaluation context $R[M]$, the context R is placed on the stack and evaluation proceeds with M in the control. The dual rule (\mapsto -REDUCE) applies when the term

is finally reduced to a value and the context is on the top of the stack; the value is placed back into the context and a primitive reduction $R[V] \mapsto_{\delta} M$ is performed.

The rule (\mapsto -LETREC) places new bindings into the heap, renaming variables to fresh names to avoid conflicts, and evaluation proceeds with the body of the letrec.

When a variable appears as the control, it is looked up in the heap by either (\mapsto -VAR-ONCE) or (\mapsto -VAR-MANY), and evaluation continues with the term to which it is bound. The binding is removed from H ; this is called *blackholing* [Jon92, Lau93] and enables the detection at runtime of certain programs that would otherwise not terminate, such as `letrec $x : \text{Int} =^! x$ in x` . If the binding is marked \bullet (not updatable), the binding is removed entirely since it should not be needed again; if instead the binding is marked $!$ (updatable), an update frame is pushed onto the stack, indicating that the binding should be restored when evaluation of the right-hand side is complete (using the newly-computed value as the right-hand side).

This update is performed by (\mapsto -UPDATE): when the control has been reduced to a value and an update frame is on the top of the stack, a binding is added to the heap, binding the named variable to the newly-computed value; the update flag is necessarily $!$ since if it were \bullet the update frame would not have been created. The side condition on (\mapsto -UPDATE) prevents copying of a value marked \bullet (i.e., not copyable); it makes use of the auxiliary function $|\cdot|$, which returns the update flag of a value if present and $!$ (copyable/not known) otherwise. Note that a binding to a dead variable (i.e., one which can never be looked up) is not considered to be a copy in this sense.²

The primitive reductions $R[V] \mapsto_{\delta} M$ are straightforward. Rule (\mapsto_{δ} -APP) is the familiar β rule (the argument is always an atom because of the restriction to A-normal form, Section 2.2). Rules (\mapsto_{δ} -PRIMOP-L) and (\mapsto_{δ} -PRIMOP-R) perform the addition primop; recall we force the evaluation of the left argument first, and then use a placeholder to force evaluation of the right argument before performing the addition itself. Finally, rule (\mapsto_{δ} -IF0) selects one of the two branches, based on whether its argument is zero or non-zero (the side condition is expressed using the conditional expression $(P ? x : y)$, meaning “if P then x else y ”).

2.4.3 Evaluation and termination

We now define initial and terminal configurations, dividing the latter into value and stuck configurations.

A program is run by placing it in the control of a machine with an empty heap and stack: the *initial configuration* of a program is given by $\langle ; ; \rangle$, where

$$(M)^{\langle ; ; \rangle} \triangleq \langle \emptyset; M; \varepsilon \rangle$$

Evaluation proceeds via the reduction relation \mapsto until a configuration is reached where no rule applies. The configuration C *terminates with configuration* C' , denoted

²The creation of such a binding is impossible in the monomorphic type system of Chapter 3, but it is possible in the polymorphic type system of Chapter 4. See the proofs of Lemmas D.7 and D.10 in the appendix.

$C \Downarrow C'$, when it can evaluate no further. That is,

$$C \Downarrow C' \triangleq C \mapsto^* C' \wedge \neg \exists C'' . C' \mapsto C''$$

Terminal configurations C' may be of several forms.

- $C' \in \text{Value}$ if C' is of the form $\langle H; V; \varepsilon \rangle$. This is normal termination: C' is a *value configuration*, and the result of the program is V .

All other terminal configurations are called *stuck configurations*, which can be partitioned into four sets:

- $C' \in \text{BlackHole}$ if C' is of the form $\langle H; x; S \rangle$ where $x \notin \text{dom}(H)$ but $x \in \text{dom}(S)$. This indicates a black hole (Section 2.4.2) has been detected.
- $C' \in \text{Wrong}$ if C' is of the form $\langle H; V; R, S \rangle$ where $R[V] \not\vdash_\delta$. This is a runtime type error.
- $C' \in \text{BadBinding}$ if C' is of the form $\langle H; x; S \rangle$ where $x \notin \text{dom}(H) \cup \text{dom}(S)$. This indicates an incorrect binding update flag.
- $C' \in \text{BadValue}$ if C' is of the form $\langle H; V; \#x : t, S \rangle$ where $|V| \neq !$. This indicates an incorrect value update flag.

These sets are exhaustive and disjoint:

Theorem 2.1 (Configuration sets)

All $LIXC_0$ (or LXC) configurations C' are either reducible or in exactly one of the sets $\{\text{Value}, \text{BlackHole}, \text{Wrong}, \text{BadBinding}, \text{BadValue}\}$.

Proof By inspection. □

Our goal is to ensure that a program never reaches a configuration C' in *Wrong* or in $\text{Bad} = \text{BadBinding} \cup \text{BadValue}$.³ Configurations in *Wrong* arise from runtime type errors, and as we show in Section 2.5, translations of well-typed source programs do not go wrong. Configurations in *Bad* arise from incorrect usage information, and a usage analysis must take care to avoid these. An LIX_0 (or LX) program is *well-annotated* if it never reaches a bad configuration, and a usage analysis is *sound* if it yields only well-annotated programs. All the usage analyses in this thesis are proven sound.

2.4.4 The trivial translation

It should now be clear that the trivial translation of Section 2.3.3 is a sound usage analysis. By not inferring any usage information at all, and instead always using the “not known” update flag $!$, it ensures that no program it translates can ever reach a configuration in *BadBinding* or *BadValue*.

³ $C' \in \text{BlackHole}$ means merely that the program is nonterminating; this is not an error and we do not attempt to avoid it. This may be clarified by imagining an additional rule $C' \in \text{BlackHole} \Rightarrow C' \mapsto C'$.

The essential observation is that, for well-scoped programs, we can only reach *BadBinding* or *BadValue* if an update flag was \bullet : *BadBinding* is caused by the rule (\rightarrow -VAR-ONCE) removing a \bullet -annotated binding, and *BadValue* is caused by an attempt to copy a \bullet -annotated lambda. Therefore, if we omit all \bullet -flags and provide only $!$ -flags, we avoid *Bad* and the analysis is sound. This justifies our interpretation of $!$ as “not known”.

Because these *Bad* configurations are never reached, one may discount the rule (\rightarrow -VAR-ONCE) and the side condition to (\rightarrow -UPDATE) and recover the standard semantics for L_0 as given in [Ses97, Lau93] *inter alia*.

2.4.5 Copying and using abstractions

Our intuitive notion of the usage of a function (Section 1.3.3) refers to the number of times it is *used*, but the side condition on (\rightarrow -UPDATE) restricts the number of times it is *copied* instead. The distinction would become important in the presence of a *seq* operator, which evaluates its first argument but does not use its value, returning the value of its second argument instead. (A strict *let* operator would similarly introduce zero usages into the system). Consider for example the program

$$\text{letrec } f =^! \lambda^{\bullet} x . x + 1; z =^{\bullet} 41 \text{ in } \text{seq } f (f z)$$

Execution of this would demand the value of f twice, but apply the lambda only once, consistent with the flags. Under the present operational semantics, however, execution would become stuck after the evaluation of f , with $\lambda^{\bullet} x . x + 1$ in the control and $\#f$ on the top of the stack.

In order to allow this, we would have to track usages apart from copies, either with pointers or with cleverer update frames. We introduce the latter in the proposed semantics of Appendix C. But in the absence of operators such as *seq*, the present semantics is sufficient for our purposes: one can see by inspection that a function cannot be used more than once without being copied first, and so interpreting \bullet as “not copyable” is a stronger restriction than interpreting it as “not used more than once”. Thus analyses sound for this interpretation are sound for the “not used more than once” interpretation.

2.5 Proof of soundness

In order for our trivial translation to make sense, we must prove that well-typed programs do not go wrong (following Milner [Mil78, §3.7]). In the present context, this means that execution of a well-typed L_0 term, translated into LIX_0 by \mathcal{IT}_0 , will never terminate in a configuration in *Wrong*, *BadBinding*, or *BadValue*. This result demonstrates that our type system and translation are sound with respect to the operational semantics.

In the present section we outline the proof of this result; the full details are given in Appendix D.

2.5.1 Definitions

We first define typing for LIX_0 . The idea is that an LIX_0 term is an L_0 term with update flags and partially saturated primops, and so typing an LIX_0 term is the same as typing its stripped version in L_0 . That is, typing for LIX_0 terms uses the rules of Figure 2.2 by ignoring all update flags, and adds a rule for $\text{add}_n M$, derived from $(\vdash_0\text{-PRIMOP})$ and $(\vdash_0\text{-LIT})$:

$$\frac{\Gamma \vdash_0 M : \text{Int}}{\Gamma \vdash_0 \text{add}_n M : \text{Int}} (\vdash_0\text{-PRIMOP-R})$$

A configuration C may be translated into the corresponding term $\text{trans}(C)$, such that when $\text{trans}(C)$ is evaluated it will unwind (by $(\rightarrow\text{-UNWIND})$ and $(\rightarrow\text{-LETREC})$) into the original configuration C . This allows us to treat $LIXC_0$ configurations simply as the equivalent terms in LIX_0 . If C is an $LIXC_0$ configuration, then $\text{trans}(C) \in LIX_0$ is defined by:

$$\begin{aligned} \text{trans}\langle H; M; R, S \rangle &\triangleq \text{trans}\langle H; R[M]; S \rangle \\ \text{trans}\langle H; M; \#x : t, S \rangle &\triangleq \text{trans}\langle H, x : t =^! M; x; S \rangle \\ \text{trans}\langle H; M; \varepsilon \rangle &\triangleq \text{letrec } H \text{ in } M \end{aligned}$$

(Notice that we allow an empty set of bindings in `letrec` for simplicity.)

Given this equivalence, we may define well-typing for configurations in the obvious way. We write $\vdash_0 C : t$, denoting that the $LIXC_0$ configuration C has type t , iff its translation $\text{trans}(C)$ has type t in the empty environment. That is,

$$\frac{\emptyset \vdash_0 \text{trans}\langle H; M; S \rangle : t}{\vdash_0 \langle H; M; S \rangle : t} (\vdash_0\text{-CONFIG})$$

2.5.2 Proof outline

We now move on to the proof itself. The crucial lemma is the progress lemma; it says that any well-typed configuration is either terminal in *Value* or *BlackHole*, or can progress further.

Lemma 2.2 (Progress)

For all $LIXC_0$ configurations C , if $\vdash_0 C : t$ then either (i) $C \in \text{Value} \cup \text{BlackHole}$ or (ii) $\exists C' . C \rightarrow C'$ and $C \rightarrow C' \Rightarrow \vdash_0 C' : t$.

Proof By cases on C , using various lemmas. The full proof is given in Appendix D, Lemma D.10. \square

We now need merely establish the initial state and apply the progress lemma repeatedly until we are done.

Lemma 2.3 (Instrumented soundness for $L_0, \vdash_0, \mathcal{IT}_0$)

For all L_0 terms M , if $\emptyset \vdash_0 M : t$ then

- (i) $(\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle}$ is well-defined.
- (ii) $\vdash_0 (\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} : t$.
- (iii) If $(\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C'$ then $C' \in \text{Value} \cup \text{BlackHole}$.

Proof

1. By inspection of the definitions of $\langle ; ; \rangle$ and \mathcal{IT}_0 .
2. By induction on the structure of M .
3. By Lemma 2.2, by induction on the length of derivation of $(\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C'$. \square

This shows that execution of a well-typed L_0 program does not go wrong, if the L_0 program is translated into LIX_0 by the trivial translation and executed using the operational semantics defined above. Thus our type system \vdash_0 is sound.

However, recall from Section 2.3.2 that in practice the program is executed with all types erased. To demonstrate that this is possible, *i.e.*, that the types are purely instrumentation and the execution behaviour does not depend on them, we prove the final result, Theorem 2.5. This involves the translation \mathcal{T}_0 used in practice, defined by $\mathcal{T}_0 \triangleq \flat \circ \mathcal{IT}_0$, *i.e.*, the instrumented translation \mathcal{IT}_0 followed by erasure \flat . In order to prove this, we require one more lemma.

The Correspondence Lemma 2.4 states that the $LIXC_0$ reduction sequence and the LXC reduction sequence proceed in lock-step when started respectively on a well-typed $LIXC_0$ configuration C and on its erasure $(C)^\flat$.

Lemma 2.4 (Correspondence)

The functions $\langle ; ; \rangle$ and \mapsto are well-defined on LX and LXC (as well as LIX_0 and $LIXC_0$), and make the following two diagrams commute:

$$\begin{array}{ccc}
 LIX_0 & \xrightarrow{\langle ; ; \rangle} & LIXC_0 \\
 \downarrow \flat & & \downarrow \flat \\
 LX & \xrightarrow{\langle ; ; \rangle} & LXC
 \end{array}
 \qquad
 \begin{array}{ccc}
 LIXC_0 & \xrightarrow{\mapsto} & LIXC_0 \\
 \downarrow \flat & & \downarrow \flat \\
 LXC & \xrightarrow{\mapsto} & LXC
 \end{array}$$

Proof By inspection of the definitions. \square

Our final result then is:

Theorem 2.5 (Soundness for L_0 , \vdash_0 , \mathcal{T}_0)

For all L_0 terms M , if $\emptyset \vdash_0 M : t$ then

- (i) $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle}$ is well-defined.
- (ii) If $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C'$ then $C' \in \text{Value} \cup \text{BlackHole}$.

Proof Follows from Lemma 2.3 by the Correspondence Lemma 2.4. \square

This states that a well-typed L_0 program may be executed by performing the trivial translation and then stripping all instrumentation from the resulting program. The usage analyses in the remainder of the thesis are all more complicated than this, but all of them satisfy this property. Each takes a well-typed L_0 program, translates it into an instrumented executable language, and then strips the instrumentation. The resulting LX program is well-defined, and if it terminates, terminates in *Value* or *BlackHole*.

2.6 Related work

The ultimate origin of the operational semantics for lazy evaluation described in this chapter is the natural semantics of Launchbury [Lau93]. This is an untyped, big-step semantics defined on (heap, expression) pairs, for a lambda calculus with letrec, case, and constructors. It features blackholing for variables, as in our Section 2.4.2. This semantics was modified slightly for [TWM95a]; independently it was translated to an abstract machine by Sestoft [Ses97] and used by, *inter alia*, Moran and Sands [MS99] and Gustavsson [Gus98], where we discovered it. Update flags and update frames first appeared in the Three Instruction Machine (TIM) and are due to Fairbairn and Wray [FW87]. The first abstract machine for a functional language was the SECD machine (Stack-Environment-Control-Dump) of Landin [Lan64].

We now consider the work of Gustavsson *et al.* in some detail, in consequence of its close relationship to the present work.

Gustavsson and Sands in [GS01a] define the *use-once-don't-drag* criterion to mean

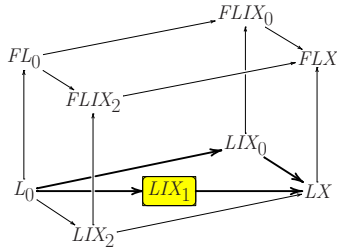
... that an argument must be used at most once and when it is used there may be no other references to the closure holding the argument.

They go on to define this formally by an operational abstract machine semantics exactly of the form given in this chapter, with the addition of a don't-drag side condition on (\rightarrow -VAR-ONCE) requiring that $x \notin fv(H, M, S)$, and the omission of the side condition on (\rightarrow -UPDATE) and update flags on abstractions; they also explain that in the presence of the don't-drag side condition a *garbage collection* rule for unreferenced heap bindings and update flags is required. They prove that the use-once-don't-drag property is sufficient to show the space- and work-safety of the inlining transformation described in Section 1.3.4. They implicitly define a usage analysis as an annotation of a source program with update flags (“use-once-don't-drag bindings”) such that evaluation never becomes stuck due to incorrect flags, and they give a result (Proposition 2.3) roughly equivalent to our Correctness Theorem 3.2 showing that the update flags (if correct) do not affect the semantics of the program.

The semantics of [Gus98] is similar, but in addition to updates it also addresses *update marker checks*, annotating values and (value-generating) primops with intervals $\iota = [\nu, \xi]$ (where $\nu, \xi \in \mathcal{P}(\mathbb{N}) \cup \{\omega\}$) denoting the number of update markers expected on the stack after the value is obtained. This enables another optimisation, *update marker check avoidance*, since the presence or absence of an update marker during evaluation may now be determined statically.

Gustavsson’s polymorphic analysis [GS00b] uses essentially the same operational semantics, except that interval annotations ι are replaced by κ annotations taken from the same annotation set as the binding annotations. These are examined by two rules corresponding to our single rule (\rightarrow -UPDATE). The semantics of these value annotations differs from ours in that they appear on primops as well as values, and whereas our semantics becomes stuck if an attempt is made to copy a \bullet -annotated value, Gustavsson and Svenningsson’s semantics simply drops the binding which is being updated, thus becoming stuck only if that binding was live at the time of update. This is related to the notion of “use-once-don’t-drag” discussed above, and permits decoupling of the usage of values and expressions without the overhead of introducing update marker check intervals.

Chapter 3.



Monomorphic Usage Types

Our first usage analysis is based on type inference for a relatively simple monomorphic usage type system. As well as being interesting in its own right, this system introduces the concepts underlying the more complex usage type systems of later chapters.

3.1 Introduction: a usage analysis

A *usage analysis* is an algorithm that takes a source term in L_0 and returns an executable term in LX (or an instrumented variant), annotating the term with usage information. The decision to use a *type-based analysis* (see Section 1.5) provides us with a standard framework within which to design our analysis and prove it correct. A type-based analysis consists of six parts, around which we structure this chapter (indeed, Chapters 4 and 5 follow the same plan):

1. a *source language* of programs to analyse (L_0 , already presented in Section 2.2);
2. an *operational semantics* defining the desired property (\mapsto , already presented in Section 2.4);
3. a *typed target language*, an instrumented executable language consisting of the source language extended with types intended to carry information about the desired property (LIX_1 , Section 3.2);
4. a set of *well-typing rules* which define the typings that carry valid information (\vdash_1 , Section 3.3);

5. a *goodness ordering* on typings, defining the *best* valid typing should there be more than one (\sqsubseteq , Section 3.4); and
6. an *inference algorithm* which computes the best valid typing for each source program (\mathcal{IT}_1 , Section 3.5).

The use of a goodness ordering is non-standard, and arises because the type systems in this thesis lack the usual principal type property [Jim96, Hin69] (see Section 3.7.2).

The analysis must also be supported by certain proofs, all of which appear in Section 3.6:

7. a *type soundness proof* demonstrating that the well-typing rules capture the desired property (Theorem 3.1);
8. a *nonrestrictivity proof* demonstrating that all source terms have a valid target typing (Theorem 3.3);
9. an *inference soundness proof* demonstrating that the inference algorithm computes a valid typing for all source terms that have one (Theorem 3.7); and
10. an *inference pseudo-completeness proof* demonstrating that the inference algorithm computes the *best* valid typing if there is more than one (Theorem 3.8).

In Section 3.6 we also prove a *complexity bound* (Theorem 3.9).

The behaviour and expressiveness of the analysis is discussed in Section 3.7. Finally, we consider the technical issues of dealing with separate compilation (Section 3.8), and summarise the related work (Section 3.9).

The analysis presented in this chapter was first published by the author (with Simon Peyton Jones) in [WPJ99]. That paper also included a discussion of algebraic data types and type polymorphism, which we here defer to Chapter 5. The analysis is not especially novel, being only a minor extension of that of Turner, Wadler, and Mossin [TWM95a]; the crucial difference is the addition of subsumption (Section 3.3.5). In addition, we believe the presentation is cleaner and clearer than that of either paper.¹

3.2 A language with usage types

Our first task is to design a *typed target language* which we shall call LIX_1 , an instrumented executable language consisting of LIX_0 extended with types carrying usage information. This language is presented in Figure 3.1. It differs from the earlier language (Figure 2.3) in only two ways: lambda terms have an additional annotation κ , and type annotations are of sort σ rather than t . The operational semantics is identical for each, *mutatis mutandis* (the κ annotations on lambdas are ignored). We now discuss the extended type and term languages in detail.

¹When reading these papers, it is important to note that the primitive ordering on usage annotations is the opposite of that in this thesis: in [WPJ99], [TWM95a], and [Gus98] the ordering is $1 \leq \omega$, whereas in the present thesis it is $\omega \leq 1$.

Figure 3.1 The usage-typed language LIX_1 (cf. Figure 2.3).

Terms	$e ::= a$ $ n$ $ \lambda^{\kappa, \chi} x : \sigma . e$ $ e a$ $ e_1 + e_2$ $ \text{add}_n e$ $ \text{if0 } e \text{ then } e_1 \text{ else } e_2$ $ \text{letrec } \overline{x_i : \sigma_i =^{\chi_i} e_i} \text{ in } e$	atom literal (integer) term abstraction term application primop (addition) partially-saturated primop zero-test conditional recursive let binding
Atoms	$a ::= x$	term variable
τ -types	$\tau ::= \sigma_1 \rightarrow \sigma_2$ $ \text{Int}$	function type primitive type (integer)
σ -types	$\sigma ::= \tau^\kappa$	usage-annotated type
Usage annotations	$\kappa ::= 1$ $ \omega$	used at most once possibly used many times
Update flags	$\chi ::= \bullet$ $!$	not updatable/copyable updatable/copyable

Shallow evaluation contexts R , values v , configurations C , heaps H , and stacks S are defined in the same manner as for LIX_0 .

3.2.1 The type language

The types of LIX_1 are the types of the source language L_0 , annotated with usage annotations κ . Types are defined by mutual recursion between two sorts, σ of types usage-annotated on top, and τ of types not usage-annotated on top. We occasionally use the variable ψ to range over both τ - and σ -types.

Usage annotations are denoted κ , and may be either 1 or ω , with the intuitive meanings “used at most once” and “possibly used many times” respectively (Section 1.3.3). Annotations are ordered by \leq , with $\omega \leq 1$. The unusual orientation of this ordering arises from the direction of the subtyping relation, as explained in Section 3.3.5. These usage annotations are intended to correspond to the update flags of the executable language: 1 corresponds to \bullet (“not updatable/copyable”), and ω to $!$ (“updatable/copyable” or “not known”).

Every expression is given a σ -type, and the topmost annotation $|\sigma|$ of that type is called the *usage* of the expression. The primitive-typed expression 42, for example, may have the type Int^ω , meaning that it is an integer which may be used many times; $|\text{Int}^\omega| = \omega$. For t an L_0 type, we write $\lceil t \rceil_\sigma^\omega$ for the function that returns a σ -

type consisting of t with ω in every usage annotation position, and similarly for $[t]_\tau^\omega$ which returns a τ -type.

While primitive types have no further structure, function types have an argument and a result type, and in designing the type system it was necessary to decide whether these each required separate annotations:

- The *usage of a function* may differ from the *usage of its result*, since the function may be applied several times and its result each time may be shared amongst any number of consumers. Hence the result type must be annotated distinctly from the function itself.
- The *usage of the argument* should be recorded, since this is a property of the function. Thus the argument type should be annotated with this usage, which is clearly distinct from the topmost annotation and that of the result.

This means that both the argument and result types of a function should be annotated. The function $\lambda x . x + 1$, then, may be given type $(\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega$, denoting a function from Int to Int , itself usable many times, each time using its argument once and returning a result that may be used many times. Compare $\lambda x . x + x$, which uses its argument *twice*: the above type becomes $(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega$, indicating that it uses the argument many times. A used-once thunk may be passed to the first function, but not to the second.

We will occasionally use the notion of the *polarity* (or *variance*) of an annotation position. As usual [TS96, def. 47] this is obtained by counting the number of arrows one must pass to the left of to reach the annotation position: if this is even the position has *positive* polarity (+) and is *covariant*; if it is odd the position has *negative* polarity (−) and is *contravariant*. For example, here is a usage type decorated with the polarity of its annotation positions:

$$\left((\text{Int}^{\omega^+} \rightarrow \text{Int}^{\omega^-})^{\omega^-} \rightarrow \text{Int}^{\omega^-} \rightarrow (\text{Int}^{\omega^-} \rightarrow \text{Int}^{\omega^+})^{\omega^+} \right)^{\omega^+}$$

A usage variable is said to *occur positively (negatively)* if it appears in a positive (negative) position.² We use the variable ε to range over polarities $\{+, -\}$.

3.2.2 The term language

The terms of LIX_1 are the terms of the instrumented executable language LIX_0 (Section 2.3.1) with types changed from t to σ and usage annotations κ added to lambda abstractions. The stripping (to L_0) and erasure (to LX) functions are extended to strip usage annotations thus:

$$(\lambda^{\omega, !} x : \text{Int}^1 . x + 1)^\natural = \lambda x : \text{Int} . x + 1 \quad \text{and} \quad (\lambda^{\omega, !} x : \text{Int}^1 . x + 1)^\flat = \lambda^! x . x + 1$$

LIX_1 is *explicitly-typed*. In such calculi, each term or subterm records enough information to reconstruct unambiguously the derivation tree used to type it (at

²Note that Gustavsson [Gus99, §5.5.2] defines covariance and contravariance oppositely, because of the opposite direction of his primitive subtype ordering (see Section 3.9).

least up to subsumption). Consequently, the type of any term or subterm may be determined purely by *local* inspection, given only the typing environment and no other context. This explicit typing property is very useful in a type-driven compiler.

The term abstraction $\lambda^{\kappa, \chi} x : \sigma . e$ of LIX_1 bears two annotations: a usage annotation and an update flag. We briefly explain why these are necessary.

Usage annotations κ . Consider the term $\lambda x : \text{Int}^1 . x + 1$. This has a type of the form $(\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega$, but there is not enough information to determine the correct topmost annotation without examining the context in which it is used. To preserve explicit typing, we record this topmost annotation in the usage annotation of the abstraction thus: $\lambda^\omega x : \text{Int}^1 . x + 1$. The type of this expression is now unambiguously $(\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega$. This is important because in LIX_1 , the type of a lambda abstraction, and the constraints the derivation places on the types of free variables, differ depending on the usage of the abstraction itself (see Section 3.3.4).

Update flags χ . We saw in Sections 2.3.1 and 2.4.2 that update flags χ are used to control the abstract machine's execution of an LX program. Since the same abstract machine is used to execute LIX_1 terms, the update flags must still be present in order to control it.

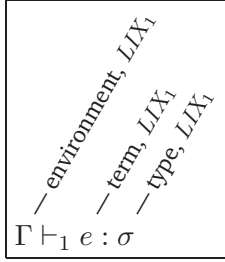
Thus lambda abstractions bear *two* annotations, a usage annotation to preserve explicit typing, and an update flag to control execution. Distinguishing them allows us to discuss the end results of the analysis (update flags, which survive type erasure, enabling safe execution) independently of the means by which they are obtained (usage annotations and a usage type system).³

Despite this separation of usage annotations and update flags, the two are very closely related. In fact, the update flag is an abstraction of the topmost usage annotation of the corresponding type: 1 corresponds to \bullet , and ω corresponds to $!$. For this reason, well-typed LIX_1 terms may contain $\lambda^{1, \bullet} x . e$ or $\lambda^{\omega, !} x . e$, but never the cross-cases.⁴ (This one-to-one relationship between update flags and usage annotations does not hold in the type system of Chapter 4.)

³In this respect our presentation is cleaner than that of [TWM95a], whose operational semantics is not strictly type-free. This property is required if we are to use a type-erased executable language in practice (Section 2.3.2).

⁴Even though $\lambda^{1, !} x . e$ would be sound, no reasonable analysis would elect to miss such an opportunity, and $\lambda^{\omega, \bullet} x . e$ is potentially unsound.

Figure 3.2 Well-typing rules for LIX_1 (cf. Figure 2.2).



$$\frac{}{\Gamma, x : \sigma \vdash_1 x : \sigma} (\vdash_1\text{-VAR}) \quad \frac{}{\Gamma \vdash_1 n : \text{Int}^\omega} (\vdash_1\text{-LIT})$$

$$\frac{\Gamma \vdash_1 e : \text{Int}^1 \quad \Gamma \vdash_1 e_i : \sigma \quad i = 1, 2}{\Gamma \vdash_1 \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma} (\vdash_1\text{-IF0})$$

$$\frac{\Gamma \vdash_1 e_i : \text{Int}^1 \quad i = 1, 2}{\Gamma \vdash_1 e_1 + e_2 : \text{Int}^\omega} (\vdash_1\text{-PRIMOP}) \quad \frac{\Gamma \vdash_1 e : \text{Int}^1}{\Gamma \vdash_1 \text{add}_n e : \text{Int}^\omega} (\vdash_1\text{-PRIMOP-R})$$

$$\frac{\begin{array}{l} \Gamma, x : \sigma_1 \vdash_1 e : \sigma_2 \\ \text{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \text{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \quad \text{for all } y \in \Gamma \end{array}}{\Gamma \vdash_1 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_1\text{-ABS})$$

$$\frac{\Gamma \vdash_1 e : (\sigma_1 \rightarrow \sigma_2)^1 \quad \Gamma \vdash_1 a : \sigma_1}{\Gamma \vdash_1 e a : \sigma_2} (\vdash_1\text{-APP})$$

$$\frac{\begin{array}{l} \Gamma, \overline{x_j : \sigma_j} \vdash_1 e_i : \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j} \vdash_1 e : \sigma \\ \left(\text{occur}(x_i, e) + \sum_{j=1}^n \text{occur}(x_i, e_j) \right) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \vdash_1 \text{letrec } x_i : \sigma_i = |\sigma_i|^\dagger e_i \text{ in } e : \sigma} (\vdash_1\text{-LETREC})$$

$$\frac{\Gamma \vdash_1 e : \sigma' \quad \sigma' \preceq \sigma}{\Gamma \vdash_1 e : \sigma} (\vdash_1\text{-SUB})$$

3.3 Usage well-typing rules

The next task is to define a set of *well-typing rules* that specify which LIX_1 typings carry valid usage information. The intuition of the type system is as follows:

We say a *variable* may be used more than once if either

- it occurs (*i.e.*, may be referred to) more than once in its scope, or
- it occurs free in a function abstraction which may be used more than once.

If a variable may be used more than once, the *value bound to that variable* may also be used more than once.

We give the resulting well-typing rules in Figure 3.2. Since the types of LIX_1 are simply those of L_0 augmented with usage information, the well-typing rules are those of L_0 (as given in Figure 2.2) augmented with extra side conditions constraining valid usage annotations and update flags, along with an additional rule, (\vdash_1 -SUB), for subsumption. In fact, a well-typed L_0 term may be translated into a well-typed LIX_1 term simply by placing ω -annotations everywhere on each t -type, yielding a σ -type, and letting all bindings be flagged by $!$ and all lambdas annotated by $\omega, !$. The judgement $\Gamma \vdash_1 e : \sigma$ may be read as stating that “In type environment Γ , the LIX_1 term e can be given type σ .” We deal only with the novel portions of the rules below. Note that $\omega \leq 1$; the reason for this will be explained in Section 3.3.5.

3.3.1 Update flags

Update flags, the output of the analysis, appear on bindings and lambda abstractions, and are computed in rules (\vdash_1 -LETREC) and (\vdash_1 -ABS) from the topmost annotation of the corresponding usage type. An annotation of ω indicates a value that might be used more than once, and so must be marked $!$, “updatable/copyable”; and an annotation of 1 indicates a value that is used at most once, and so may be marked \bullet , “not updatable/copyable”. The function \cdot^\dagger denotes this correspondence: $\omega^\dagger \triangleq !$, and $1^\dagger \triangleq \bullet$.

3.3.2 Basic uses

Literals and the results of primops⁵ are primitive values with no internal structure, and may safely be used many times, as shown in rules (\vdash_1 -LIT), (\vdash_1 -PRIMOP), and (\vdash_1 -PRIMOP-R). The latter rule can in fact be derived from the other two.

The scrutinee of an if0 and the operands of a primop are both used at most once (in fact, exactly once), and similarly in an application the function is used (applied) exactly once, as shown in rules (\vdash_1 -IF0), (\vdash_1 -PRIMOP) and (\vdash_1 -PRIMOP-R), and (\vdash_1 -APP).

⁵Primitive operations; see Section 2.2.

3.3.3 The syntactic occurrence function

In order to compute the usage according to the intuition stated above, we define a *syntactic occurrence* function to count the number of occurrences of a variable in its scope. By using this, rules (\vdash_1 -ABS) and (\vdash_1 -LETREC) compute the usage of each variable at its binding site. The value of $\text{occur}(x, e)$ is the number of free syntactic occurrences of the variable x in the expression e . For example, $\text{occur}(x, x + x + y) = 2$; $\text{occur}(x, \lambda x . x \ y) = 0$. If a variable occurs more than once in its scope, the topmost annotation of its type is required to be ω ; otherwise its type is unrestricted. An alternative method more directly related to affine linear logic is discussed in Section 3.9.7; our approach is closer to the “liabilities” of [GH90].

We define *occur* inductively; the only interesting case is *if0*, where we conservatively approximate by taking the maximum number of syntactic occurrences in either branch. Thus:⁶

$$\text{occur}(x, \text{if0 } e \text{ then } e_1 \text{ else } e_2) \triangleq \text{occur}(x, e) + \max(\text{occur}(x, e_1), \text{occur}(x, e_2))$$

We take the *maximum* (or least upper bound) of the branches because our usage annotations are *upper* bounds on the number of uses: 1 denotes at *most* once, and ω denotes no restriction. The type-based strictness analysis of Barendsen and Smetters [BS98] takes the *minimum* here, since strictness annotations indicate a *lower* bound. Appendix C shows how one might use more precise annotations to combine usage and strictness.

3.3.4 Occurrences in a closure

A further constraint on usage annotations is that if a variable y is free in a lambda abstraction (\vdash_1 -ABS), the topmost annotation $|\Gamma(y)|$ of the type of y is required to be less than the usage annotation of the abstraction. Recalling the annotation ordering $\omega \leq 1$ (Section 3.2.1), this means that if the lambda may be used more than once, so may its free variables – precisely the second clause of the intuition stated in Section 3.3 above. The reason for this is that y is *shared* between applications of the abstraction – each time the abstraction is applied, y may be demanded.

For example, consider the following expression:

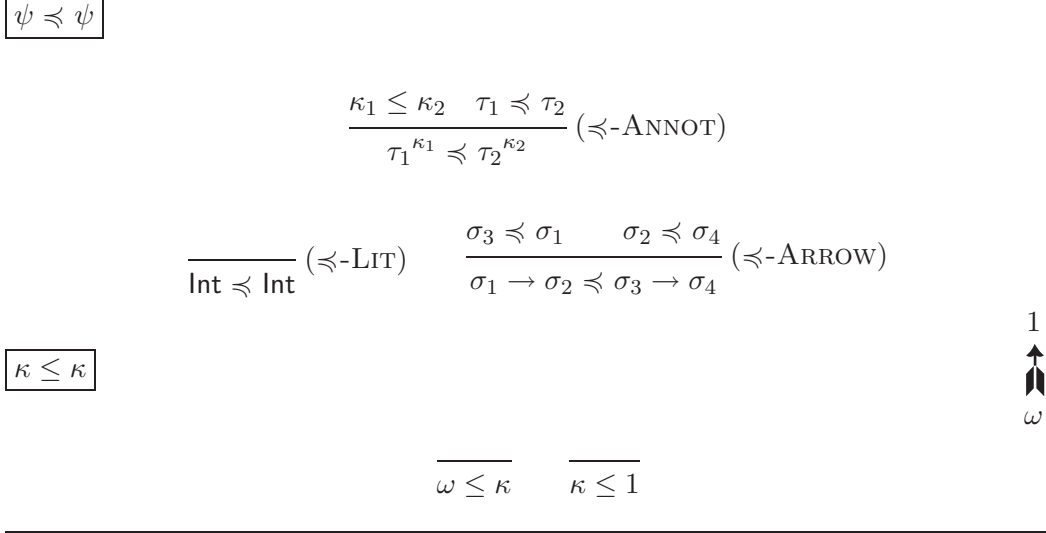
$$\begin{array}{l} \text{letrec } y : \text{Int}^\omega =! 1 + 2 \\ \text{in } \quad \text{letrec } f : (\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega =! \lambda^{\omega,!} x : \text{Int}^1 . x + y \\ \quad \text{in } \quad f \ 3 + f \ 4 \end{array}$$

Here y is shared between applications of the abstraction, and so is used twice during evaluation even though it occurs only once in its scope.

3.3.5 Subsumption

A distinctive feature of the present system compared with more conventional linear type systems, including [TWM95b], is our use of *subsumption*. Rule (\vdash_1 -SUB) states

⁶In the type system of Clean, this is referred to as “taking the evaluation order into account” [PvE98, §4.5.4] [dMJB⁺99, §4.3.6]. For a quasi-linear type system that takes the evaluation order into account rather more seriously, see [Kob99] (discussed briefly in Section 1.3.5).

Figure 3.3 The subtype (\preccurlyeq) and primitive (\leq) orderings over LIX_1 .

that an expression of type σ' can be used in place of an expression of type σ if those types are related as $\sigma' \preccurlyeq \sigma$ by the subtyping relation, defined in Figure 3.3. For example, an expression of type Int^ω may soundly be used in place of an expression of type Int^1 .

Consider the expression

```

letrec  f = λx . x + x
        a = 2 + 3
        b = 5 + 6
in      a + (f a) + (f b)

```

Here it is clear by inspection that the values of both a and b will be demanded more than once. We therefore expect both to be given type Int^ω . On the other hand, in

```

letrec  f' = λx . x + 1
        a  = 2 + 3
        b  = 5 + 6
in      a + (f' a) + (f' b)

```

it is clear that while a will be demanded twice, b will be demanded only once, and we expect a and b to be given types Int^ω and Int^1 , respectively.

Comparing the two, we see that applying f to an argument should constrain it to a used-many type (as for b), but applying f' to an argument should impose no constraint – it should be applicable equally to used-once and used-many arguments. We give f the type $(\text{Int}^\omega \rightarrow \text{Int}^1)^\omega$, and f' the type $(\text{Int}^1 \rightarrow \text{Int}^1)^\omega$, and in addition we note that while f is not applicable to an argument of type Int^1 , f' is applicable to an argument of type Int^ω .

This one-way notion of compatibility is exactly *subtyping*, and so we define a subtyping relation \preccurlyeq on usage types and add a subsumption rule ($\vdash_1\text{-SUB}$) to the set

of well-typing rules to obtain the desired behaviour. In terms of the example above, (\vdash_1 -SUB) allows us to conclude that argument a of type Int^ω also has type Int^1 and hence can be an argument of f' . Subsumption enables us to give to a function a type that conveys information about the function alone, irrespective of the arguments it is passed.

There is no need for an explicit coercion of the argument, because values of the subtype ‘just work’ in place of values of the supertype. This is sometimes known as *subset inclusion* or *containment subtyping* [Mit96, §10.4.3] [Gun92, § 9.1] as opposed to *coercion* or *conversion subtyping*, where values of a subtype must be explicitly converted to values of the supertype, as with $\text{Int} \preceq \text{Float}$ in machine arithmetic, or $S \preceq D$ in Mossin’s binding-time analysis [HM94, §5] [Mos93, §4.2].

The subtyping relation \preceq is the type ordering induced by the annotation ordering \leq , contravariant on function types as usual. We have $\omega \leq 1$, because a thunk annotated ω may be used any number of times, including once, while a thunk annotated 1 may only be used at most once – “ ω can be used in place of 1”. From this we may derive, for example:

$$\begin{aligned} \tau^\omega &\preceq \tau^1 \\ \tau_1^1 \rightarrow \tau_2^\omega &\preceq \tau_1^\omega \rightarrow \tau_2^1 \end{aligned}$$

and so on. Contravariance arises because of the duality between input and output: just as a (function returning a) ω -thunk may be used in place of a (function returning a) 1-thunk, a function expecting a 1-thunk may be used in place of a function expecting a ω -thunk. That is, $\omega \leq 1$ (rather than $1 \leq \omega$) leads to the correct subtype ordering.

Subsumption addresses a problem, which we call the *poisoning problem*, found in the work of Turner *et al.* [TWM95a]. For the second example above, their analysis infers the type Int^ω for b . Why? Clearly a ’s type must be Int^ω , since a is used more than once. So a non-subsumptive type system must attribute the type $(\text{Int}^\omega \rightarrow \text{Int}^1)^\omega$ to f' . But since f' is applied to b as well, b gets f' ’s argument type Int^ω . We call this the poisoning problem because one call to f' poisons all the others. We have seen that this problem does not arise in our system.

Crucially, the implementations we have in mind, and specifically the abstract machine described in Section 2.3, use *self-updating* thunks [PJ92, §3.1.2]. This means that thunks a and b in the example each carry information on whether they require to be updated, and f' ’s behaviour is independent of this information. In an implementation where the *consumer* of a thunk performs the update [BHY88], variable *occurrences* are annotated with update flags, and our subsumption rule would not be sound. Soundness can however be recovered by additional runtime support, as in Kobayashi’s system [Kob99, rem. 2.6].

3.3.6 Demands and recursive binding groups

The *letrec* construct defines a mutually-recursive binding group. In general the usages of the bound variables are interdependent: the body may demand certain variables, which may demand others, which may demand yet others, some of which may

already have been demanded. If we define $demanded(x_i)$ to be the number of times the variable x_i is actually demanded during evaluation of the letrec expression, then we may rewrite the occurrence-check clause in $(\vdash_1\text{-LETREC})$ as follows:⁷

$$demanded(x_i) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i$$

This says that for each variable, if it is demanded more than once then it must be annotated ω .

However, calculating $demanded(x_i)$ accurately at compile time is non-trivial (even over the restricted domain $\{0, 1, \omega\}$), and not really necessary. The $(\vdash_1\text{-LETREC})$ rule approximates $demanded(x_i)$ as follows:

$$demanded(x_i) \leq occur(x_i, e) + \sum_{j=1}^n occur(x_i, e_j)$$

This says that the demands on x_i during evaluation of the letrec are no more than the sum of its occurrences in the body and those in the right-hand sides of each binding. Since each right-hand side is evaluated at most once, this is a safe (over-) approximation; the inequality is strict only when one or more bindings in the letrec are dead (*i.e.*, never demanded). For letrec $\langle x = y + 1; y = 3; z = y + 2 \rangle$ in x the approximation yields $demanded(y) \leq \omega$, but since z is dead⁸ the only demand comes from x 's right-hand side, and the true value is 1. Since in practice other phases of compilation can be expected to remove dead bindings, this approximation is reasonable.⁹

3.4 The goodness ordering

With the well-typing rules of the previous section, many L_0 programs have more than one valid LIX_1 typing. For example, the program letrec $x : \text{Int} = 5$ in $x + 2$ may be annotated either as letrec $x : \text{Int}^1 = 5$ in $x + 2$ or as letrec $x : \text{Int}^\omega = 5$ in $x + 2$. Both have the same L_0 types, and both are well- LIX_1 -typed, but obviously we prefer the

⁷Not counting uses arising indirectly from applications, which depend instead on rules $(\vdash_1\text{-ABS})$ and $(\vdash_1\text{-APP})$ for correct annotation.

⁸“Whose chopper is this?” “Zed’s” “Who’s Zed?” “Zed’s dead, baby; Zed’s dead.” – Fabienne and Butch, *Pulp Fiction* [Var94, track 8] [Tar94, p. 135].

⁹We could be more accurate and obtain the correct demand for y above as follows. Write \triangleright for the guard function $n \triangleright m = (n = 0 ? 0 : m)$:

\triangleright	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	1	ω

Then the vector $\overline{demanded(x_i)}$ may be defined as the pointwise least fixed point of:

$$demanded(x_i) \triangleq occur(x_i, e) + \sum_{j=1}^n demanded(x_j) \triangleright occur(x_i, e_j)$$

This more accurate formulation turns out to be useful later in Appendix C.

first typing, because the first allows us to avoid updating the binding for x whereas the second does not. In this section, we define a *goodness ordering* \sqsubseteq that formalises our intuition as to which typing is the best.

3.4.1 Intuition

The first typing in the example above is better than the second because the update flag χ on the binding is \bullet (“not updatable”) rather than $!$. Since the update flag is computed from the corresponding usage annotation κ , this means that for bindings the best annotation is the *maximum* annotation: 1, since $\omega \leq 1$. Similarly for abstractions: the best update flag is \bullet (“not copyable”) and hence the best annotation is 1. None of the other usage annotations in the term or types are relevant, since only these two affect the update flags and hence subsequent execution.

3.4.2 Existence

Now observe that in the well-typing rules, we impose no negative constraints on annotations: increasing one annotation may require other annotations to increase, but will never require them to decrease (*i.e.*, all constraints are covariant). This means that there is always a typing that (pointwise) maximises the annotations of interest; in fact, there is always a typing that (pointwise) maximises *all* annotations, making as many as possible 1. We define this to be the *best* typing, and call the pointwise extension of \leq to types (covariant on function types) the *goodness ordering* \sqsubseteq .

We can formally prove the existence of a best solution using the constraint theory of Sections 3.5.1 and D.3 below (*q.v.*). Briefly, a solution of a constraint C is defined to be a substitution S such that $\vdash^e SC$, *i.e.*, for all κ, κ' , we have $C \vdash^e \langle \kappa \leq \kappa' \rangle \Rightarrow S\kappa \leq S\kappa'$. The set of solutions $\mathcal{S}_C = \{S \mid \vdash^e SC\}$ of a given constraint C can be partially ordered under the obvious pointwise ordering $S \sqsubseteq S' = \forall u. Su \leq S'u$. Furthermore, the poset $(\mathcal{S}_C, \sqsubseteq)$ is lub-closed: if $\vdash^e SC$ and $\vdash^e S'C$ then $\vdash^e (S \sqcup S')C$.¹⁰ (In other words, \mathcal{S}_C is a *Moore family* [NNH99, §3.2.3].) Thus if there exists any solution at all, there exists a greatest solution under the pointwise ordering.

3.4.3 Covariance of \sqsubseteq

The goodness ordering is covariant on function types:

$$\sigma_1 \sqsubseteq \sigma'_1 \wedge \sigma_2 \sqsubseteq \sigma'_2 \Rightarrow (\sigma_1 \rightarrow \sigma_2) \sqsubseteq (\sigma'_1 \rightarrow \sigma'_2)$$

This is in contrast with the subtype ordering:

$$\sigma'_1 \preceq \sigma_1 \wedge \sigma_2 \preceq \sigma'_2 \Rightarrow (\sigma_1 \rightarrow \sigma_2) \preceq (\sigma'_1 \rightarrow \sigma'_2)$$

¹⁰**Proof** Consider a constraint $\langle \kappa \leq \kappa' \rangle$ in C . By assumption we have $S\kappa \leq S\kappa'$ and $S'\kappa \leq S'\kappa'$. We wish to prove $(S \sqcup S')\kappa \leq (S \sqcup S')\kappa'$. Observe that $(S \sqcup S')\kappa = (S\kappa \sqcup S'\kappa) \in \{S\kappa, S'\kappa\}$. Now $S\kappa \leq S\kappa' \leq (S\kappa' \sqcup S'\kappa') = (S \sqcup S')\kappa'$, and similarly $S'\kappa \leq S'\kappa' \leq (S\kappa' \sqcup S'\kappa') = (S \sqcup S')\kappa'$. The result follows by transitivity of \leq . \square

and may be somewhat surprising. To see why this is correct, consider the usage annotations on a function f with type $(\tau_1^{\kappa_1} \rightarrow \tau_2^{\kappa_2})^\omega$:

- We want to maximise κ_2 , because it may annotate a lambda or a letrec binding, and 1 will yield better results than ω .
- We want to maximise κ_1 , because if this function is exported it should place the least demand possible on the caller. (We need consider only exports because within a single module, maximising the argument's topmost annotation will force κ_1 upwards anyway.)

While maximising κ_1 is consistent with the subtype ordering, maximising κ_2 is in opposition to it. This is because we want not the most general or usable type but the one that is most informative about the context (Section 3.7.2). In a sense, for the present inference information flows from context to term, whereas for conventional type systems information flows from input to output. The goodness ordering reflects this flow.

3.5 Usage inference

The final part of our type-based analysis is the *inference algorithm* \mathcal{IT}_1 , which must compute the best valid LIX_1 typing for any L_0 program. That is, when presented with an L_0 program, this algorithm must infer an equivalent LIX_1 program which is well-typed according to the rules of Section 3.3; and if there is more than one such, it should choose the one that is the ‘best’ according to the goodness ordering of Section 3.4.

The well-typing rules in Figure 3.2 include a number of side conditions that constrain usage annotations. For example, the second condition of the $(\vdash_1\text{-ABS})$ rule states that if the bound variable occurs syntactically more than once in the body, the topmost annotation of its type must be ω . Treated as rules for *checking* a typing, these are trivial to implement; but the *construction* of such a typing *ex nihilo* is not so trivial. Many such constraints may apply to a single annotation, and annotations may be interdependent.

Such difficulties can be overcome in a general and powerful way. Whenever we are required to provide an annotation that is not at present completely determined, we simply generate a fresh *usage variable* for that annotation. Whenever we encounter a constraint that cannot yet be checked because not all its annotations are ground, we record the constraint and proceed. This first phase yields a pair: an expression with free usage variables, and a set of constraints over those variables. Then in a second phase we find values for all the variables such that all the constraints are satisfied. Since all the side conditions are satisfied by these values, the ground typing resulting from applying this substitution satisfies the well-typing rules. Indeed, in general there is more than one such substitution, and we may choose the best one as defined in Section 3.4.

This section begins by introducing the constraint notation we use (Section 3.5.1). The first phase \blacktriangleright_1 of the inference is described in Section 3.5.2. A *pessimisation*

Figure 3.4 Type inference rules from L_0 to LIX_1 .

<div style="border: 1px solid black; padding: 5px; width: fit-content;"> <div style="display: flex; flex-direction: column; align-items: center;"> <div style="margin-bottom: 5px;">— environment, LIX_1</div> <div style="margin-bottom: 5px;">— source term, L_0</div> <div style="margin-bottom: 5px;">— target term, LIX_1</div> <div style="margin-bottom: 5px;">— target type, LIX_1</div> <div style="margin-bottom: 5px;">— constraint</div> <div style="margin-bottom: 5px;">— occurrence multiset</div> </div> $\Gamma \blacktriangleright_1 M \rightsquigarrow e : \sigma; C; V$ </div>	$\frac{}{\Gamma, x : \sigma \blacktriangleright_1 x \rightsquigarrow x : \sigma; \emptyset; \{x\}} (\blacktriangleright_1\text{-VAR})$
	$\frac{}{\Gamma \blacktriangleright_1 n \rightsquigarrow n : \text{Int}^\omega; \emptyset; \{n\}} (\blacktriangleright_1\text{-LIT})$
	$\frac{\begin{array}{l} \Gamma \blacktriangleright_1 M \rightsquigarrow e : \text{Int}^\kappa; C; V \\ \Gamma \blacktriangleright_1 M_i \rightsquigarrow e_i : \sigma_i; C_i; V_i \quad i = 1, 2 \\ (C_3, \sigma) = \text{FreshLUB}(\sigma_1, \sigma_2) \end{array}}{\Gamma \blacktriangleright_1 \text{if0 } M \text{ then } M_1 \text{ else } M_2 \rightsquigarrow \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma; C \wedge C_1 \wedge C_2 \wedge C_3; V \uplus (V_1 \sqcup V_2)} (\blacktriangleright_1\text{-IF0})$
	$\frac{\Gamma \blacktriangleright_1 M_i \rightsquigarrow e_i : \text{Int}^{\kappa_i}; C_i; V_i \quad i = 1, 2}{\Gamma \blacktriangleright_1 M_1 + M_2 \rightsquigarrow e_1 + e_2 : \text{Int}^\omega; C_1 \wedge C_2; V_1 \uplus V_2} (\blacktriangleright_1\text{-PRIMOP})$
	$\frac{\begin{array}{l} \sigma_1 = [t_1]_\sigma^{\text{fresh}} \quad \text{fresh } v \\ \Gamma, x : \sigma_1 \blacktriangleright_1 M \rightsquigarrow e : \sigma_2; C_1; V \\ C_2 = \{V(x) > 1 \Rightarrow \langle \sigma_1 = \omega \rangle\} \\ C_3 = \bigwedge_{y \in \Gamma} \{V(y) > 0 \Rightarrow \langle \Gamma(y) \leq v \rangle\} \end{array}}{\Gamma \blacktriangleright_1 \lambda x : t_1 . M \rightsquigarrow \lambda^{v, v^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^v; C_1 \wedge C_2 \wedge C_3; V \setminus \{x\}} (\blacktriangleright_1\text{-ABS})$
	$\frac{\begin{array}{l} \Gamma \blacktriangleright_2 M \rightsquigarrow e : (\sigma_1 \rightarrow \sigma_2)^\kappa; C_1; V_1 \\ \Gamma \blacktriangleright_2 A \rightsquigarrow a : \sigma'_1; C_2; V_2 \\ C_3 = \{\sigma'_1 \preceq \sigma_1\} \end{array}}{\Gamma \blacktriangleright_2 M A \rightsquigarrow e a : \sigma_2; C_1 \wedge C_2 \wedge C_3; V_1 \uplus V_2} (\blacktriangleright_1\text{-APP})$
	$\frac{\begin{array}{l} \sigma_i = [t_i]_\sigma^{\text{fresh}} \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j} \blacktriangleright_1 M_i \rightsquigarrow e_i : \sigma'_i; C_1^i; V_i \quad \text{for all } i \\ C_1 = \bigwedge_i (C_1^i \wedge \{\sigma'_i \preceq \sigma_i\}) \\ \Gamma, \overline{x_j : \sigma_j} \blacktriangleright_1 M \rightsquigarrow e : \sigma; C_2; V \\ C_3 = \bigwedge_i \{(V(x_i) + \sum_j V_j(x_i)) > 1 \Rightarrow \langle \sigma_i = \omega \rangle\} \end{array}}{\Gamma \blacktriangleright_1 \text{letrec } \overline{x_i : t_i = \overline{M_i}} \text{ in } M \rightsquigarrow \text{letrec } \overline{x_i : \sigma_i = \sigma_i ^\dagger} e_i \text{ in } e : \sigma; C_1 \wedge C_2 \wedge C_3; (\biguplus_i V_i \uplus V) \setminus \{\overline{x_i}\}} (\blacktriangleright_1\text{-LETREC})$

pass P_{ess} , presented in Section 3.5.3, permits use of the inference for separately-compiled modules. Finally in Section 3.5.4 we pass to the second phase \mathcal{CS} of the inference, constraint solution. The combination of these three parts yields the complete inference:

$$\mathcal{IT}_1(\Gamma, M) \triangleq (\mathcal{CS} \circ P_{ess} \circ \blacktriangleright_1)(\Gamma, M) = (e, \sigma)$$

where $e : \sigma$ is the best well-typed LIX_1 term corresponding to M .

3.5.1 Constraints

Informally, a *constraint* is simply a set of equalities $\langle \kappa = \kappa' \rangle$ or inequalities $\langle \kappa \leq \kappa' \rangle$ which constrain the valid assignments to the variables. Constraints are combined with \wedge , and the trivial constraint is denoted \emptyset . We write $\{P \Rightarrow C\}$ to abbreviate $\begin{cases} C & \text{if } P \\ \emptyset & \text{otherwise} \end{cases}$; note that this is *not* a conditional constraint, since P must be statically known. A *solution* S to a constraint is an assignment (or *substitution*) that satisfies all the equalities and inequalities. The constraint C *entails* the constraint D , written $C \vdash^e D$, iff all solutions of C are also solutions of D . Two constraints are considered *equal* if each entails the other. We abbreviate the statement “ S is a solution to C ” by writing $\vdash^e SC$. Full details are given in the appendix, Section D.3.

3.5.2 Inference algorithm phase 1

The first phase of the inference algorithm, \blacktriangleright_1 , converts an L_0 program into an LIX_1 program with usage *variables* in place of ground usage annotations, along with a *constraint* on these variables describing valid assignments. The algorithm proceeds inductively on the syntax of the input program.

This phase is defined in Figure 3.4. The figure defines a relation

$$\Gamma \blacktriangleright_1 M \rightsquigarrow e : \sigma; C; V$$

which may be read “In the LIX_1 type environment Γ , the L_0 term M translates to LIX_1 term e , which has type σ , generated constraints C , and free term variables V .” Notice that \blacktriangleright_1 may be viewed as a function: Γ and M are inputs (inherited attributes), and e , σ , C , and V are outputs (synthesized attributes). The multiset¹¹ of free term variables V is used to count syntactic occurrences, implementing the *occur* function of Section 3.3.3 without an additional pass over the term. We denote usage variables by u, v , and permit them to occur wherever a κ annotation is permitted.

The well-typing rules of Figure 3.2 need little modification to be used as inference rules. As is standard for inference algorithms [Mit96, lemma 10.4.9], we restrict uses of $(\vdash_1\text{-SUB})$ to the argument derivation of $(\blacktriangleright_1\text{-APP})$, the condition and branches of $(\blacktriangleright_1\text{-IF0})$, arguments of $(\blacktriangleright_1\text{-PRIMOP})$, and bindings of $(\blacktriangleright_1\text{-LETREC})$. Constraints

¹¹A multiset [Bli89] is a function taking each possible element to a natural number multiplicity. The empty multiset is denoted \emptyset , the singleton multiset $\{x\}$, multiset union (addition of multiplicities) by $V_1 \uplus V_2$, multiset lub (pointwise maximum multiplicity) by $V_1 \sqcup V_2$, and deletion of a *set* of elements from a multiset by $V \setminus A$, where $(V \setminus A)(x) = 0$ if $x \in A$ and $V(x)$ otherwise.

are generated at these points by using the rules of Figure 3.3 in reverse: to obtain the conclusion, we must generate the constraints required by the antecedents (reading $\langle \kappa \leq \kappa' \rangle$ for $\kappa \leq \kappa'$). The result is a syntax-directed rule set, and hence an algorithm.¹²

The technical details of fresh variable generation have been omitted for clarity, but are entirely standard. The operation “fresh v ” chooses a fresh usage variable v . The operation $\lceil t \rceil_\sigma^{\text{fresh}}$ returns a σ -type of the same shape as the t -type t , but with a fresh usage variable in each annotation position; e.g., $\lceil \text{Int} \rightarrow \text{Int} \rceil_\sigma^{\text{fresh}} = (\text{Int}^{u_1} \rightarrow \text{Int}^{u_2})^{u_3}$. The operation *FreshLUB* takes a vector of types and returns a constraint and a type (possibly containing fresh usage variable annotations) such that under the constraints, the result type is the least upper bound of the arguments.¹³ For example

$$\text{FreshLUB}((\text{Int}^1 \rightarrow \text{Int}^{u_1})^1, (\text{Int}^\omega \rightarrow \text{Int}^{u_2})^\omega) = (\{\langle u_1 \leq u_3 \rangle, \langle u_2 \leq u_3 \rangle\}, (\text{Int}^\omega \rightarrow \text{Int}^{u_3})^1)$$

where u_3 is fresh.

The update flags χ on lambdas and bindings are dependent on the corresponding ground usage annotations, according to the function \cdot^\dagger , defined in Section 3.3.1. Since their values depend on the *ground* values of the corresponding annotations, they must be computed *after* the substitution performed at the end of the second phase (Section 3.5.4); for simplicity of presentation we disregard this issue and write, e.g., v^\dagger directly in $(\blacktriangleright_1\text{-ABS})$, intending this to be computed after the final substitution. Since the substitution chosen by our algorithm takes all free usage variables to either 1 or ω , the value of \cdot^\dagger is well-defined everywhere it is used.

3.5.3 Pessimisation

The analysis must in practice deal with separate compilation of modules. Specifically, it must allow for uses by subsequent modules of variables exported from the current module, even though these uses are not visible to the analysis. We handle this problem with a *pessimisation* step in the analysis,¹⁴ performed after inferring the constraints but before finding a solution to them.

Pessimisation (which we discuss in detail in Section 3.8) requires adding a set of *pessimising constraints* to C . These constraints simulate all possible future uses of exported variables, by forcing all positive annotations in types of exported variables to ω . Let Δ be the type environment consisting of only those variables which are exported from the current module. Then $\text{Pess}(\Delta) \triangleq \{\text{Pess}^+(\sigma) \mid \sigma \in \text{rng}(\Delta)\}$, where

$$\begin{aligned} \text{Pess}^+(\tau^\kappa) &= \text{Pess}^+(\tau) \wedge \langle \kappa = \omega \rangle \\ \text{Pess}^-(\tau^\kappa) &= \text{Pess}^-(\tau) \\ \text{Pess}^\varepsilon(\sigma \rightarrow \sigma') &= \text{Pess}^\varepsilon(\sigma) \wedge \text{Pess}^\varepsilon(\sigma') \\ \text{Pess}^\varepsilon(\text{Int}) &= \emptyset \end{aligned}$$

¹²This clear explanation of the inference process is taken more-or-less verbatim from [RF01, §5.1].

¹³In other words, $(C, \sigma) = \text{FreshLUB}(\sigma_1, \sigma_2)$ only if $C \vdash^e \langle \sigma_1 \leq \sigma \rangle$ and $C \vdash^e \langle \sigma_2 \leq \sigma \rangle$, and for all σ' , if $C \vdash^e \langle \sigma_1 \leq \sigma' \rangle$ and $C \vdash^e \langle \sigma_2 \leq \sigma' \rangle$ then $C \vdash^e \langle \sigma \leq \sigma' \rangle$. The definition is left as an exercise.

¹⁴Thanks are due to David N. Turner and Clem Baker-Finch for this apt term; Joshua Lawrence directed the author to a prior use in *The Story of Mel* [Ray91, pp. 406ff].

Then the constraint passed to the constraint solving algorithm below is actually C' , where $C' = C \wedge \text{Pess}(\Delta)$. For example, let $\Delta = \{f : ((\text{Int}^{u_1} \rightarrow \text{Int}^{u_2})^{u_3} \rightarrow \text{Int}^{u_4})^{u_5}, g : \text{Int}^{u_6}\}$. Then $\text{Pess}(\Delta) = \{\langle u_5 = \omega \rangle, \langle u_1 = \omega \rangle, \langle u_4 = \omega \rangle, \langle u_6 = \omega \rangle\}$.

3.5.4 Inference algorithm phase 2

Recall that the first phase has obtained from the source term M and environment Γ a triple (e', σ', C') representing the space of possible annotations of M (the set V is of no further interest). We must choose the (unique) best solution S as defined in Section 3.4, and apply it to e' and σ' to obtain a ground LIX_1 term Se' and a ground type $S\sigma'$, the results of the analysis.

The algorithm \mathcal{CS} takes as input a set $\bigwedge_{i=1}^n C_i$ of atomic constraints, each either $\langle \kappa = \kappa' \rangle$ or $\langle \kappa \leq \kappa' \rangle$. It outputs an assignment of values 1 or ω to the usage variables $\overline{u_i}$ such that as many as possible are set to 1 while satisfying all the constraints, or fails if the constraints are unsatisfiable.¹⁵

We make use of a union-find [Tar75] [TvL84] [CLR90, §22.3] data structure to maintain a set of equivalence classes of usage variables, with two operations: $\text{FIND}(u_i)$ to find the distinguished (root) variable in the class of which u_i is a member, and $\text{UNION}(u_i, u_j)$ to merge the equivalence classes of which u_i and u_j are members. In addition, we maintain a finite map from the root variables of the equivalence classes to either a constant usage annotation 1 or ω , or a pair $(\overline{u_k}, \overline{v_l})$ of sets of variables bounding the equivalence class from below and above respectively. Initially each variable belongs to a singleton equivalence class of its own, and every equivalence class maps to the pair (\emptyset, \emptyset) of empty bounds.

We may interpret the data structure as a constraint in the following way. Firstly, interpret each equivalence class having k members as a conjunction of $k - 1$ atomic equality constraints forming a spanning tree of members of the class. Secondly, interpret each mapping from a root variable u_i to a constant κ as an equality constraint $\langle u_i = \kappa \rangle$. Thirdly, interpret each mapping from a root variable u_i to a pair of bounds $(\overline{u_k}, \overline{v_l})$ as a conjunction of inequality constraints $\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle$. The conjunction of these three sets of constraints is the constraint denoted by the data structure. Algorithm failure is interpreted as the unsatisfiable constraint $\langle 1 \leq \omega \rangle$. Further input is permitted after failure, but yields only repeated failure.

The algorithm considers each atomic constraint C_1, C_2, \dots in turn, using it to update the data structure in such a way that after stage m , the denotation of the data structure is equal to the conjunction $\bigwedge_{i=1}^m C_i$ of constraints seen so far, and thus the conjunction of the data structure and the remaining constraints $\bigwedge_{i=m+1}^n C_i$ is equal to the complete set of constraints. Once the algorithm is complete ($m = n$), the solution may simply be read off the data structure. Alternatively, at any point the algorithm may fail, thus indicating that no solution exists.

¹⁵Thanks are due to Fritz Henglein for pointing out to the author that this problem is simply reachability. Consider a directed graph with vertices 1, ω , and the usage variables of the constraint, and for each constraint $\langle \kappa \leq \kappa' \rangle$ an edge from κ to κ' (equality constraints are represented by a pair of edges). Then if vertex 1 is reachable from vertex ω , the constraint is insoluble; otherwise, each variable reachable from ω should be set to ω , and the remainder to 1. This can be solved by a simple worklist algorithm, in time linear in the number of edges.

Although there are eighteen different types of constraint (κ and κ' may each be 1, ω , or a variable, and the constraint may be equality or inequality), we need consider only four: $\langle u_i = 1 \rangle$, $\langle u_i = \omega \rangle$, $\langle u_i = u_j \rangle$, and $\langle u_i \leq u_j \rangle$. All the others are either equivalent to one of these, trivial (e.g., $\langle u_j \leq 1 \rangle$), or cause immediate failure (e.g., $\langle 1 \leq \omega \rangle$).

For each atomic constraint, the algorithm first reduces it to one of the six cases above. If it is one of the four nontrivial constraint forms, it begins by replacing each variable in the constraint with the root variable of the equivalence class to which it belongs. It then proceeds as follows:

- $\langle u_i = 1 \rangle$ or $\langle u_i = \omega \rangle$: if u_i maps to 1 or ω respectively, then do nothing. If u_i maps to ω or 1 respectively, then fail (we have found a conflict). Otherwise, u_i maps to $(\overline{u_k}, \overline{v_l})$; update the map so u_i maps to 1 or ω respectively, and recursively add the constraints $\langle v_l = 1 \rangle$ or $\langle u_k = \omega \rangle$ respectively.
- $\langle u_i = u_j \rangle$: if u_i maps to 1 or ω , recursively add the constraint $\langle u_j = 1 \rangle$ or $\langle u_j = \omega \rangle$ respectively; similarly for u_j . Otherwise u_i maps to $(\overline{u_k}, \overline{v_l})$ and u_j maps to $(\overline{u'_k}, \overline{v'_l})$; merge the two equivalence classes and map the root of the combined equivalence class to $(\overline{u_k u'_k}, \overline{v_l v'_l})$, i.e., the lower (upper) bound is the union of the lower (upper) bounds of u_i and v_j .
- $\langle u_i \leq u_j \rangle$: if u_i maps to 1 or ω , recursively add the constraint $\langle u_j = 1 \rangle$ or do nothing respectively; dually for u_j . Otherwise u_i maps to $(\overline{u_k}, \overline{v_l})$ and u_j maps to $(\overline{u'_k}, \overline{v'_l})$; update the map so that u_i maps to $(\overline{u_k}, \overline{u_j v'_l})$ and u_j maps to $(\overline{u_i u'_k}, \overline{v'_l})$.

Once all the constraints have been considered, the data structure maps each variable (by equivalence class) either to a constant or to a pair of bounds.

Now we are in a position to determine the optimal solution. If a variable is mapped to a constant, then clearly it must take that value. If a variable is mapped to a pair of bounds, it may take either 1 or ω as value, so long as this is consistent with the bounds. It is clearly safe to set all such floating variables to the same value; if we choose 1 then we obtain the *best* solution (the most 1s possible, Section 3.4). We thus derive from the data structure a substitution taking each variable to either 1 or ω in a way consistent with the constraint $\bigwedge_{i=1}^n C_i$ and optimal with respect to the goodness ordering.

Having derived the optimal substitution S from C' , algorithm \mathcal{CS} finally applies it to e' and σ' , returning the pair $(Se', S\sigma')$ as result.

The constraint solver described in this section is used again in the inference system of Chapter 4, to compute both the final solution and the intermediate transitive closures required for generalisation. In the latter case, we extend the solver by permitting access to the internal data structure and reification of the data structure as a constraint. We discuss this further in Section 4.5.5.

The algorithm makes essential use of the fact that the domain has only two points; were the usage annotation domain to be extended (as, e.g., Appendix C), it would have to be replaced with a more general constraint solver.

3.6 Proofs

For the type-based usage analysis we have so far described to be correct, recall from Section 3.1 that certain proofs are required. We present these proofs in outline below. In Section 3.6.1 we prove that the well-typing rules are sound with respect to our operational definition of usage, and that all L_0 terms have a valid LIX_1 typing. In Section 3.6.2 we prove that the inference algorithm computes a valid LIX_1 typing for every source term. In Section 3.6.3 we prove that the result of the inference algorithm is in fact the best solution according to the goodness ordering. Additionally, we prove a complexity bound for the inference algorithm (Section 3.6.4). We use the same notation as Chapter 2.

3.6.1 Well-typing rules

To be correct, the well-typing rules must guarantee the desired properties: that

- the computed χ -annotations are correct, *i.e.*, respected by the execution, and
- the result of executing a well-typed target program is the same as that of executing the corresponding source program.

Since the operational semantics has been designed to validate the χ -annotations by failing if they are invalid (Section 2.3), the first property may be demonstrated by showing that the well-typing rules are sound with respect to the operational semantics. As in Section 2.5, this means that a well-typed LIX_1 term e must never terminate in a configuration in *Wrong*, *BadBinding*, or *BadValue*; if it terminates it must do so in a configuration in *Value* or *BlackHole*.

The operational semantics we use for LIX_1 is simply that of LIX_0 , modified appropriately in order to carry τ - and σ -types as instrumentation rather than t -types and to ignore κ annotations on lambdas. An analogue of the Correspondence Lemma 2.4 states that the instrumentation is ignored, *i.e.*, that LIX_1 and LX execute in lock-step. The results are as follows.

Theorem 3.1 (Type soundness)

For all $e \in LIX_1$, if $\emptyset \vdash_1 e : \sigma$ and there exists a configuration C' such that $(e)^{\langle :: \rangle} \downarrow C'$, then $C' \in \text{Value} \cup \text{BlackHole}$.

Proof By Progress Lemma D.10 (proven by cases on C), and induction on the length of derivation of $(e)^{\langle :: \rangle} \downarrow C'$. The full proof is given in the appendix, Theorem D.11. \square

Progress Lemma D.10 states that the configuration remains well-typed after a reduction step. It follows that it remains closed, and this is equivalent to the don't-drag side condition of Gustavsson and Sands on $(\rightarrow\text{-VAR-ONCE})$, described in Section 2.6, that if a variable is used at most once, no references to it remain after it is used.

The second property makes use of the stripping operation $(e)^{\natural} = M$ (defined in Section 3.2.2), which takes an LIX_1 term to the corresponding L_0 term by stripping

both all update flags and all usage annotations. We must show that e gives the same result as M , i.e., if M terminates with result V (or a black hole), then e terminates with the same result (or a black hole, respectively), and *vice versa*. It is sufficient to observe results at ground types only (here just Int).

Theorem 3.2 (Correctness)

For all LIX_1 target programs e , where $\emptyset \vdash_1 e : \sigma$, let M be the corresponding L_0 source program $(e)^\natural$. Then we have

- (i) $(e)^{\langle \cdot \rangle} \downarrow \Leftrightarrow (\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow$
(i.e., the LIX_1 program e terminates iff the L_0 program M does); and
- (ii) If $(e)^{\langle \cdot \rangle} \downarrow C'$ and $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow C''$, then all the following hold:
 - (a) $C' \in \text{BlackHole} \Leftrightarrow C'' \in \text{BlackHole}$
 - (b) $C' \in \text{Value} \Leftrightarrow C'' \in \text{Value}$
 - (c) $C' = \langle H; n; \varepsilon \rangle \Leftrightarrow C'' = \langle H'; n; \varepsilon \rangle$

(i.e., if the two programs terminate, they both terminate in the same way, viz., black hole, non-ground value, or the same ground value).

Proof By the Correspondence Lemma we may ignore the instrumentation, and consider the two FLX terms $M_1 = (e)^\flat$ and $M_0 = (M)^\flat$. The two directions of (i) are proven separately, showing by induction on the length of the respective reduction sequence that each can simulate the other and relating terminal configurations. The full proof appears in the appendix, Theorem D.17. \square

We also must show that all source terms have a target typing:

Theorem 3.3 (Nonrestrictivity)

For all L_0 environments Γ , terms M , and types t , if $\Gamma \vdash_0 M : t$ then there exists an LIX_1 environment Γ' , term e , and type σ such that $(\Gamma')^\natural = \Gamma$, $(e)^\natural = M$, $(\sigma)^\natural = t$, and $\Gamma' \vdash_1 e : \sigma$.

Proof Let $\Gamma' = \lceil \Gamma \rceil^\omega$, $e = \lceil M \rceil^\omega$, and $\sigma = \lceil t \rceil_\sigma^\omega$, where $\lceil \cdot \rceil^\omega$ extends $\lceil \cdot \rceil_\sigma^\omega$ to typing environments and terms, placing ω annotations and $!$ flags everywhere (see Section 3.3). \square

Together, these three results strongly support our claim that the type system models our operational notion of usage. Theorem 3.1 demonstrates that well-typed programs do not go wrong: the analysis yields operationally correct update flags. Theorem 3.2 demonstrates that the observable behaviour of a well-typed LIX_1 program is identical to that of its corresponding L_0 program: the analysis does not affect behaviour. And Theorem 3.3 reassures us that there is at least one corresponding LIX_1 program for every L_0 program: the analysis is *soft* (see Section 1.5.5), and does not reject any program. However, these results do not tell us whether or not the type system chooses *good* update flags; for this we must rely on practical experience, which we obtain in Section 3.7.

3.6.2 Inference phase 1

There are two desirable properties of an inference (with respect to the well-typing rules). *Soundness* states that if a term is typeable, the inference yields a well-typing for it. *Completeness*, on the other hand, states that *all* well-typings for the term are instances of the result of the inference.

The inference algorithm described in Figure 3.4 is sound and complete with respect to the well-typing rules of Figure 3.2. That is, for every well-typed L_0 term \blacktriangleright_1 yields a constraint with at least one solution, every solution yields a well-typed LIX_1 term, and all well-typed annotations of the L_0 term appear in the set of solutions.

In stating this formally we make use of Theorem 3.3 to justify quantifying over all LIX_1 terms rather than all L_0 terms; the theorem states that every L_0 term has at least one corresponding LIX_1 term. We state our claims formally as follows.

Theorem 3.4 (Soundness and completeness of inference phase 1)

For all Γ in LIX_1 and M, t in L_0 such that $(\Gamma)^\sharp \vdash_0 M : t$ and $1 \notin \text{ann}^+(\Gamma)$,

- (i) $\blacktriangleright_1 (\Gamma, (e)^\sharp) = (e', \sigma', C, V)$ is well defined¹⁶
(i.e., the algorithm \blacktriangleright_1 is deterministic and does not fail);
- (ii) $(e')^\sharp = (e)^\sharp$ and $(\sigma')^\sharp = (\sigma)^\sharp$
(i.e., the inference algorithm merely annotates the source term, and does not alter it or its source type);
- (iii) $\forall S. \vdash^e SC \Rightarrow \Gamma \vdash_1 Se' : S\sigma'$
(i.e., all solutions of the resulting constraint are well-typed); and
- (iv) For all e, σ such that $(e)^\sharp = M$, $(\sigma)^\sharp = t$, and $\Gamma \vdash_1 e : \sigma$, there exists a substitution S such that $\vdash^e SC$ and $Se' = e$, $S\sigma' = \sigma$.
(i.e., all well-typed annotations of the source term are solutions of the resulting constraint).

Proof Proofs of (i) and (ii) are straightforward. (iii) is proved by induction over the structure of the inference derivation tree and inspection of each inference rule, comparing it with the corresponding well-typing rule. (iv) follows from the syntax-directed nature of the rules, and a similar argument to (iii). A full proof appears in the Appendix, Section D.17. \square

3.6.3 Inference phase 2

The constraint solution algorithm described in Section 3.5.4 always terminates with a solution if one exists. Furthermore, the solution returned is the unique best solution determined by the criteria of Section 3.4. However, the solution is not necessarily principal (see Section 3.7.2).

Theorem 3.5 (Soundness and pseudo-completeness of inference phase 2)

For all constraints C ,

¹⁶It is well defined modulo the names of fresh variables; we continue to omit details of fresh variable management.

- (i) Algorithm CS terminates.
- (ii) If there exists an S such that $\vdash^e SC$, then CS succeeds.
- (iii) If CS succeeds with substitution S , then $\vdash^e SC$, and for all S' such that $\vdash^e S'C$, $S' \sqsubseteq S$ (i.e., S is the best solution to C).

Proof (i) follows from Theorem 3.6 below. (ii) and (iii) follow from consideration of the invariant stated in Section 3.5.4, namely that the data structure is an isomorphic representation of the constraint. A full proof appears in the appendix, Section D.19. \square

The algorithm is very efficient, due to the simple nature of the constraints:

Theorem 3.6 (Complexity of constraint solver)

Algorithm CS has essentially $O(1)$ amortized cost per constraint.

We prove this amortized complexity result using the *accounting method* [CLR90, §18.2]. By reifying costs as dollars and attaching them to parts of the data structure, we are able to charge extra for cheap operations and use these saved costs to pay for more expensive ones. If the charge for each operation is $O(f)$, and there is always sufficient money to pay for the time we actually use, then the algorithm has $O(f)$ amortized cost.¹⁷

Proof We maintain the invariant that every variable u_k, v_l appearing in a $(\overline{u_k}, \overline{v_l})$ entry in the mapping has \$1 on it.

Pay \$1 for a $\langle u_i = 1 \rangle$ or $\langle u_i = \omega \rangle$ constraint, \$2 for a $\langle u_i = u_j \rangle$ constraint, and \$4 for a $\langle u_i \leq u_j \rangle$ constraint.

- Adding a constraint $\langle u_i = 1 \rangle$ or $\langle u_i = \omega \rangle$ costs \$1 plus the cost of adding one $\langle u_k = 1 \rangle$ or $\langle u_k = \omega \rangle$ constraint for each u_k or u_l . This costs \$1 per variable (by induction), but each variable has \$1 on it already, by our invariant. Thus the total cost is \$1.
- Adding a constraint $\langle u_i = u_j \rangle$ costs \$1 plus possibly the cost of adding an $\langle \cdot = 1 \rangle$ or $\langle \cdot = \omega \rangle$ constraint, which is \$1 (by induction). Thus the total cost is (at most) \$2.
- Adding a constraint $\langle u_i \leq u_j \rangle$ costs \$1 plus possibly the cost of adding an $\langle \cdot = 1 \rangle$ or $\langle \cdot = \omega \rangle$ constraint, which is \$1 (by induction). We also add two variables to entries in the mapping, and must place \$1 on each. Thus the total cost is (at most) \$4.

Each \$1 pays for an $O(1)$ operation, and the invariant is maintained. We pay at most \$4 per constraint. Hence the algorithm has an amortized cost of at most $O(1)$ per constraint. \square

¹⁷The image beneath the text here is of a New Zealand \$1 coin, depicting the kiwi *apteryx australis*, our national bird.

This analysis assumes we can implement the *UNION* and *FIND* operations in constant time. In fact there is no known algorithm to do this, but the union-find algorithm with balancing and path compression achieves m operations on n variables in $O(m \alpha(m, n))$ time, where $\alpha(m, n) \leq 4$ for all $n \leq 2^{2^{\dots^2}}$ ¹⁷, essentially constant time per operation [CLR90, pp. 449, 453].

3.6.4 Overall results

We now combine the results of the above sections into three key proofs about the inference as a whole.

Theorem 3.7 (Inference soundness)

For all M in L_0 , if $(CS \circ \blacktriangleright_1)(\emptyset, M) = e : \sigma$ then $\emptyset \vdash_1 e : \sigma$ and $(e)^\natural = M$.

Proof Follows directly from Theorems 3.4 and 3.5, and inspection of the definition of \natural . \square

Theorem 3.8 (Inference pseudo-completeness)

For all M , if $(CS \circ \blacktriangleright_1)(\emptyset, M) = e : \sigma$ then for any e', σ' such that $(e')^\natural = M$ and $\emptyset \vdash_1 e' : \sigma'$, we have $e' \sqsubseteq e$ and $\sigma' \sqsubseteq \sigma$ (where \sqsubseteq is extended pointwise to types, usage annotations, and update flags in terms).

Proof Follows from Theorems 3.4 and 3.5 and the definition of \sqsubseteq . \square

Theorems 3.7 and 3.8 above relate to whole-program usage analysis, with no imported or exported variables. In order to extend them to separate compilation, we would have first to define more precisely what we mean by a module and a signature. We would then restate the above theorems, extending them to modules, nonempty initial environments, and the pessimisation step as detailed informally below:

- “For all well-typed L_0 modules and all maximally-applicable LIX_1 environments, the inference $(\blacktriangleright_1; Pess; CS)$ succeeds, yielding a well-typed LIX_1 module whose erasure is the original module, and whose signature is maximally applicable.”
- “For all well-typed L_0 modules and all maximally-applicable LIX_1 environments, the inference $(\blacktriangleright_1; Pess; CS)$ yields a LIX_1 module that is better (in terms of the goodness ordering) than any other well-typed LIX_1 module whose erasure is the original module.”

We do not give formal details of this extension.

Theorem 3.9 (Inference complexity)

If we assume that nesting of conditionals and abstractions is limited to a constant depth and a linear algorithm exists for union-find, then the complexity of the inference \mathcal{IT}_1 is bounded by $O(n)$, where n is the size of the program.

Proof Because the rules are syntax-directed and types are explicit, phase 1 (\blacktriangleright_1) generates a constraint set of size approximately linear in the size of the program text. The approximation arises from the $(\vdash_1\text{-IF0})$ and $(\vdash_1\text{-ABS})$ rules. In the former, the subtyping of the result type generates constraints proportional to the result type (which is not specified separately in the program text but is bounded by the size of the branches of the conditional). In the latter, constraints are generated for each lambda that are proportional to the number of its free variables. Hence arbitrary nesting of conditionals and/or abstractions can yield arbitrarily large constraint sets. We do not expect this situation to arise in practice and so we rule it out by assumption above, but even if we were to relax this the constraint set would be bounded by the square of the program text size.¹⁸

Pessimisation generates at most one constraint per annotation position in types of signature variables, and this is bounded by the size of the program.

Both are straightforward walks over the abstract syntax tree, taking $O(1)$ time per node.

By Theorem 3.6 the constraint solver (CS) in phase 2 takes time essentially linear (actually $O(m \alpha(m, n))$) for m constraints and n usage variables, where

$\alpha(m, n) \leq 4$ for all $n \leq 2^{2^{\dots^2}}\}^{17}$ in the size of the constraint set. The final substitution takes time linear in the size of the program.

Combining the above yields the desired result. \square

3.7 Discussion

When we implemented the monomorphic analysis of this chapter (extended to a full-featured language) as described in Section 6.2.1, we discovered that it gave extremely poor results in practice. In the entirety of the standard libraries just two thunks were annotated as used-once. More recent trials of the same analysis (Section 6.8.3) confirmed these results. In Section 3.7.1 we explain the reason for this poor behaviour. Section 3.7.2 discusses the lack of principal types and its consequences.

3.7.1 Curried functions and separate compilation

The problem with the monomorphic usage analysis is most clearly evident for curried functions, particularly in conjunction with separate compilation. Consider the following innocuous-looking definition:

$$g = \lambda x . \lambda y . x + y - 1$$

¹⁸Thanks to Jörgen Gustavsson for pointing this case out to us [Gus99, §5.7].

On first glance, it would seem as if g demands its arguments only once, yielding a type like:

$$g : (\text{Int}^1 \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega)^\omega$$

However, this type is not correct. The function g may be partially applied, as in the following example:

```
let h = g z
in h 3 + h 4
```

Here the value of z will in fact be demanded *twice*, once for each call of h . So the type of g above must be wrong. Returning to the definition of g , we see that x is used inside the λy -abstraction, and so by (\vdash_1 -ABS) the topmost annotation of the type of x is at most that of the λy -abstraction – if the abstraction is used more than once, so may x be.

There are two valid types we can assign to g :

$$\begin{aligned} g &: (\text{Int}^\omega \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega)^\omega \\ g &: (\text{Int}^1 \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^1)^\omega \end{aligned}$$

In the first, the first argument is given the (bad) annotation ω , but in exchange the function returned by $g \ z$ has a ω -type. In the second, the first argument has the (good) annotation 1, but the price is that the partial application of g cannot be shared.

These constraints are perfectly reasonable: if $g \ z$ is called many times, then z will be demanded many times. But the LIX_1 analysis is forced to make the choice once and for all, at the definition site of g . If g is *ever* partially applied, then *all* uses of g will have an ω annotation on their first argument. We have already called this bad behaviour *poisoning* (Section 3.3.5), because the single partial application poisons g , which then goes on to poison all its application sites.

The same is true even for the identity function $\text{idInt} = \lambda x : \text{Int} . x$, which the LIX_1 analysis must type either as $(\text{Int}^1 \rightarrow \text{Int}^1)^\omega$ or as $(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega$, not both. Although the problem does not solely lie with curried functions, it is the frequency of curried functions in Haskell code that makes the LIX_1 strategy such a poor one.

Indeed, if a function of two or more curried arguments is exported from the current module, we must assume that it *will* be partially applied, and pessimisation adds constraints to enforce this (Section 3.5.3). In the common case that all the lambdas are adjacent, *i.e.*, all arguments are required before any are used, this means that *all arguments but the last will be free in the final abstraction*, and thus that *all arguments but the last will be forced to ω* .

This is fatal for idiomatic Haskell, which strongly encourages curried function definitions, both in syntax and in evaluator design [PJ92, §3.2.1] (unlike ML which tends to encourage tupled arguments and uncurried functions). It means that the LIX_1 analysis can provide interesting usage annotations only for the last argument of a function; inevitably poisoning ensures that this is insufficient to yield significant (or even measurable) gains for real programs.

This failure was the strong motivation for the development of the polymorphic usage analysis described in the next chapter.¹⁹

¹⁹Another possible way of addressing the problem of curried functions would be to treat adjacent \triangleright

3.7.2 No principal types?

The use of a goodness ordering on types is a little unusual. Conventionally, one has instead a *principal type property* [Jim96, Hin69], which states that every term (in an environment) has a *principal* or *most general* type from which all other types that it can be given may be derived, by subtyping or instantiation. Since the principal type encapsulates all possible types of the term, it is chosen as the best type.

The type system \vdash_1 , however, does not have the principal type property. The (slightly contrived) function

$$f = \lambda x . \lambda y . \text{letrec } d = x + x \text{ in if0 } x \text{ then } y \text{ else } d$$

for example may be typed either as $(\text{Int}^\omega \rightarrow (\text{Int}^1 \rightarrow \text{Int}^1)^\omega)^\omega$ or $(\text{Int}^\omega \rightarrow (\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega)^\omega$. If \vdash_1 had the principal type property, there would be a single type with both of these as instances, and this would be the obvious type to choose for f . In fact, however, neither type is a subtype of the other and their greatest lower bound $(\text{Int}^\omega \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega)^\omega$ is not a valid type for f ; in other words, there is no principal type for f .

This lack of principal types means we must find another means of choosing one best type from a set of possible types. The goodness ordering (Section 3.4) is such a means; in this case it selects the type $(\text{Int}^\omega \rightarrow (\text{Int}^1 \rightarrow \text{Int}^1)^\omega)^\omega$ as the best, enabling updates of the d thunk to be avoided. But type systems with the principal type property enjoy a number of pleasant properties, notably completeness and predictability, which the goodness ordering is unable to provide. Is the loss worth it?

One very practical consequence of the lack of principal types is a lack of predictability. Since the full range of allowable types for a given function is not necessarily representable within the type system, the analysis must heuristically choose one. Inlining or duplicating the definition of a function, moving the definition to another module, or even apparently unrelated changes to the program, may cause the heuristic to choose differently in a subsequent compilation, leading to dramatically different analysis results. This is not the case in a system with principal types.

Jim suggests [Jim96, §6] that some of the benefits of principal typings may be had in their absence by finding a *representation* for all possible typings. In the system of [TWM95a], closely related to ours, Turner *et al.* prove such a property by pairing the type with the global constraint set Θ before solution (corresponding to our C in \blacktriangleright_1). This representation certainly captures all possible typings, as we have shown in Theorem 3.4. However, this trick does not allow us to give a principal LIX_1 type to every L_0 term; selecting an LIX_1 translation requires committing to a particular solution of the constraint set, in general giving it a non-principal type.

Of course, principal types could be preserved by reifying the constraint and storing it within the type, using *constrained polymorphism* (Section 4.8.1). But Section 4.7.3 argues that in fact principal types and usage analysis may be mutually exclusive, explaining that generalisation of a function leads to the loss of usage information within it.

abstractions together, with a single rule for multiple abstraction (as, e.g., Nordlander [Nor98]) [personal communication, Mick Francis, January 2002]. A corresponding multiple application rule could constrain the annotations of the supplied arguments as appropriate for the number of arguments provided. We have not investigated this. However, the problem exemplified by *idInt* would still remain.

In summary, then, there is a tension between having principal types (which gives predictability and avoids poisoning) and not having them (which gives more usage information and simpler types). We discuss this further in Section 4.7.3.

3.8 Separate compilation

In this section we justify the use of pessimisation in Section 3.5.3.

Section 1.5.6 considered the difficulties of performing usage analysis in conjunction with separate compilation. We must perform the analysis without reference to information on the usage of variables exported from the present module (signature variables), and with only the usage types of variables imported from other modules (not their right-hand sides). In order for this to work, we explained the necessity for *maximal applicability*: we must ensure that usage types we provide for exported variables permit *any possible use* by modules that import the present one.

What does this mean? Consider an exported function $f = \text{letrec } c = 21 + 20 \text{ in } \lambda g . g \ c + 1$. In what ways must we allow an importing module to use this function?

- We must allow for each *variable* to be used multiple times, and so its topmost annotation must be ω : e.g., $f \ id + f \ id$ requires $f : ((\text{Int}^\omega \rightarrow \text{Int}^1)^1 \rightarrow \text{Int}^\omega)^\omega$.
- We must allow for each function's *result value* to be used multiple times, so the result annotation must be ω : e.g., $\text{letrec } x = f \ id \text{ in } x + x$ requires $f : ((\text{Int}^\omega \rightarrow \text{Int}^1)^1 \rightarrow \text{Int}^\omega)^\omega$.
- We must allow for the function to be passed a function which uses its argument multiple times, so the annotation of the *argument of the function argument* must also be ω : e.g., $\text{letrec } g = \lambda x . x + x \text{ in } f \ g$ requires $f : ((\text{Int}^\omega \rightarrow \text{Int}^1)^1 \rightarrow \text{Int}^\omega)^\omega$.
- And similarly for all other positive annotation positions.

Negative annotation positions do not matter because they do not affect the use of the function; instead, they specify the use it makes of *other* functions.

More formally, to see how to do this we must examine the meaning of *any possible use*.²⁰ Careful inspection of the type inference rules (Figure 3.4) shows that positive (negative) annotations in the typing environment only ever appear to the left (right) of a primitive inequality constraint $\langle \cdot \leq \cdot \rangle$, and never in an equality constraint.²¹ Thus we guess that allowing *any possible use* would mean *forcing any or all positive*

²⁰Trifonov considers the related notion of “future use” in [TS96, §4.1].

²¹Specifically: Because rule (\blacktriangleright_1 -VAR) takes the type σ of variable x from the type environment and yields it as the expression target type, we must consider not only types in the environment, but also the expression target type. In (\blacktriangleright_1 -IF0), the branch types σ_1 and σ_2 are constrained to be *subtypes* of the least upper bound type σ . Rule (\blacktriangleright_1 -PRIMOP) places no constraint on the usages of the e_i . The abstraction rule (\blacktriangleright_1 -ABS) bounds the topmost annotation of certain variables in the environment to be *less than* v . Application (\blacktriangleright_1 -APP) places no constraint on the topmost usage annotation of the function e_1 , but requires $\sigma'_1 \preceq \sigma_1$, where σ'_1 (appearing on the *left*) is an expression target type (potentially from the typing environment), and σ_1 is a *negative* portion of the type of e_1 , appearing on the *right*. Finally in (\blacktriangleright_1 -LETREC), the body types σ'_i all appear as *subtypes* of the binder types.

annotations to ω and negative annotations to 1. However, further inspection of the inference rules show that no constraint ever forces an annotation to 1 (the constant 1 simply never appears); thus we need consider only the first half, *forcing any or all positive annotations to ω* .

When we import a module, the types of the imported variables are fixed. To avoid violation of the constraints under any possible use, then, we must ensure that when each module is compiled, all positive annotations of exported variables are pre-forced to ω . Adding these constraints will not affect the solvability of the present module, because they could have been induced by an appropriate program body anyway. This is *pessimisation*, as defined formally in Section 3.5.3.

The approach of this section relates interestingly with the usual scheme in which exported variables are given their principal types. These are minimal in the sub-type ordering and permit all sound uses by means of subtyping, and correspond to minimising positive annotations and maximising negative ones. Here we do not have principal types (see Section 3.7.2), and instead force positive annotations to the absolute minimum value to achieve maximal applicability, and allow negative annotations to be maximised according to the goodness ordering (Section 3.4).

3.9 Related work

A number of researchers have addressed similar problems to those considered in this chapter. We have already discussed *Once Upon a Type* [TWM95a, TWM95b], the direct ancestor to the present work, in Section 1.3.5; we discuss the others below.

3.9.1 Gustavsson

The work of Gustavsson is closely related to ours, and so we address it in some detail.

Gustavsson [Gus98] presents a monomorphic, type-based usage analysis for essentially the same source language as ours, but with the addition of update marker check intervals ι as described in Section 2.6. Gustavsson’s type system features the same subtyping relation as LIX_1 (extended to cover the interval annotations), but notationally the primitive “capability ordering” is opposed to ours: Gustavsson writes $\checkmark \leq !$ [Gus98, §4.1], whereas we write $\omega \leq 1$.

Multiple occurrences are checked and annotated by linear environment management in the style of Turner *et al.* [TWM95a] (Section 3.9.7). Values and expressions have separate typing judgements, corresponding roughly with the fact that update markers apply to values and updates apply to (bound) expressions. Occurrence of variables free in a closure (Section 3.3.4) is dealt with by the rule (Value) which injects values into expressions, such that if the value may be duplicated (by an update) then all its free variables must be marked updatable. This allows an elegant treatment of constructors later (see Section 5.3.2), similar to that of [Mar93]. It also allows the separation of the use of a binding from the use of the value to which it is bound, permitting the typing of

$$\text{letrec } x = \bullet 1 + 2 \text{ in letrec } y = \!^1 (\lambda z . z) x \text{ in } y + y$$

which while well-annotated in LX is ill-typed in our system LIX_1 (LIX_1 cannot express that while x 's value is used twice, the binding itself is used only once). Gustavsson and Svenningsson [GS00b] attribute this separation to Faxén [Fax95]. There is a connection here with the extended type system of Appendix C, which has separate notions of demand and use and is able to type this expression successfully.

Although the type soundness results proven by Gustavsson [Gus98, §6] are not of quite the same form as those proven in the present chapter, their import is identical. The inference developed in [Gus99, c.5] is essentially the same as ours, modulo the additional complexity of the constraints relating to the interval annotations. The discussion of principal types in [Gus99, §5.5] refers to the combination of inferred type and constraint set. The best solution is defined by the same ordering as ours, and his constraint language strictly contains ours, so presumably Gustavsson's solution algorithm would be applicable to our constraints also; this algorithm is also believed to be linear [Gus99, §5.7].

Gustavsson reports [Gus98, §8] that his monomorphic usage analysis discovers an average of 58% of all thunks used at most once, although this figure varies wildly from 0% to 100% for different programs. This seems to demonstrate the effectiveness of carrying update marker check intervals in addition to usage annotations in the type system, since these results are better than those we obtained with \mathcal{T}_1 for similar programs. However, these results were for very small programs of between two and thirty lines of Haskell. For the same reasons as ours, his analysis is expected to be far less effective for large programs, and this led him [Gus99, §8.2.4] to design a constrained-polymorphic analysis [GS00b] (Section 4.8.1.6). Furthermore, his results are based on measurements of a prototype interpreter for a simplified language, rather than on a full-featured implementation in a production compiler.

The semantics of [GS00b] removes the interval annotations, thus simplifying the constraint language, reducing it to that of Section 3.5.1 (plus the hiding operator required for constraint abstraction as discussed in Section 4.8.4). [GS01b] gives an algorithm that solves these constraints in time cubic in the number of variables. This is less efficient than the algorithm of Section 3.5.4 in the case where the number of constraints is linear in the number of variables; however Gustavsson and Svenningsson's algorithm applies much more generally, and may be of use when applying the inferences of this chapter and the next to type systems such as that of Appendix C which have richer annotation lattices.

3.9.2 Uniqueness types

The programming language Clean [PvE98] has a *uniqueness type* system, which is closely related to usage typing. Uniqueness types allow stateful objects such as files or mutable arrays to be accessed in a purely functional way by ensuring that each such object has a *reference count* of at most one. Objects having at most a single reference to them may be updated in place without violating purity since such an update is indistinguishable from a copy; this is not the case when more than one reference to the object exists.

In Clean, types of expressions, function arguments, and results are given uniqueness annotations in the same manner as LIX_1 has usage annotations: \bullet denotes

“unique” and \times “non-unique”, with subtyping relation $\bullet \leq \times$. This is opposite to that for usage: an expression of type Int^\bullet having only a single reference may be passed to a function that accepts arguments of type Int^\times having multiple references, but the converse is not possible. Apart from this difference, however, subtyping is defined largely as for our languages, including the accommodation of data types in Chapter 5.²² The uniqueness type system of Clean uses a form of constrained polymorphism to give most-general types to functions such as *foldr* that can be given several different monomorphic uniqueness types.

Uniqueness and usage are not the same. An object that is used at most once certainly has at most one reference to it,²³ and an object with multiple references is almost certainly used more than once, but an object may be used many times through a single reference as well as through multiple references. It is significant that uniqueness enables a function to make an assumption about its *argument* (no other references, so don’t copy), whereas used-at-most-onceness enables a thunk to make an assumption about *itself* (no other uses, so don’t update). This means that information in the two analyses flows in opposite directions, and reverses the sense of the subtyping relation.²⁴

The theory of uniqueness typing is based on term-graph rewriting systems, and a notion of type system appropriate for term graphs. This leads to some infelicities, notably in the treatment of curried and higher-order functions and case statements, as well as to a generally unfamiliar method. On the other hand, Clean is a practical lazy functional programming language with a good production-quality compiler that has been in wide use for many years, with uniqueness analysis incorporated since 1992 [AvGP92]. Thus uniqueness analysis is one of the very few linear-logic-based analyses that has seen successful use in the field.

Uniqueness typing is described first in [BS93b, BS93a], and more recently in [BS96]. However, the most accessible introduction is [BS95b]. Barendsen *et al.* have also presented their system in a more conventional manner by means of an inductive type system over a term language with a natural semantics in [BS95a], and this led to a related strictness analysis [BS98].

3.9.3 Subsumption

The use of subtyping turns out to have been rediscovered several times. It was proposed independently²⁵ by myself [WPJ99], Gustavsson [Gus98], and Faxén [Fax95], but in the context of usage analysis it had already appeared in [GH90] and [LGH⁺92], and in the context of binding-time analysis it was already well known, *e.g.*, [Mos93]. It is also present in the uniqueness type system of Clean, where it has the opposite sense to usage subtyping and is referred to as “coercion” [dMJB⁺99, §4.3.4].

²²There are two other significant differences: if two functions lie in the subtype relation then their topmost annotations must be *identical* rather than merely related; and so-called “consistent substitution” requires the same for type variables. The latter is discussed further in relation to data types, in Section 5.4.4.3.

²³Assuming we actually mean use-once-don’t-drag (Section 1.3.4).

²⁴See also the discussion in [Gus99, §7.4].

²⁵Gustavsson confirms this in [Gus99, p. 218].

3.9.4 Type inference

The standard Hindley–Milner type inference Algorithm \mathcal{W} for inferring the principal type scheme of an expression in the lambda calculus with let was introduced by Milner in [Mil78, DM82], although the algorithm without let had already appeared in [Hin69]. The inference of Section 3.5 is based on this algorithm, but instead of performing unification while passing over the term, it accumulates constraints to be solved after examination of the entire term (see Section 4.8.1.2).

3.9.5 The goodness ordering

Sewell [Sew98, §4] uses a “modified ‘subtype’ order”, similar to our goodness ordering, to define “most local possible” types in a capability inference system for a distributed π -calculus.

Flanagan and Felleisen [FF99, §4.2] discuss in detail the problem of selecting one of multiple solutions to a constraint. Like ours, their subtype ordering is contravariant on function types, and multiple solutions to a constraint may be incomparable under this ordering even though some obviously contain more information than others. To resolve this they introduce an ordering which ranks solutions according to their *accuracy*; like ours, this is *covariant* on function types.

In the system of Flanagan *et al.*, as in our own, internal variables (those not mentioned in either type or context) are still significant if they annotate a term. This means that the notion of observability must be different from that of systems where only external variables are significant, such as [TS96]. This may explain the need for a covariant goodness ordering, and also Pottier’s criticism of it [Pot01, §6]. Such variables must not be simplified away, and should be treated as positive when searching for a least solution.

3.9.6 Constraint solution

Henglein [Hen91] gives a constraint solution algorithm for constraints rather more complex than ours, over a two-point domain. He derives an almost-linear algorithm, and his reduction [Hen91, p. 17] is related to our algorithm, but the bulk of his algorithm deals with the complexities of his constraint terms. Rehof and Mogensen [RM99] discuss implementing similar constraints to ours, over finite (semi-)lattices, using both a fixpoint technique due to Kildall and a reduction to Horn-clause satisfiability due to Dowling and Gallier. Their algorithm is linear.

3.9.7 Occurrences and affine linear logic

Due to the constraints on occurrences of variables, the well-typing rules of Section 3.3 describe an *affine linear* [Jac94] type system: in the terminology of linear logic, we permit arbitrary weakening (bound variables need not be used) but restricted contraction (only bound variables annotated with ω may be used more than once). The observation that affine logics correspond to lazy (call-by-need) languages is due to Maraist *et al.* [MOTW95].

We could equally well have expressed this explicitly, using a system with separated environments as introduced by Girard's Logic of Unity [Gir93]. An example is Barber and Plotkin's *Dual Intuitionistic Linear Logic*, *DILL* [BP97]. *DILL* $D(\mathbb{C})$ judgements are of the form $\Gamma ; \Delta \vdash t : A$, where t is a term, A is a type, Γ is an intuitionistic environment (of variables for which weakening and contraction are permitted) and Δ is a linear environment (of variables for which weakening and contraction are not permitted). Loosely, types can be intuitionistic or linear; a type of the form $!A$ is intuitionistic and a type A is linear.

There are two variable rules, one for intuitionistic and one for linear variables, respectively taking the type from the intuitionistic and linear environments:

$$\frac{}{\Gamma_1, x : A, \Gamma_2 ; - \vdash x : A} (Int-Ax) \quad \frac{}{\Gamma ; x : A \vdash x : A} (Lin-Ax)$$

The rule for lambda abstractions, $(-\circ I)$, introduces linear functions that use their argument linearly:

$$\frac{\Gamma ; \Delta, x : A \vdash t : B}{\Gamma ; \Delta \vdash \lambda x : A . t : A \multimap B} (-\circ I)$$

If we want to use the function multiple times, we must derive a typing using $(! - I)$:

$$\frac{\frac{\Gamma ; x : A \vdash t : B}{\Gamma ; - \vdash \lambda x : A . t : A \multimap B} (-\circ I)}{\Gamma ; - \vdash !\lambda x : A . t : !(A \multimap B)} (!I)$$

and it is clear that to obtain a function usable with type $!(A \multimap B)$, we have had to guarantee that it has no free linear variables – Δ must be empty. This is exactly the free variable condition we discussed above in Section 3.3.4, clause 3 of $(\vdash_1\text{-ABS})!$

Multiple occurrences are counted not syntactically, but by the way the environments are combined in the rules. For example, consider the application rule:

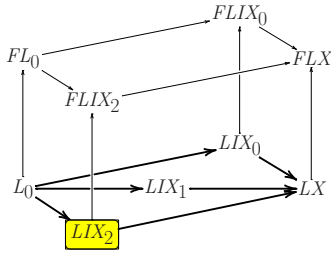
$$\frac{\Gamma ; \Delta_1 \vdash u : A \multimap B \quad \Gamma ; \Delta_2 \vdash t : A}{\Gamma ; \Delta_1, \Delta_2 \vdash u t : B} (-\circ E)$$

Intuitionistic variables are shared between subterms, but linear variables are counted separately and then added together. The implicit condition that no variable is duplicated in Δ_1, Δ_2 ensures that no linear variable may be used more than once.

The connection with our annotated system should be clear: to the left of the turnstile ω -annotated types go into Γ and 1-annotated types into Δ ; to the right of the turnstile $!A$ corresponds to τ^ω and A corresponds to τ^1 .

The present system could have used such careful manipulation of environments to compute usage annotations; either using separated environments as above, or a single environment with special annotations as in [TWM95b, Gus98, GS00b]. Indeed, the increased resolution of the system of Appendix C requires it. But for the present, we prefer the syntactic occurrence function: it is easy to understand, simple to implement, and uses only conventional environment manipulation. These factors become important when integrating the type system into an existing type-directed compiler.

Chapter 4.



Polymorphic Usage Types

In order to address the limitations of monomorphic usage analysis, we introduce a more powerful system, a *simple polymorphic* usage analysis. This novel analysis adds enough power to the monomorphic analysis to be useful, without raising the complexity of the analysis so far as to be impractical.

The analysis is an approximating one, and a key claim of the present thesis is that this approximation is *desirable*: the overhead of manipulating an alternative representation not involving approximation is high, and the approximation forced by simple polymorphism dramatically reduces this overhead whilst still yielding good pragmatic results for the inference. We discuss the support for this claim in Chapter 6; in the present chapter we present the type system and inference algorithm.

The presentation largely follows the structure of Chapter 3. Section 4.1 summarises the problems of the monomorphic analysis and motivates simple polymorphism as a solution. Section 4.2 introduces the polymorphic language, LIX_2 ; the well-typing rules are given in Section 4.3. Section 4.4 presents the basic inference algorithm \mathcal{IT}_2 , but defers the theory and practice of finding the best simple-polymorphic type to Section 4.5. The algorithm is proved correct in Section 4.6. Section 4.7 considers a number of possible improvements to the algorithm, and examines the results of the implementation. Section 4.8 considers related work.

4.1 Introduction

As we saw in Section 3.7, the monomorphic usage analysis of the previous chapter has severe limitations, and turns out to be essentially useless in practice. The sim-

ple polymorphic analysis of the present chapter extends that analysis to one that is useful in practice, increasing the power while still keeping the costs reasonable and thus remaining within the design criteria of Section 1.5.2. We begin by recalling the problem we encountered with the monomorphic usage analysis (Section 4.1.1), consider an obvious but unsatisfactory solution (Section 4.1.2), and then propose an extension that will solve it (Section 4.1.3).

4.1.1 Problems with monomorphic usage analysis

Section 3.7.1 showed that the monomorphic usage analysis is unable to avoid poisoning the first argument of functions like $g = \lambda x . \lambda y . x + y - 1$, a two-argument curried function, if both saturated and partial applications are permitted. Since separate compilation forces the analysis to assume that any exported function may be partially applied, this means that for all exported functions, all arguments but the last are forced to ω , obviously an undesirable situation.

The two possible monomorphic types of g are:

$$\begin{aligned} g_1 &: (\text{Int}^\omega \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^\omega)^\omega \\ g_2 &: (\text{Int}^1 \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^1)^\omega \end{aligned}$$

Only one of these two incomparable types may be chosen, and since only one is maximally applicable (Section 3.8) the choice is usually forced.

The problem is even more starkly demonstrated by the identity function, $\text{idInt} = \lambda x : \text{Int} . x$. Clearly the result of idInt may be used as many times as the argument; but the monomorphic analysis of Chapter 3 must choose either $(\text{Int}^1 \rightarrow \text{Int}^1)^\omega$ or $(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega$, not both. If idInt is once applied to an argument used ω , then the second type is chosen and it will poison all arguments to which it is subsequently applied.

It seems that the problem stems from a lack of principal types – we would like to give the function a single type having both alternatives as instances, thus permitting the best type to be used for each application site independently. Gustavsson [GS00b] refers to this as “taking the context into account”: the type of the function is made sensitive to the context of its application.

4.1.2 An idea: polymorphism

There is an obvious way of extending the monomorphic type system to admit principal types, namely *constrained polymorphism* [Cur90, AW93]. Constrained polymorphism adds the ability to abstract over a set of usage variables and the constraints over them. For example:

$$g_3 :: (\forall u, v : \langle u \leq v \rangle . \text{Int}^u \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^v)^\omega$$

is a constrained-polymorphic type. In general, given an expression e one may use the inference algorithm already presented in Chapter 3 to find its type τ and the constraints C over it; it then trivially follows that if $\overline{u_i}$ is the set of usage variables

free in τ then e can be given type $\forall \overline{u_i} : C . \tau$ and that this is principal (all other types for e are instances of this type).

However, we are looking for a simple and lightweight solution and constrained polymorphism is neither simple nor lightweight (as we discuss in Section 4.8.1). Instead, we propose a novel form of polymorphism, *simple polymorphism*, which uses approximation to give *unconstrained* polymorphic types to generalised expressions.

4.1.3 A solution: Simple polymorphism

Consider again the example. The function g may be given one type having both g_1 and g_2 as instances simply by using a polymorphic type, making the (equal) usages of the argument and the partial application a parameter:

$$g_4 :: (\forall u . \text{Int}^u \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^u)^\omega$$

Now the two valid types are instances of this polymorphic type, and each call site can instantiate g_4 's type appropriately. (The reader may be surprised by the location of the $\forall u$ *inside* the outermost usage annotation; see Section 4.2.1 for a discussion of this point.)

That solves the problem nicely. The remaining challenge is how to come up with this type for g . After all, here is another possible type for g :

$$g_5 :: (\forall u_1, u_2, u_3 . \text{Int}^{u_1} \rightarrow (\text{Int}^{u_2} \rightarrow \text{Int}^{u_3})^{u_1})^\omega$$

Here we have replaced the ω and 1 annotations with usage variables, and then quantified over them. This type is sound in the type system we define in Section 4.3, but it is unnecessarily complicated, because *nothing is gained by the extra polymorphism*. For example, the usage u_2 on the second argument of g_5 does not give any extra information to the caller (1 was as informative as possible), nor does it make g_5 any more applicable. Similarly, the usage u_3 instead of ω carries no benefit.

We can characterise the situation quite precisely, as follows:

- A usage variable in a *positive* position¹ – for example, u_3 in the type of g_5 – may as well be turned into the constant ω .
- A usage variable in a *negative* position – for example, u_2 in the type of g_5 – may as well be turned into the constant 1.
- Only usage variables that appear *both* covariantly and contravariantly in the function's type need be universally quantified.

Thus we use polymorphism purely to represent *dependencies* between usage annotations, *i.e.*, between inputs and outputs. All other uses of polymorphism are redundant, because they are already covered by subsumption (Section 3.3.5). A denotational semantics of types is introduced in Section 4.5.2 that makes this redundancy clear.

¹Positive (covariant) and negative (contravariant) are defined in Section 3.2.1.

Figure 4.1 Some sample typings.

<i>plus</i>	:	$(\forall u_1 . \text{Int}^{u_1} \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^{u_1})^\omega$
<i>id</i>	:	$(\forall u_1 . \forall \alpha . \alpha^{u_1} \rightarrow \alpha^{u_1})^\omega$
<i>and</i>	:	$(\forall u_1, u_2 . \text{Bool}^{u_1} \rightarrow (\text{Bool}^{u_2} \rightarrow \text{Bool}^{u_2})^{u_1})^\omega$
<i>apply</i>	:	$(\forall u_1, u_2, u_3 . \forall \alpha, \beta . (\alpha^{u_1} \rightarrow \beta^{u_2})^{u_3} \rightarrow (\alpha^{u_1} \rightarrow \beta^{u_2})^{u_3})^\omega$
<i>plus3</i>	:	$(\forall u_1 . \text{Int}^{u_1} \rightarrow (\text{Int}^{u_1} \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^{u_1})^{u_1})^\omega$
<i>flip</i>	:	$(\forall u_1, u_2, u_3 . \forall \alpha, \beta, \gamma .$ $(\alpha^{u_1} \rightarrow (\beta^{u_2} \rightarrow \gamma^{u_3})^1)^{u_2} \rightarrow (\beta^{u_2} \rightarrow (\alpha^{u_1} \rightarrow \gamma^{u_3})^{u_2})^{u_2})^\omega$
<i>compose</i>	:	$(\forall u_1, u_2, u_3, u_4 . \forall \alpha, \beta, \gamma .$ $(\beta^{u_1} \rightarrow \gamma^{u_2})^{u_3} \rightarrow ((\alpha^{u_4} \rightarrow \beta^{u_1})^{u_3} \rightarrow (\alpha^{u_4} \rightarrow \gamma^{u_2})^{u_3})^{u_3})^\omega$
<i>map</i>	:	$(\forall u_1, u_2 . \forall \alpha, \beta .$ $(\alpha^{u_1} \rightarrow \beta^{u_2})^\omega \rightarrow ((\text{List } u_1 \alpha)^1 \rightarrow (\text{List } u_2 \beta)^{u_2})^\omega)^\omega$
<i>foldr</i>	:	$(\forall u_1, u_2 . \forall \alpha, \beta .$ $(\alpha^{u_1} \rightarrow (\beta^{u_2} \rightarrow \beta^{u_2})^1)^\omega \rightarrow (\beta^\omega \rightarrow ((\text{List } u_1 \alpha)^1 \rightarrow \beta^{u_2})^\omega)^\omega)^\omega$
<i>build</i>	:	$(\forall u_1 . \forall \alpha .$ $(\forall \beta . (\alpha^\omega \rightarrow (\beta^\omega \rightarrow \beta^\omega)^\omega)^\omega \rightarrow (\beta^\omega \rightarrow \beta^{u_1})^1)^1 \rightarrow (\text{List } \omega \alpha)^{u_1})^\omega$
<i>augment</i>	:	$(\forall u_1, u_2, u_3 . \forall \alpha .$ $(\forall \beta . (\alpha^\omega \rightarrow (\beta^\omega \rightarrow \beta^\omega)^\omega)^\omega \rightarrow (\beta^{u_1} \rightarrow \beta^{u_2})^1)^{u_3}$ $\rightarrow ((\text{List } \omega \alpha)^{u_1} \rightarrow (\text{List } \omega \alpha)^{u_2})^{u_3})^\omega$
<i>zipWith</i>	:	$(\forall u_1, u_2, u_3, u_4 . \forall \alpha, \beta, \gamma .$ $(\alpha^{u_1} \rightarrow (\beta^{u_2} \rightarrow \gamma^{u_3})^1)^\omega$ $\rightarrow ((\text{List } u_1 \alpha)^{u_4} \rightarrow ((\text{List } u_2 \beta)^1 \rightarrow (\text{List } u_3 \gamma)^{u_3})^{u_4})^\omega)^\omega$
<i>mkPair</i>	:	$(\forall u_1 . \forall \alpha, \beta .$ $\alpha^{u_1} \rightarrow (\beta^{u_1} \rightarrow (\text{Pair } u_1 u_1 \alpha \beta)^{u_1})^{u_1})^\omega$
<i>fst</i>	:	$(\forall u_1 . \forall \alpha, \beta .$ $(\text{Pair } u_1 1 \alpha \beta)^1 \rightarrow \alpha^{u_1})^\omega$

One might argue that it would be easier to get rid of subtyping and use polymorphism instead. But in our explicitly-typed intermediate language, at every call site of a polymorphic function the function is applied to all the type and usage arguments necessary to instantiate all its universally quantified variables. So the more variables we quantify over, the larger our intermediate programs will become. Furthermore, by separating the two we are able to distinguish two distinct properties:

- we use *subtyping* to express the fact that a function may accept a range of argument types, and
- we use *polymorphism* to express dependencies between the usage annotations within a single type.

Simple usage polymorphism, therefore, describes a type system that supports both subtyping and polymorphism, but does not permit subtyping constraints on quantified variables. Like all type systems, it is an attempt to strike a balance between practical considerations (such as decidability, complexity, predictability) and expressive power. Simple usage polymorphism strikes a new balance between simplicity and power: simple quantification adds some principality to the monomorphic type system, but not all, while retaining relatively simple (and compact) types.

4.1.4 Examples

We embark on the technical material in Section 4.2, but first we pause to examine some motivating examples typed in our proposed system.

Figure 4.1 gives the usage-polymorphic typings inferred by our system for a number of standard Haskell library functions.² (In order to include real examples we anticipate somewhat by using the type polymorphism and algebraic data types of Chapter 5.) When typing such library functions, we clearly must choose types that permit all possible uses; that is, that make no assumptions about how often the function or its partial applications are called. The types in the figure make use of usage polymorphism; without this, all variable annotations in the figure would take the “not known” value ω , thus yielding dreadful results from the analysis.

- Usage polymorphism is used to describe dependencies between argument and result. The simplest possible example of this is the identity function, *id*, which simply returns its argument untouched. Clearly, if *id* is passed a use-once (use-many) thunk, it returns a use-once (use-many) thunk; this is expressed by its type.
- The short-circuit “and” is defined by

$$and\ a\ b = \text{case } a \text{ of } \{\text{True} \rightarrow b; \text{False} \rightarrow \text{False}\}$$

Its type contains the same partial application dependency as Section 4.1.3’s g_4 (in u_1). However, it also contains a dependency (in u_2) between its second

²Most of these should be self-explanatory. *apply*, written as infix $\$$ in Haskell, is strictly unnecessary, but useful (because of its precedence) for avoiding excess parentheses in expressions. *build* and *augment* are used in “cheap deforestation”, described in [Gil96, §3.4.2].

argument and result: if the first argument is `True` the function behaves like *id*. This dependency does not conflict with the `False` case because the returned value (`False`) shared between all invocations of *and* is given the type Bool^ω , which is a subtype of Bool^{u_2} .

- The three-argument curried addition function *plus3* demonstrates that curried functions of more than two arguments still only have a single usage argument dealing with partial application. A type of the form $(\cdot^{u_1} \rightarrow (\cdot^{u_2} \rightarrow (\cdot^1 \rightarrow \cdot^\omega)^{u_4})^{u_3})^\omega$ would have the constraint $\langle u_1 \leq u_3 \rangle$ as before (the first argument is used at least as many times as the first partial application), along with the unsurprising $\langle u_2 \leq u_4 \rangle$ (the second argument is used at least as many times as the second partial application), but in addition there is a constraint $\langle u_1 \leq u_4 \rangle$ (the first argument is also used as many times as the *second* partial application). These three constraints are resolved by unifying all four usage variables, according to the approximation algorithm described in Section 4.5.4.
- The *flip* function is defined by $\text{flip } f \ x \ y = f \ y \ x$. Notice the 1 in its type, indicating that when it calls *f* it always fully applies it.
- Function composition *compose* is defined conventionally:

$$\text{compose } f \ g \ x = f \ (g \ x)$$

It exhibits a common pattern in which to each $\forall\alpha$ corresponds a $\forall u$, and all occurrences of α are decorated by that u . This pattern does not however justify abstracting over σ -types (i.e., allowing *id* to have type $(\forall\alpha . \alpha \rightarrow \alpha)^\omega$), as we would then be unable to express such useful types as α^1 : see Section 5.2.2.

- Constructor functions like *mkPair* and destructor functions like *fst* can now be given regular types, at least with the (\blacktriangleright -DATA-EQUAL) annotation scheme (see Chapter 5); in a system such as [WPJ99] with monomorphic usage types only, these require typing rules.

4.2 A language with polymorphic usage types

Once again, our first task is to design a new *typed target language* which we shall call LIX_2 . This extends LIX_1 (described in Section 3.2) with usage polymorphism: it possesses usage-generalised types, usage abstraction (generalisation), and usage application (instantiation). The extension is conservative: all well-typed LIX_1 terms are well-typed in LIX_2 also. The language is presented in Figure 4.2. Here and elsewhere, **lowlighted text** is unchanged from a previous presentation (in this case Figure 3.1). We now discuss the term and type languages and the extended operational semantics in detail.

4.2.1 The type language

In order to permit usage generalisation, we extend the type language with usage variables and universal quantification over them.

The monomorphic language LIX_1 was augmented with usage variables for the purposes of inference (Section 3.5); they were not a part of the language proper. In LIX_2 , usage variables are properly included in the type language. We use variables u , v , w , and occasionally x to denote usage variables (the context will always make it clear whether x refers to a usage or a term variable). Usage variables range over the domain $\{1, \omega\}$.

Quantification of usage variables is introduced by a *usage for-all quantifier*, $\forall u . \tau$, denoting the type “ τ for any value of u ”. The scope of the bound usage variable u is exactly τ , as usual. The type may be instantiated with any particular value κ of u by performing an appropriate substitution, yielding $\tau[\kappa/u]$.

The two-tier structure of our usage types means that we must choose whether our quantifier should generalise σ -types or τ -types. Consider the type of the identity function id :

- (i) The quantifier could generalise τ -types, so $\forall u . \tau$ is a τ -type and id would have type $(\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega$; or
- (ii) The quantifier could generalise σ -types, so $\forall u . \sigma$ is a σ -type and id would have type $\forall u . (\text{Int}^u \rightarrow \text{Int}^u)^\omega$.

The correct choice is (i), as becomes apparent when we consider formulating well-typing rules for the system. If we quantified σ -types, then we might be presented with a type such as $\forall u . \tau^u$ on a variable used multiple times in its scope. Is this well-typed? We want to require that the topmost annotation is ω , but $|\forall u . \tau^u|$ is not even well-defined! If instead we quantified τ -types, the topmost annotation would not lie within the scope of the quantifier and this situation could not arise. Operationally, we may observe that the topmost annotation of the type of an expression should give the usage of that expression. Usage quantification is erased at runtime (like all other type information, Section 2.3.2), and so the actual usage of an expression of generalised type is necessarily identical to that of any instance. This identity is readily modelled at the type level by permitting quantifiers only beneath the topmost annotation of a type.

Furthermore, it is not clear that the types expressible in system (ii) would be useful: $\forall u . \tau^u$ can be expressed by τ^ω (τ^1) in a positive (negative) position if u is not free in τ ; otherwise τ is a function type, and dependencies between the usage of a function and the usages of its argument or result do not arise from the type rules. This same choice is made by Gustavsson in [GS00b].

For notational convenience, we write $\forall \vec{u}_i . \tau$ to abbreviate $\forall u_1 . \forall u_2 . \dots \forall u_n . \tau$. As usual we consider types up to α -conversion of bound usage variable names.

We make use below of the notion of *polarity* of an annotation position or usage variable occurrence, as introduced in Section 3.2.1 (see also Figure 5.8). For target types, we define a *positive occurrence* of a usage variable u in a type τ or σ to be one annotating a covariant position, and a *negative occurrence* to be one annotating a contravariant position. In the case of a σ -type, the topmost annotation is considered to be covariant. We define functions $fu v^+(\sigma)$ ($fu v^-(\sigma)$) as the set of usage variables occurring positively (negatively) in σ ; similarly for τ -types. Thus if $\sigma = ((\alpha^{u_1} \rightarrow \beta^{u_2})^{u_3} \rightarrow (\gamma^{u_4} \rightarrow \delta^{u_5})^{u_6})^{u_7}$, we have $fu v^+(\sigma) = \{u_1, u_5, u_6, u_7\}$ and

Figure 4.2 The polymorphically usage-typed language LIX_2 (cf. Figure 3.1).

Terms	$e ::= a$ $ n$ $ \lambda^{\kappa, \chi} x : \sigma . e$ $ e a$ $ \Lambda u . e$ $ e \kappa$ $ e_1 + e_2$ $ \text{add}_n e$ $ \text{if0 } e \text{ then } e_1 \text{ else } e_2$ $ \text{letrec } x_i : \sigma_i =_{\chi_i} e_i \text{ in } e$	atom literal (integer) term abstraction term application usage abstraction usage application primop (addition) partially-saturated primop zero-test conditional recursive let binding
Atoms	$a ::= x$ $ a \kappa$	term variable atom usage application
τ -types	$\tau ::= \sigma_1 \rightarrow \sigma_2$ $ \text{Int}$ $ \forall u . \tau$	function type primitive type (integer) usage-generalised type
σ -types	$\sigma ::= \tau^{\kappa}$	usage-annotated type
Usage annotations	$\kappa ::= 1$ $ \omega$ $ u, v$	used at most once possibly used many times usage variable
Update flags	$\chi ::= \bullet$ $!$	not updatable/copyable updatable/copyable

Shallow evaluation contexts R , values v , configurations C , heaps H , and stacks S are defined in the same manner as for LIX_0 and LIX_1 .

$fuw^-(\sigma) = \{u_2, u_3, u_4\}$. The notation $fuw(\sigma)$ denotes the set of usage variables occurring positively or negatively in σ ; i.e., $fuw^+(\sigma) \cup fuw^-(\sigma)$. We use ε to range over $\{+, -\}$, and write $\bar{\varepsilon}$ for sign negation, defined by $\bar{-} \triangleq +$ and $\bar{+} \triangleq -$.

4.2.2 The term language

We accommodate the extension of the type language by adding explicit term forms for usage abstraction (generalisation) and application (instantiation), in the style of Girard and Reynolds [Gir72, Rey74]. As Peyton Jones describes in [PJS98a, §3.2] with respect to type polymorphism, this explicit usage polymorphism deals smoothly with scoping issues for term annotations and integrates well with a type-preserving optimising compiler.³

We write usage abstraction as $\Lambda u . e$, and usage application as $e \kappa$. The former generalises e by abstracting it with respect to usage variable u ; the latter instantiates the usage argument of e to κ . As with $\forall \bar{u}_i . \tau$, we write $\Lambda \bar{u}_i . e$ to abbreviate $\Lambda u_1 . \Lambda u_2 . \dots \Lambda u_n . e$, and similarly for $e \bar{\kappa}_i$. As usual we consider terms up to α -conversion of bound usage variable names. Atoms are extended to include variables applied to a vector of usages for technical reasons described in Section 4.2.3 below.

Although usage annotations are extended to include usage variables, update flags remain the same. This is because update flags must be available statically in order to control code generation for thunks. The well-typing rules and inference must take this into account when computing the update flags (Section 4.3.1).

4.2.3 The operational semantics: type erasure without erasing types

At runtime, all type information is erased (Section 2.3.2) since the soundness proofs assure us that a well-typed program cannot have a runtime type error. This type information includes usage-generalised types, usage abstractions, and usage applications, so in one sense these have no operational semantics at all. But while this is true of the target language LX , we work mostly with the instrumented target language LIX_2 , which preserves the types in order to permit easier soundness proofs. This places us in a novel position: we wish to write a type-erasure operational semantics *without erasing the types*.

In the monomorphic setting, this was relatively straightforward, and we have successfully done this for LIX_0 (Section 2.3) and LIX_1 (Section 3.2). In the presence of polymorphism, however, it becomes a little more difficult.

Since usage applications are invisible to the LX operational semantics we must permit them everywhere, even within the argument of a function application; thus atoms become variables *applied to zero or more usage arguments*. These applications are “administrative” – they are part of the instrumentation required to keep types in order, and do not correspond to any operational behaviour of the uninstrumented semantics or implementation.

Since usage abstractions are also invisible to the LX operational semantics, they

³In fact, the compiler we target (GHC, Section 1.2.5) already uses this technique for type polymorphism, and our implementation is greatly simplified by simply reusing the same machinery for usage polymorphism (Section 6.2.2).

Figure 4.3 The operational semantics of usage polymorphism, $LIXC_2$ (extends Figure 2.4).

$$\begin{aligned}
R &::= \dots \mid [\cdot] \kappa \\
v &::= \dots \mid \Lambda u . v \\
a &::= \dots \mid a \kappa
\end{aligned}$$

$$\begin{aligned}
\langle H; \Lambda u . e; S \rangle &\rightarrow_{\overline{u_m}} \langle H'; \Lambda u' . e'; S' \rangle && (\rightarrow\text{-ULAM}) \\
&\text{if } \langle H; e[u'/u]; S \rangle \rightarrow_{\overline{u_m}, u'} \langle H'; e'; S' \rangle \\
&u' \text{ fresh} \\
\\
\langle H; \text{letrec } \overline{x_i} : \sigma_i =^{\kappa_i} e_i \text{ in } e; S \rangle &\rightarrow_{\overline{u_m}} \langle H, y_i : \forall \overline{u_m} . \sigma_i =^{\kappa_i} \Lambda \overline{u_m} . e_i[\phi]; e[\phi]; S \rangle && (\rightarrow\text{-LETREC}) \\
&\text{where } \overline{y_i} \not\in \text{dom}(H) \cup \text{dom}(S) \\
&\phi = [y_i \overline{u_m} / \overline{x_i}] \\
\\
(\Lambda u . e) \kappa &\rightarrow_{\delta} e[\kappa/u] && (\rightarrow_{\delta}\text{-UAPP})
\end{aligned}$$

All other rules unchanged from Figure 2.4.

cannot stop evaluation. That is, *we must permit evaluation underneath usage abstractions!* If the usage lambda is deleted at runtime, $\Lambda u . e$ cannot be a value in general – it looks just like e . This is a rather surprising consequence of our programme, and it necessitates some technical trickery in the formulation of the operational semantics.

The modifications to the semantics are summarised in Figure 4.3. We achieve the desired behaviour by giving a reduction rule ($\rightarrow\text{-ULAM}$) for $\Lambda u . e$, if e is reducible. If v is a value, then $\Lambda u . v$ is also a value. The unfortunate overlap between the term $\Lambda u . e$ and the value $\Lambda u . v$ does not cause any difficulties, since if e is reducible it is certainly not a value.

When reducing under a usage lambda using ($\rightarrow\text{-ULAM}$), we must α -convert the term to ensure the bound usage variable is distinct from every usage variable occurring in H , S , and $\overline{u_m}$, lest an occurrence appear in e' and be inadvertently captured.⁴ (This is merely a proof device, and obviously does not occur at runtime.)

Furthermore, the semantics maintains the set $\overline{u_m}$ of in-scope usage variables. This set is used to ensure that all heap bindings are closed with respect to usage variables. If a binding in the heap were to have a free usage variable, a subsequent reference to that variable from underneath a usage lambda might capture that variable, leading

⁴Alternatively, since we work up to α -equivalence, we could simply choose u to be distinct from these bound names and avoid the need to substitute.

ultimately to a type error. To avoid this, in the (\rightarrow -LETREC) rule we close over all in-scope usage variables before adding a binding to the heap. This is similar to the techniques used in the Glasgow Haskell Compiler to float let-bindings in the presence of type lambdas [PJPS96, §4], and by Morrisett *et al.* to perform polymorphic closure conversion while translating System F programs into TAL [MWCG99, §5.2]. In fact, the latter paper also observes that in a type-erasure interpretation $a \tau$ (in our notation) should be considered a value; however, their translation does not appear to perform evaluation underneath a type abstraction.

The primitive reduction rule (\rightarrow_{δ} -UAPP) for usage application is unsurprising.

Exactly the same technique is used to give an operational semantics for type polymorphism in the full language (Section 5.1.4), where the reduction relation is parameterised over two sets $\overline{\alpha_l}, \overline{u_m}$, and type abstraction and application are treated in the same way as usage abstraction and application.

4.3 Polymorphic usage well-typing rules

The next task is to define a set of *well-typing rules* that specify which LIX_2 programs are valid; *i.e.*, in particular, which programs carry valid usage types. The rules extend those of LIX_1 (described in Section 3.3); we simply add new rules to deal with the new term forms, usage abstraction and usage application. Update flags and the subtype relation are also computed slightly differently due to the addition of usage variables.

The well-typing rules for LIX_2 are presented in Figure 4.4. The judgement $\Gamma \vdash_2 e : \sigma$ may be read as stating that “In type environment Γ , the LIX_2 term e can be given type σ .” The discussions of Section 3.3 on basic uses, syntactic occurrence, occurrences in a closure, subsumption, and demands and recursive binding groups remain unchanged for the extended language, and we discuss only the new rules and issues below.

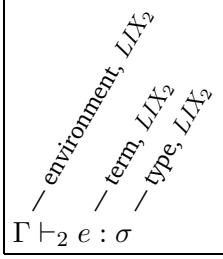
4.3.1 Update flags

Update flags are computed in (\vdash_2 -LETREC) and (\vdash_2 -ABS) from the topmost usage annotation of the corresponding type. Although for LIX_2 usage annotations have been extended to include variables, update flags must still be either \bullet (not updatable or copyable) or $!$ (updatable or copyable). A usage variable is treated the same as ω because ω expresses complete lack of information (it is the bottom element of the annotation lattice). If the topmost annotation is a universally-quantified variable, then statically we do not know whether the expression will be used at most once or many times; thus we must mark it updatable or permit it to be copied.⁵

The conversion is performed by the function \cdot^\dagger , which now has the following

⁵Conceivably we could check to see if the variable is always instantiated to the same value (*e.g.*, due to an enclosing application to a constant), and use that value; in practice however we expect this optimisation to be of little value (*e.g.*, such an application should be β -reduced by the optimiser) and do not implement it.

Figure 4.4 Well-typing rules for LIX_2 (cf. Figure 3.2).



$$\frac{}{\Gamma, x : \sigma \vdash_2 x : \sigma} (\vdash_2\text{-VAR}) \quad \frac{}{\Gamma \vdash_2 n : \text{Int}^\omega} (\vdash_2\text{-LIT})$$

$$\frac{\Gamma \vdash_2 e : \text{Int}^1 \quad \Gamma \vdash_2 e_i : \sigma \quad i = 1, 2}{\Gamma \vdash_2 \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma} (\vdash_2\text{-IF0})$$

$$\frac{\Gamma \vdash_2 e_i : \text{Int}^1 \quad i = 1, 2}{\Gamma \vdash_2 e_1 + e_2 : \text{Int}^\omega} (\vdash_2\text{-PRIMOP}) \quad \frac{\Gamma \vdash_2 e : \text{Int}^1}{\Gamma \vdash_2 \text{add}_n e : \text{Int}^\omega} (\vdash_2\text{-PRIMOP-R})$$

$$\frac{\begin{array}{l} \Gamma, x : \sigma_1 \vdash_2 e : \sigma_2 \\ \text{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \text{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \quad \text{for all } y \in \Gamma \end{array}}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-ABS})$$

$$\frac{\Gamma \vdash_2 e : (\sigma_1 \rightarrow \sigma_2)^1 \quad \Gamma \vdash_2 a : \sigma_1}{\Gamma \vdash_2 e a : \sigma_2} (\vdash_2\text{-APP})$$

$$\frac{\begin{array}{l} \Gamma, \overline{x_j : \sigma_j} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j} \vdash_2 e : \sigma \\ \left(\text{occur}(x_i, e) + \sum_{j=1}^n \text{occur}(x_i, e_j) \right) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \vdash_2 \text{letrec } x_i : \sigma_i = |\sigma_i|^\dagger e_i \text{ in } e : \sigma} (\vdash_2\text{-LETREC})$$

$$\frac{\Gamma \vdash_2 e : \sigma' \quad \sigma' \preceq \sigma}{\Gamma \vdash_2 e : \sigma} (\vdash_2\text{-SUB})$$

$$\frac{\Gamma, u \vdash_2 e : \tau^\kappa \quad u \notin (fuv(\Gamma) \cup fuv(\kappa))}{\Gamma \vdash_2 \Lambda u . e : (\forall u . \tau)^\kappa} (\vdash_2\text{-UABS})$$

$$\frac{\Gamma \vdash_2 e : (\forall u . \tau)^\kappa}{\Gamma \vdash_2 e \kappa' : (\tau[\kappa'/u])^\kappa} (\vdash_2\text{-UAPP})$$

extended definition (cf. Section 3.3.1):

$$\begin{array}{lll} 1^\dagger & \triangleq & \bullet \\ \omega^\dagger & \triangleq & ! \\ u^\dagger & \triangleq & ! \end{array}$$

Notice that the usage annotation and update flag are no longer in a one-to-one relationship, justifying the decision of Section 3.2.2 to separate them.

Section 4.7.3 discusses the effect of generalisation on the usage information computed by the analysis: clearly the fact that $u^\dagger = !$ suggests that we should not generalise too often.

4.3.2 Usage abstraction and application

We have two new term forms for which to provide type rules: usage abstraction and usage application.

The usage abstraction rule ($\vdash_2\text{-UABS}$) abstracts a usage variable from an expression $e : \tau^\kappa$, yielding a generalised type.

Considering how this is done for the comparable case of type abstraction in ML [MTHM97, §4.8, §4.10(15)], we see that this is only valid if the abstracted usage variable does not occur free in the type environment. Without such a restriction, the relationship between the type and the environment would be broken by the new binder: consider that from

$$x : \text{Int}^u \vdash_2 \lambda^{\omega, !} g : (\text{Int}^u \rightarrow \text{Int}^\omega)^\omega . g \ x : ((\text{Int}^u \rightarrow \text{Int}^\omega)^\omega \rightarrow \text{Int}^\omega)^\omega$$

we would be able to derive

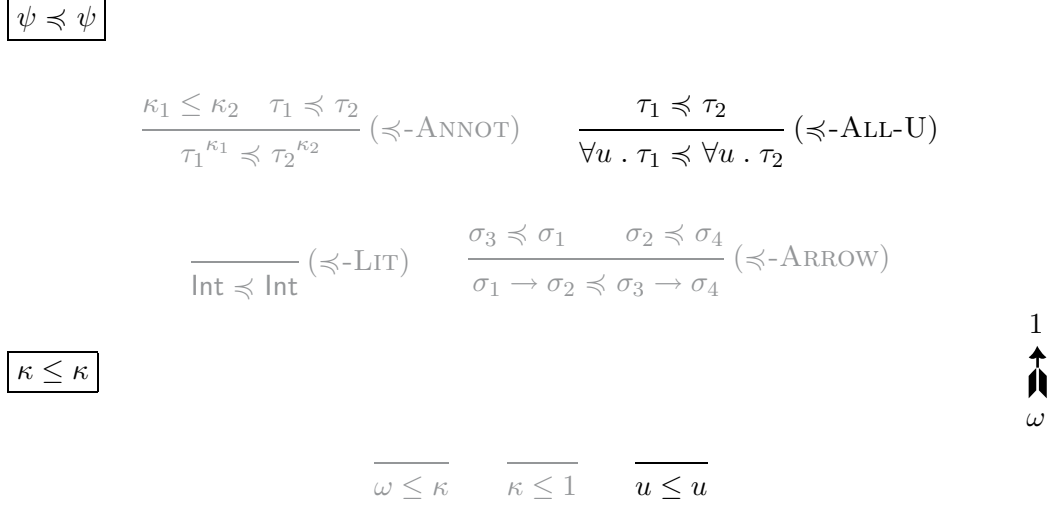
$$x : \text{Int}^u \not\vdash_2 (\Lambda u . \lambda^{\omega, !} g : (\text{Int}^u \rightarrow \text{Int}^\omega)^\omega . g \ x) \ \omega \ (\lambda y . y + y) : \text{Int}^\omega$$

which potentially violates soundness by using x twice without forcing its topmost annotation to ω .

There is a further restriction unique to the present situation. Observe that the expression to be generalised has a σ -type, and yet generalisation is permitted only at the τ -level. Section 4.2.1 explains that the topmost annotation remains the same despite generalisation or instantiation, and so we simply lift it over the quantifier, generalising $e : \tau^\kappa$ to $\Lambda u . e : (\forall u . \tau)^\kappa$. This leads to the crucial further restriction on generalisation, namely that we may not generalise the topmost annotation itself, since this would violate the scoping rules. That is, we may *not* generalise $e : \tau^u$ to $\Lambda u . e : (\forall u . \tau)^u$.

These restrictions are implemented in the side condition of ($\vdash_2\text{-UABS}$), which requires that the abstracted variable not occur in either $fuv(\Gamma)$ or $fuv(\kappa)$.

The usage application rule ($\vdash_2\text{-UAPP}$) applies an expression $e : (\forall u . \tau)^\kappa$ (of usage-generalised type) to a usage annotation κ' . This is completely straightforward: the actual usage argument is substituted for the formal one, and the topmost usage annotation is dropped back down into place (cf. [MTHM97, §4.10(2)]).

Figure 4.5 The subtype (\preccurlyeq) and primitive (\leq) orderings over LIX_2 (cf. Figure 3.3).

4.3.3 Subtyping

Our language now permits usage variables u as annotations, in addition to the constants 1 and ω . We accommodate this in our definition of the primitive ordering \leq (Figure 4.5), stating that for all variables u , the relations $\omega \leq u$, $u \leq 1$, and $u \leq u$ all hold; distinct usage variables are incomparable. Similarly, in the definition of the subtype ordering \preccurlyeq , usage-quantified types are comparable only if the quantified variable is the same in both cases; this of course can be achieved by α -conversion.

This definition of subtyping is particularly simple to state and to work with. The main reason for this is that our subtyping relation is purely *structural*; for example, we have

$$(\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega \preccurlyeq (\forall u . \text{Int}^\omega \rightarrow \text{Int}^1)^\omega$$

but we have

$$(\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega \not\preccurlyeq (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega$$

because the two types being compared are of different shapes (we consider a definition of subtyping that *would* allow this in Section 4.5.2).

The (\preccurlyeq -ALL-U) rule corresponds precisely to that of Systems Fun , F_{\leq} , $F_{<}$, and F_{\leq}^ω [CW85, CG92, CMMS94, SP94] in the case where the bound is omitted; indeed, it is hard to imagine a different rule.

4.4 Polymorphic usage inference

Once again, the final part of our type-based analysis is the *inference algorithm* \mathcal{IT}_2 , which must compute the best valid LIX_2 typing for any L_0 program. That is, when presented with an L_0 program, the algorithm must infer an equivalent LIX_2 program

which is well-typed according to the rules of Section 4.3, and if there is more than one such, it should choose the one that is the ‘best’ in some appropriate sense.

The algorithm works in much the same way as the \mathcal{IT}_1 inference of Section 3.5. Phase 1 of the inference, \blacktriangleright_2 (Section 4.4.1), passes over the program annotating it with fresh usage variables and generating constraints over them. Unlike \mathcal{IT}_1 , this phase must also introduce appropriate usage abstractions as explained in Section 4.4.2. This is followed by a pessimisation pass, P_{ess} (Section 4.4.3), which allows for usage of exported functions by other modules. Finally phase 2, \mathcal{CS} (Section 4.4.4), finds the best solution to the constraints and applies it to the program, yielding the best valid LIX_2 typing of the source program. The combination of these three parts yields the complete inference:

$$\mathcal{IT}_2(\Gamma, M) \triangleq (\mathcal{CS} \circ P_{ess} \circ \blacktriangleright_2)(\Gamma, M) = (e, \sigma)$$

where $e : \sigma$ is the best well-typed LIX_2 term corresponding to M .

4.4.1 Inference phase 1

The first phase of the inference algorithm, \blacktriangleright_2 , takes an L_0 term and yields an equivalent LIX_2 term, along with an appropriate constraint on its free variables. This phase is defined in Figures 4.6 and 4.7. The figures define a relation

$$\Gamma \blacktriangleright_2 M \rightsquigarrow e : \sigma; C; V$$

which may be read “In the LIX_2 type environment Γ , the L_0 term M translates to LIX_2 term e , which has type σ , generated constraints C , and free term variables V .”

In Section 3.5.2 we were able to obtain an inference algorithm for LIX_1 by restricting uses of $(\vdash_1\text{-SUB})$ to certain canonical locations in the derivation tree. Similarly, here we obtain an inference algorithm for LIX_2 by restricting also the locations of the additional non-syntax-directed rules $(\vdash_2\text{-UABS})$ and $(\vdash_2\text{-UAPP})$ as described in Section 4.4.2. Thus the \blacktriangleright_2 algorithm remains syntax-directed.

The constraints used and the basic inference rules are identical to those already discussed (in Sections 3.5.1 and 3.5.2 respectively), and so we examine them no further. Update flags are computed using \cdot^\dagger in the same manner as before, but with the extended definition found in Section 4.3.1.

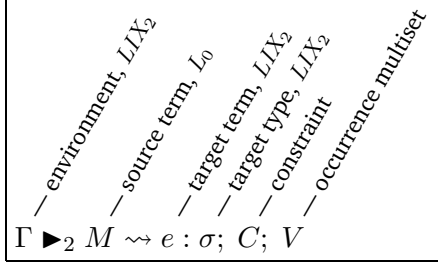
The $(\blacktriangleright_2\text{-VAR})$ rule is extended to deal with usage-generalised term variables in the type environment; these variables are fully instantiated with fresh usage variables in the usual way [MTHM97, §4.10(2)]. The side condition “ τ a usage-monotype”, *i.e.*, not of the form $\forall u . \tau'$, simply ensures the fullest possible instantiation; it preserves the invariant that the expression type σ synthesized by \blacktriangleright_2 is a monotype (*i.e.*, in ML terminology, a type rather than a type scheme).

The most complicated inference rule, $(\blacktriangleright_2\text{-LETREC})$, deals with recursive binding groups, and is discussed separately below.

4.4.2 Generalisation

Up to this point, LIX_2 looks very much like LIX_1 . But phase 1 of the inference is significantly more complicated for the simple-polymorphic language. Constraints are

Figure 4.6 Basic type inference rules from L_0 to LIX_2 , omitting (\blacktriangleright_2 -LETREC) (cf. Figure 3.4).



$$\frac{\text{fresh } \overline{v_i} \quad \tau \text{ a usage-monotype}}{\Gamma, x : (\forall \overline{u_i}. \tau)^\kappa \blacktriangleright_2 x \rightsquigarrow x \overline{v_i} : (\tau[\overline{v_i}/\overline{u_i}])^\kappa; \emptyset; \wr x} (\blacktriangleright_2\text{-VAR})$$

$$\frac{}{\Gamma \blacktriangleright_2 n \rightsquigarrow n : \text{Int}^\omega; \emptyset; \wr} (\blacktriangleright_2\text{-LIT})$$

$$\frac{\begin{array}{l} \Gamma \blacktriangleright_2 M \rightsquigarrow e : \text{Int}^\kappa; C; V \\ \Gamma \blacktriangleright_2 M_i \rightsquigarrow e_i : \sigma_i; C_i; V_i \quad i = 1, 2 \\ (C_3, \sigma) = \text{FreshLUB}(\sigma_1, \sigma_2) \end{array}}{\Gamma \blacktriangleright_2 \text{if0 } M \text{ then } M_1 \text{ else } M_2 \rightsquigarrow \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma; C \wedge C_1 \wedge C_2 \wedge C_3; V \uplus (V_1 \sqcup V_2)} (\blacktriangleright_2\text{-IF0})$$

$$\frac{\Gamma \blacktriangleright_2 M_i \rightsquigarrow e_i : \text{Int}^{\kappa_i}; C_i; V_i \quad i = 1, 2}{\Gamma \blacktriangleright_2 M_1 + M_2 \rightsquigarrow e_1 + e_2 : \text{Int}^\omega; C_1 \wedge C_2; V_1 \uplus V_2} (\blacktriangleright_2\text{-PRIMOP})$$

$$\frac{\begin{array}{l} \sigma_1 = [t_1]_\sigma^{\text{fresh}} \quad \text{fresh } v \\ \Gamma, x : \sigma_1 \blacktriangleright_1 M \rightsquigarrow e : \sigma_2; C_1; V \\ C_2 = \{V(x) > 1 \Rightarrow \langle |\sigma_1| = \omega \rangle\} \\ C_3 = \bigwedge_{y \in \Gamma} \{V(y) > 0 \Rightarrow \langle |\Gamma(y)| \leq v \rangle\} \end{array}}{\Gamma \blacktriangleright_1 \lambda x : t_1. M \rightsquigarrow \lambda^{v, v^\dagger} x : \sigma_1. e : (\sigma_1 \rightarrow \sigma_2)^v; C_1 \wedge C_2 \wedge C_3; V \setminus \{x\}} (\blacktriangleright_2\text{-ABS})$$

$$\frac{\begin{array}{l} \Gamma \blacktriangleright_2 M \rightsquigarrow e : (\sigma_1 \rightarrow \sigma_2)^\kappa; C_1; V_1 \\ \Gamma \blacktriangleright_2 A \rightsquigarrow a : \sigma'_1; C_2; V_2 \\ C_3 = \{\sigma'_1 \preceq \sigma_1\} \end{array}}{\Gamma \blacktriangleright_2 M A \rightsquigarrow e a : \sigma_2; C_1 \wedge C_2 \wedge C_3; V_1 \uplus V_2} (\blacktriangleright_2\text{-APP})$$

no longer merely accumulated for later solution: whereas the monomorphic inference builds the constraint in phase 1 and then solves it in phase 2, the polymorphic inference must *interleave* the building of the constraint with its partial solution or approximation. This is because of generalisation. In the monomorphic language, typings differ only in the value of their usage annotations and we are able to infer a constraint C for a given source program M that encodes all possible target typings of that program, deferring the choice of a best solution to phase 2. In the simple-polymorphic language, typings may also differ in structure: the generalisation rule (\vdash_2 -UABS) introduces a quantifier, but is not source-syntax-directed. Thus we must choose *while inferring the constraint* C whether to usage-generalise a term or not, and this decision *fixes the structure* and necessarily loses information. Furthermore, precisely because our polymorphism is simple (and thus we cannot quantify over constraints), we must approximate the constraint at every generalisation step, rather than only approximating the whole, just once, in phase 2.

Determining where to perform these generalisation steps is another issue that must be considered, since the well-typing rules of Figure 4.4 give us no guidance. The same issue arises for type generalisation in languages such as ML with implicit polymorphism, and the standard choice is that of Milner [Mil78, pp. 354–355]: polymorphism is introduced at let(rec) bindings. This is done by using a *closure operator* to compute the most general types for the binders in each binding group, from the inferred types of their right-hand sides [DM82, p. 210] [MTHM97, §4.8]. Thus, usage abstractions are permitted only at letrec bindings, and usage applications are generated at each variable occurrence to freshly instantiate its type.

Usage generalisation at letrec leads to the use of predicative or “rank-1” [Lei83] polymorphism only: as in ML, letrec-bound variables are given type *schemes* where all quantifiers appear at the front, while expressions and lambda-bound variables are always given monomorphic types. While our inference system generates and uses only such predicative polymorphism, a simple modification to the inference algorithm would allow impredicative types to occur in the initial environment. It is not obvious how one might *infer* useful impredicative types, although we do discuss this issue briefly in Section 4.7.4.

The inference rule for letrec, then, is presented in Figure 4.7. This rule translates a mutually-recursive group of bindings,⁶ and generalises their types according to a suitably-modified variant of the Damas–Milner generalisation rule.

Translation of a source term letrec $x_i : t_i = M_i$ in M begins by adding fresh annotations to convert the source binder types t_i to target types $\overline{\tau_i^{v_i}}$. The right-hand sides of the bindings are now translated, in an environment extended with the newly-computed target types of the bindings being translated (thus permitting recursion). We collect the resulting constraints, and add further constraints requiring the types of the right-hand sides to be subtypes of those of their binders.

The crucial step now follows: we invoke the *closure operation*

$$Clos(C_1, \Gamma, \overline{\tau_i^{v_i}}) = (C'_1, \overline{u_k}, S)$$

⁶We assume that letrecs have already been broken into strongly-connected components by an earlier analysis (not described in this thesis).

Figure 4.7 Type inference rule (\blacktriangleright_2 -LETREC) (cf. Figure 3.4).

$$\boxed{\Gamma \blacktriangleright_2 M \rightsquigarrow e : \sigma; C; V}$$

$$\begin{array}{c}
\tau_i^{v_i} = \lceil t_i \rceil_{\sigma}^{fresh} \text{ for all } i \\
\Gamma, \overline{x_j : \tau_j^{v_j}} \blacktriangleright_2 M_i \rightsquigarrow e_i : \sigma'_i; C_1^i; V_i \text{ for all } i \\
C_1 = \bigwedge_i (C_1^i \wedge \{\sigma'_i \preceq \tau_i^{v_i}\}) \\
(C'_1, \overline{u_k}, S) = Clos(C_1, \Gamma, \overline{\tau_i^{v_i}}) \\
\Gamma, \overline{x_j : (\forall \overline{u_k} . S \tau_j)^{v_j}} \blacktriangleright_2 M \rightsquigarrow e : \sigma; C_2; V \\
C_3 = \bigwedge_i \{ (V(x_i) + \sum_j V_j(x_i)) > 1 \Rightarrow v_i = \omega \} \\
\hline
\Gamma \blacktriangleright_2 \text{ letrec } \overline{x_i : t_i = M_i} \text{ in } M \\
\rightsquigarrow \text{ letrec } x_i : (\forall \overline{u_k} . S \tau_i)^{v_i = v_i^\dagger} \Lambda \overline{u_k} . S e_i[\overline{(x_j \overline{u_k}) / \overline{x_j}}] \text{ in } e : \sigma; \\
C'_1 \wedge C_2 \wedge C_3; (\biguplus_i V_i \uplus V) \setminus \{\overline{x_i}\} \\
\hline
\end{array}$$

in order to determine the vector of usage variables over which the binders are to be generalised, and to perform any approximation necessary. *Clos* is passed three things: the vector $\overline{\tau_i^{v_i}}$ of (monomorphic) binder types to generalise, the type environment Γ within which to generalise, and the constraint C_1 over these types. Implicitly there are three sorts of usage variable in the domain of the constraint C_1 :

- *candidate* variables, from the types of the binders, which might or might not be generalised,
- *forbidden* variables, from the types in the environment and the topmost annotations $\overline{v_i}$ of the binder types, which cannot be generalised because they are outside the scope of the usage binders to be introduced, and
- *internal* variables, which participate in constraints and may annotate the right-hand side terms $\overline{e_i}$ within the scope of the usage binders but do not occur in $\overline{\tau_i^{v_i}}$ or Γ , and might or might not be generalised.

The closure operation returns a vector $\overline{u_k}$ of usage variables to generalise, along with a substitution S which has the effect of unifying each of these variables with a cluster of candidate and/or internal variables. It also returns a constraint C'_1 , the residual constraint after unification and necessary approximation have been performed.

The intent is that the closure algorithm returns the largest possible set of variables $\overline{u_k}$ over which the bindings may be abstracted without violating scoping rules or soundness. This is stronger than Damas–Hindley–Milner, where the closure operation need only avoid violating scoping rules; here the constraint set introduces additional dependencies not apparent merely from the set of free usage variables of Γ , which must remain satisfiable after generalisation.

In order to generate this set, the closure operation may approximate the constraint set in some way, possibly involving unification of some variables. This is why C'_1 (the approximated constraint set) and S (the unifying substitution, to be applied to the binding types and right-hand sides) must be returned also.

Given the vector $\overline{u_k}$ of generalisable usage variables, we construct the *polymorphic* types of the binders, and use them to translate the body of the letrec.⁷ We then add the multiple-use constraints in the same way as for (\blacktriangleright_2 -ABS) (notice here another reason we must not generalise over the topmost annotations $\overline{v_i}$: if we did so, we could not set them equal to ω here). The translated term incorporates the polymorphic binder types and right-hand sides with the unifying substitution applied; we also must instantiate all recursive calls within the binding group with appropriate usage applications (see [HHPJW94]).⁸

As an aside, observe that this strategy permits only monomorphic recursion. Usage-polymorphic recursion is also permitted by the type rules, and could be inferred by using the so-called Kleene–Mycroft iteration technique of [DHM95], taking advantage of the finiteness of our annotation lattice (see also [Myc84, Hen93]). The constrained-polymorphic analysis of [GS00b, §3.9] performs inference for usage-polymorphic recursion.

The details of the closure operation are particularly interesting and also somewhat involved, so we defer a discussion of them to Section 4.5; Section 4.5.4 presents the closure algorithm itself.

⁷We abstract over the same vector $\overline{u_k}$ for each binding. It was pointed out to the author by Mark Jones [personal communication] that in general it is not necessary to quantify all types in the same mutually-recursive binding group over the same vector of variables, but redundant quantifiers introduced in this way have no ill effect (other than the size increase of the type). Jones gave the following Haskell type generalisation example (where *undefined* has type $\forall \alpha . \alpha$):

$$\begin{array}{lll} m \ x \ y & = & n \ y \quad (\text{inferred type: } \alpha \rightarrow \beta \rightarrow \gamma) \\ n \ y & = & m \ \text{undefined} \ y \quad (\text{inferred type: } \beta \rightarrow \gamma) \end{array}$$

This may be translated with explicit quantification as follows:

$$\begin{array}{lll} m \ \alpha \ \beta \ \gamma \ x \ y & = & n \ \alpha \ \beta \ \gamma \ y \\ n \ \delta \ \epsilon \ y & = & m \ \perp \ \delta \ \epsilon \ (\text{undefined} \ \perp) \ y \end{array}$$

Here n requires only two type arguments, whereas m requires all three. The constant type substituted for \perp is arbitrary (the Glasgow Haskell Compiler uses type $()$, “unit”, for this purpose). A similar example may be constructed for usage types in LIX_2 . A slightly more sophisticated (\blacktriangleright_2 -LETREC) rule would take this into account and drop unnecessary variables when generalising each binder, but for simplicity we do not attempt this.

⁸As written these substitutions induce quadratic behaviour, but careful implementation can perform all the expression substitutions (but not the type substitutions) in a single pass over the whole program.

4.4.3 Pessimisation

Once the first phase of inference has completed, and before the final phase, we must allow for separate compilation by pessimising exported functions (cf. Section 3.5.3). The addition of generalised types changes the pessimisation operation slightly: a generalised type is maximally applicable in the same way as a ω -annotated type, since it may always be instantiated at ω if necessary. The definition therefore becomes: $Pess(\Delta) \triangleq \{Pess_{\emptyset}^+(\sigma) \mid \sigma \in \text{rng}(\Delta)\}$, where

$$\begin{aligned} Pess_U^+(\tau^\kappa) &= Pess_U^+(\tau) \wedge \begin{cases} \emptyset, & \text{if } \kappa = u \text{ and } u \in U \\ \langle \kappa = \omega \rangle, & \text{otherwise} \end{cases} \\ Pess_U^-(\tau^\kappa) &= Pess_U^-(\tau) \\ Pess_U^\varepsilon(\forall u . \tau) &= Pess_{U \cup \{u\}}^\varepsilon(\tau) \\ Pess_U^\varepsilon(\sigma \rightarrow \sigma') &= Pess_U^\varepsilon(\sigma) \wedge Pess_U^\varepsilon(\sigma') \\ Pess_U^\varepsilon(\text{Int}) &= \emptyset \end{aligned}$$

This simply forces all positive annotations to either a bound usage variable or ω . Again, the resulting constraints are added to the constraint arising from \blacktriangleright_2 before passing to the final phase of the algorithm. For example, let $\Delta = \{f : (\forall u . (\text{Int}^{u_1} \rightarrow \text{Int}^{u_2})^u \rightarrow \text{Int}^{u_3})^{u_5}, g : \text{Int}^{u_6}\}$. Then $Pess(\Delta) = \{\langle u_5 = \omega \rangle, \langle u_1 = \omega \rangle, \langle u_6 = \omega \rangle\}$.

4.4.4 Inference phase 2

The final phase of the inference must, given an LIX_2 program with free usage variables and a constraint over them, find the best assignment to the variables that satisfies the constraint.

At this point, the location of and variables bound by the usage abstractions have all been determined, and the only remaining choice to be made is the values of those usage variables still free. Just as in Section 3.4 with LIX_1 , we have a typing with unbound variables in some annotation positions, under some set of constraints, and we wish to select the best one. Exactly the same algorithm we applied previously (Section 3.5.4) is appropriate here: take the best solution to the constraint according to the goodness ordering \sqsubseteq , and apply the resulting substitution to the program. Generalised variables have already been bound, and are unaffected; taking these as fixed, the result is the corresponding well-typed LIX_2 program with the maximum number of \bullet -flags, as required. Since the ordering is lub-closed, there is always a best solution (Section 3.4.2).

4.5 Generalisation and simple polymorphism

This section describes a key contribution of the present thesis: it describes how to infer the best possible types in a simple-polymorphic language; specifically, how to perform generalisation. The language LIX_2 does not in general admit principal types, but Section 4.5.1 argues that it is crucial to ensure the weaker property of *maximal*

applicability. Section 4.5.2 illustrates and motivates our approach to generalisation, considering the simple case of closed types only. This approach is extended in Section 4.5.3 to the general case of types within a typing environment. Section 4.5.4 presents the resulting closure algorithm and describes its operation in detail. Finally, Section 4.5.5 explains the extensions required to the constraint solver of the previous chapter.

4.5.1 Most-general types and maximal applicability

Milner’s paper [Mil78, pp. 354–355] clearly states the intent of let-polymorphism: the behaviour of $\text{let } x = e \text{ in } e'$ should be exactly the same as that of $e'[e/x]$. That is, the different uses of x should not interfere, and all possible behaviour of the term e should be exposed in the type chosen for x . This is what is meant by *principal* or *most general* type. It turns out that with our simple-polymorphic type system, most-general types do not always exist. However, there is a related but weaker property, *maximal applicability*, that we can and must achieve, and which we have already encountered in another context.

Even if we cannot ensure that $\text{let } x = e \text{ in } e'$ and $e'[e/x]$ behave identically with respect to usage, we must at least have that if $e'[e/x]$ is typeable then so is $\text{let } x = e \text{ in } e'$. That is, if we can infer usage annotations for the L_0 term $e'[e/x]$ (as we always can when $\text{let } x = e \text{ in } e'$ is well- L_0 -typed), then we can infer usage annotations also for the L_0 term $\text{let } x = e \text{ in } e'$. This is simply a consequence of the soft typing property (Section 1.5.5). We say that an annotated type for x satisfying the condition that it can be used in any well- L_0 -typed L_0 -context is *maximally applicable*.

From the discussion in Section 3.8 it is clear that a type is maximally applicable iff no positive annotation in it is equal to 1. Briefly, this is because of the following. The inference algorithm fails exactly when a constraint $\langle 1 \leq \omega \rangle$ is directly or indirectly asserted. The inference algorithm always uses types from the environment on the left-hand side of the subtype relation, thus:⁹ $\Gamma(x) \preceq \cdot$. Explicit annotations contained in the inference algorithm are always ω , never 1. Thus to avoid inference failure due to types in the environment, it is sufficient to ensure that no positive annotation in $\text{rng}(\Gamma)$ is ever 1.¹⁰ Observe that most-general types are always maximally applicable, but that the converse is not true in general.

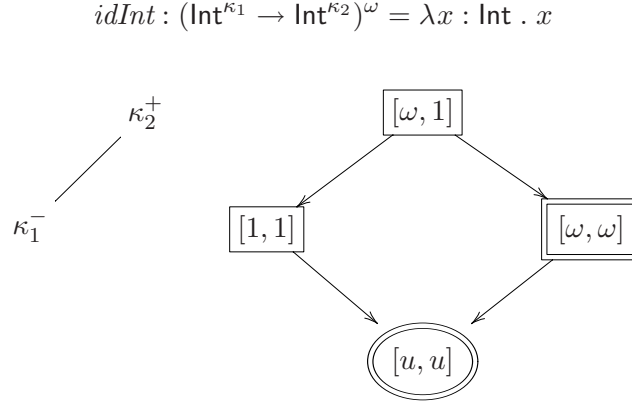
4.5.2 Which type is ‘best’?

Generalisation seeks to find the *most general type possible* for letrec-bound variables, given the restriction to simple polymorphism and the requirement that the type be maximally applicable. The present section attempts to formalise this intuitive guideline. We discuss in Section 4.7.3 below whether choosing the most general type possible is actually the best thing to do.

Consider as a first example the problem of generalising the type of $\text{idInt} = \lambda x :$

⁹See footnote 13 in Section 4.5.3.

¹⁰Note that the polymorphic annotation u is OK, because $(\blacktriangleright\text{-VAR})$ will instantiate it to a fresh variable v , and $\langle v = \omega \rangle$ merely forces v to ω without causing failure.

Figure 4.8 Constraint and subtyping lattice of $idInt$.

$Int . x$; that is, computing the closure¹¹

$$Clos(\langle \kappa_1 \leq \kappa_2 \rangle, \emptyset, (Int^{\kappa_1} \rightarrow Int^{\kappa_2})^\omega)$$

If we allowed constrained polymorphism we could obtain the (principal) type

$$(\forall u_1, u_2 : \langle u_1 \leq u_2 \rangle . Int^{u_1} \rightarrow Int^{u_2})^\omega$$

With simple polymorphism, however, this type is not available. Our intention is to *approximate* in order to avoid such constraints. The simple-polymorphic types satisfying the constraints (i.e., the possible results of generalisation) are as follows:

- (1) $(Int^\omega \rightarrow Int^\omega)^\omega$
- (2) $(Int^\omega \rightarrow Int^1)^\omega$
- (3) $(Int^1 \rightarrow Int^1)^\omega$
- (4) $(\forall u . Int^\omega \rightarrow Int^u)^\omega$
- (5) $(\forall u . Int^u \rightarrow Int^1)^\omega$
- (6) $(\forall u . Int^u \rightarrow Int^u)^\omega$

Of these, (2), (3), and (5) are not maximally applicable, and thus (as we have seen) are unacceptable. (1) is the poor solution chosen by the monomorphic analysis \mathcal{IT}_1 , as we saw in Section 4.1.1. Inspection of the remaining two types shows that (4) is no better than (1): if the result is used ω then (4) may be instantiated at ω and (1) applies directly; but if the result is used 1 then (4) may be instantiated at 1 but (1) still applies because of the subsumption rule (\vdash_2 -SUB). This leaves only type (6), which is maximally applicable yet usable at both $(Int^\omega \rightarrow Int^\omega)^\omega$ and $(Int^1 \rightarrow Int^1)^\omega$ without poisoning. We therefore return this type as result:

$$Clos(\langle \kappa_1 \leq \kappa_2 \rangle, \emptyset, (Int^{\kappa_1} \rightarrow Int^{\kappa_2})^\omega) = (\langle \kappa_1 = u \rangle \wedge \langle \kappa_2 = u \rangle, [u], \{\kappa_1 \mapsto u, \kappa_2 \mapsto u\})$$

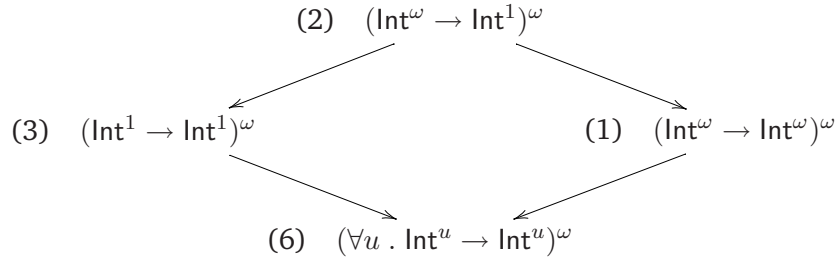
¹¹To avoid consideration of forbidden and internal variables at this stage we have assumed the topmost annotation is fixed to ω in advance and simplified the constraint by removing intermediate variables.

Our arguments here have been rather informal. To enable us to explore more formally the reasons for choosing one abstracted type over another, we introduce a *semantic* subtyping relation \preceq , distinct from the syntactic relation \preceq of Section 4.3.3 and elsewhere in this thesis. Following [Pot01, def. 12], we consider a type to denote the set of all its ground (syntactic, structural) subtype and instantiation instances.¹² Thus $\llbracket \text{Int}^\omega \rrbracket = \{\text{Int}^\omega, \text{Int}^1\}$, $\llbracket \text{Int}^1 \rrbracket = \{\text{Int}^1\}$, and $\llbracket (\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega \rrbracket = \{(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega, \{(\text{Int}^1 \rightarrow \text{Int}^1)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}\}$. Now we may define the subtype relation as reverse set inclusion: $\sigma \preceq \sigma'$ iff $\llbracket \sigma \rrbracket \supseteq \llbracket \sigma' \rrbracket$. Clearly this relation strictly includes the syntactic, structural one. It also accurately reflects the behaviour of our type system, which has arbitrary instantiation and subsumption: if a variable in the environment has type σ , then it may be used at exactly those types σ' where $\sigma \preceq \sigma'$. In this model, “more general” simply means “smaller in the semantic subtype ordering”.

Returning to the example, we may now consider the denotations of the legal types for *idInt* above. κ_1 lies in a negative position, and κ_2 in a positive position. Thus we have

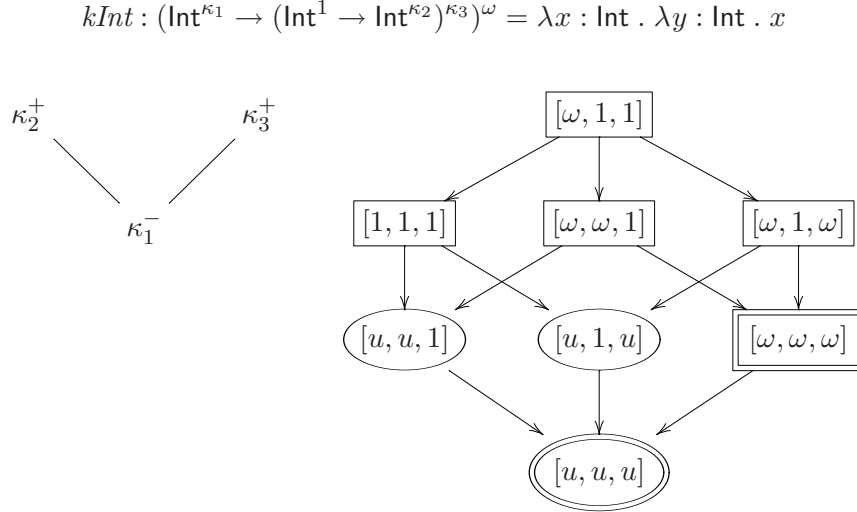
- (1) $\llbracket (\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega \rrbracket = \{(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}$
- (2) $\llbracket (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega \rrbracket = \{(\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}$
- (3) $\llbracket (\text{Int}^1 \rightarrow \text{Int}^1)^\omega \rrbracket = \{(\text{Int}^1 \rightarrow \text{Int}^1)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}$
- (4) $\llbracket (\forall u . \text{Int}^\omega \rightarrow \text{Int}^u)^\omega \rrbracket = \{(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}$
- (5) $\llbracket (\forall u . \text{Int}^u \rightarrow \text{Int}^1)^\omega \rrbracket = \{(\text{Int}^1 \rightarrow \text{Int}^1)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}$
- (6) $\llbracket (\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega \rrbracket = \{(\text{Int}^\omega \rightarrow \text{Int}^\omega)^\omega, \{(\text{Int}^1 \rightarrow \text{Int}^1)^\omega, (\text{Int}^\omega \rightarrow \text{Int}^1)^\omega\}\}$

Now observe that (4) and (5) are redundant: they are equivalent to (1) and (3) respectively. We prefer structural subtyping over quantification where the two are equivalent, and so retain (1) and (3), because the former yields ground annotations which are both cheaper to manipulate and easier to interpret: by this expedient we know that variable annotations occur only when the analysis cannot assign 1 or ω , *i.e.*, only when there is a dependency between usage annotations, as we intended in our introduction (Section 4.1.3). This for example allows the code generator to consider only 1-annotated thunks to be single-entry, rather than having also to investigate certain u -annotated thunks somehow (hence the simple definition of \cdot^\dagger in Section 4.3.1). We may now put the remaining four types in a subtyping lattice, thus:



This clearly shows that the generalised type $(\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega$ is a subtype of all other types of *idInt*, and is therefore the most general type for *idInt*; it is also maximally applicable.

¹²This approach is related to the so-called *ideal* model [MPS84] [CW85, §3] in which types denote certain sets of values, and more closely to the *powertypes* of [Car88].

Figure 4.9 Constraint and subtyping lattice of $kInt$.

We may abbreviate the constraint set arising from the definition of $idInt$ and the corresponding subtyping lattice as shown in Figure 4.8, where it is implicit that the usage annotations are given in order, and that free usage variables are quantified. Ground instances are denoted by rectangular nodes; maximally-applicable types are denoted by double borders; arrows point to smaller types in the subtype ordering.

We may treat a function like $kInt$ similarly (Figure 4.9). Once again, the generalised type $(\forall u . \text{Int}^u \rightarrow (\text{Int}^1 \rightarrow \text{Int}^u)^u)^\omega$ covers all the others.

This works for combinations of the above patterns of constraints as well: $\searrow \swarrow$, $\swarrow \searrow$, and even \bowtie all admit simple-polymorphic most-general types; the lattice for the latter (sometimes called a *bowtie* for obvious reasons) is shown in Figure 4.10.

So far it seems like we have a better solution than we had hoped for – there has been a unique best solution in each case – but the problems considered have been very simple. The next-smallest problem is exemplified by the functions *twice* and *plus3*, which both have the same set of constraints and subtyping lattice (Figure 4.11).

It is clear from this diagram that no single simple-polymorphic type can cover all the instances. The best candidates, namely the minimal ones (none of them is minimum), are $[1, 1, u, u]$, which omits $\{[\omega, \omega, \omega, \omega], [\omega, \omega, \omega, 1]\}$, $[u, u, \omega, \omega]$, which omits $\{[1, 1, 1, 1], [\omega, 1, 1, 1]\}$, and $[u, u, u, u]$, which omits just $\{[1, 1, \omega, \omega]\}$. Of these, only the latter two are maximally applicable. The algorithm presented in Section 4.4 chooses the latter, but if the omitted instance is desired in a particular case a different choice may be preferable (Section 4.7.2). The usage-generalised types chosen by the algorithm for the functions above are thus:

$$\begin{aligned} twice & : (\forall u . (\text{Int}^u \rightarrow \text{Int}^u)^\omega \rightarrow (\text{Int}^u \rightarrow \text{Int}^u)^\omega)^\omega \\ plus3 & : (\forall u . \text{Int}^u \rightarrow (\text{Int}^u \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^u)^u)^\omega \end{aligned}$$

Figure 4.10 Bowtie constraint and subtyping lattice.

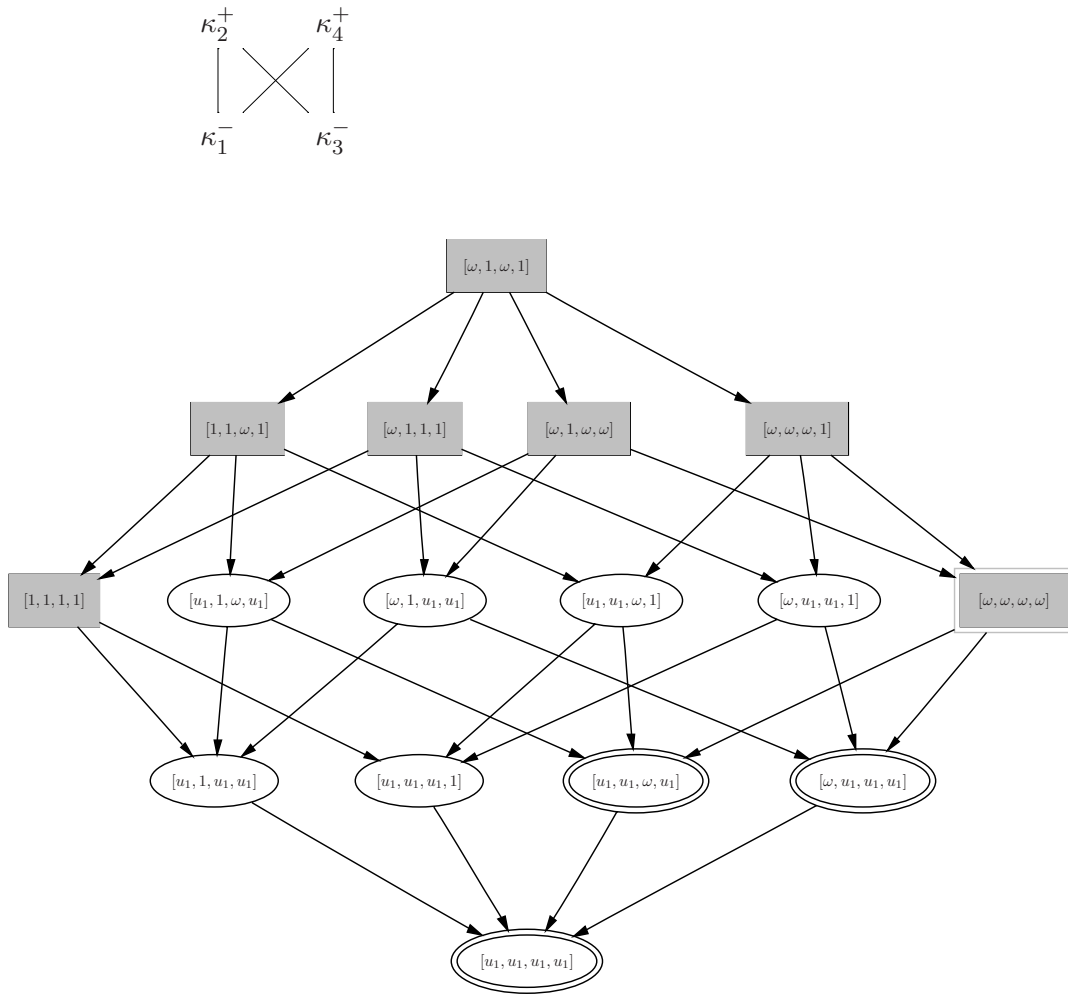
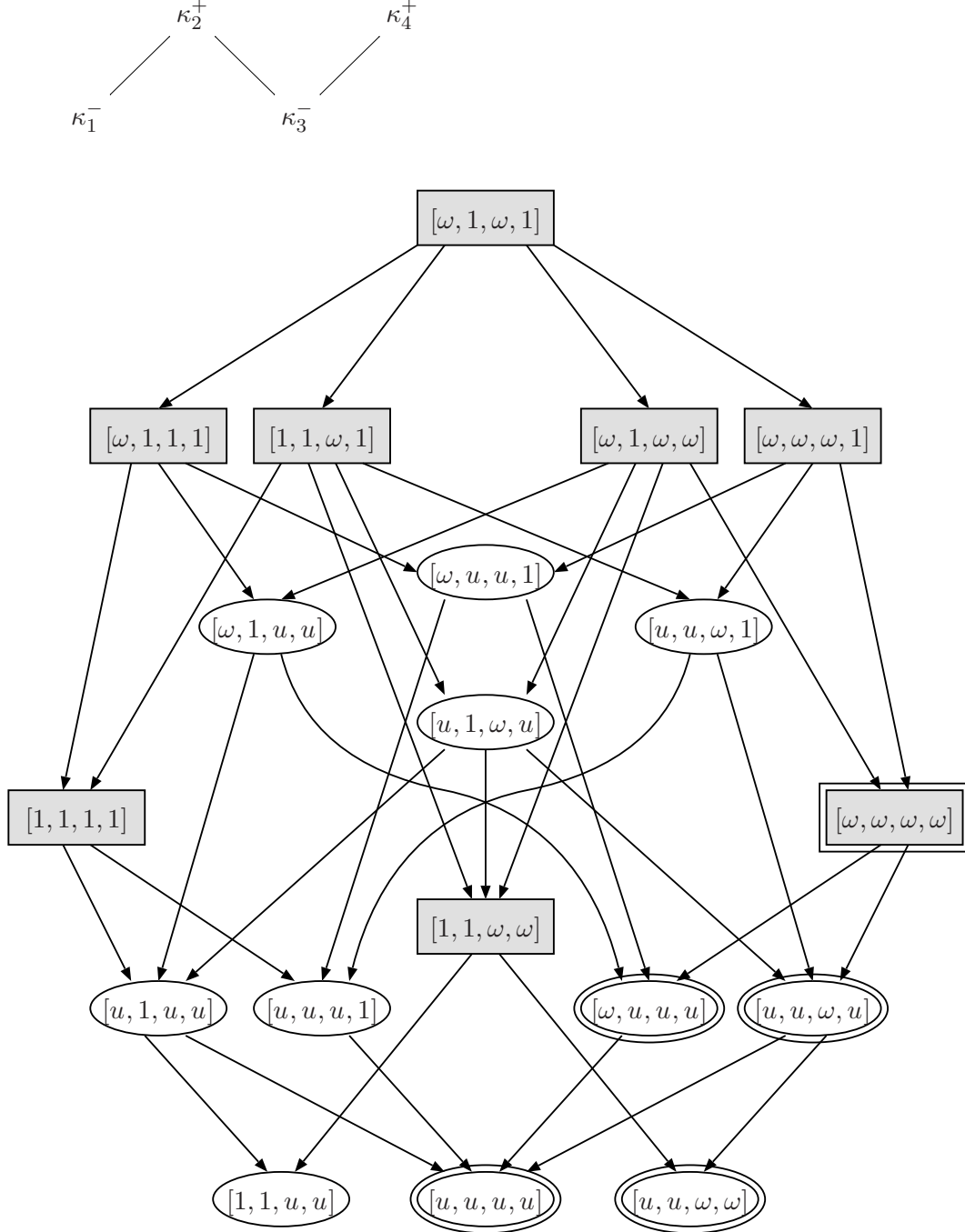


Figure 4.11 Constraint and subtyping lattice of *twice* and *plus3*.

$$\begin{aligned}
\text{twice} & : ((\text{Int}^{\kappa_2} \rightarrow \text{Int}^{\kappa_3})^\omega \rightarrow (\text{Int}^{\kappa_1} \rightarrow \text{Int}^{\kappa_4})^\omega)^\omega = \lambda f . \lambda x . f (f x) \\
\text{plus3} & : (\text{Int}^{\kappa_3} \rightarrow (\text{Int}^{\kappa_1} \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^{\kappa_2})^{\kappa_4})^\omega = \lambda x . \lambda y . \lambda z . x + y + z
\end{aligned}$$



4.5.3 Forbidden variables

So far we have considered only the simple case in which the only variables are those occurring in the type of the binder (*i.e.*, *candidate* variables), and all constraints are between these variables. In general, as we saw in Section 4.4.2, the situation is complicated by the presence of forbidden and internal variables. *Forbidden* variables are variables that are outside the scope of the generalisation operation, and therefore must not be generalised: variables occurring free in the type environment Γ , and the topmost annotations of the binder types (these are syntactically ungeneralisable, as described in Section 4.2.1). *Internal* variables are generated during inference but do not appear in the binder types or environment; they do however participate in the constraint and may annotate lambdas and bindings within the binding right-hand sides.¹³

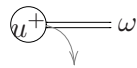
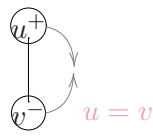

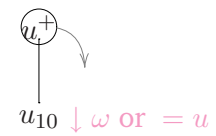

In generalising we must deal appropriately with constraints over all three sorts of variable, making generalisation for LIX_2 significantly more difficult than for a type system without constraints such as that of Standard ML. In such a type system, the only restriction on generalisation is that generalised variables must not themselves be forbidden (see Section 4.3.2). With constraints, however, we must also restrict generalisation of variables *related to* forbidden variables, in order to preserve soundness. Consider a constraint $\langle u_1 \leq x \rangle$, where x is forbidden. Generalising u_1 introduces a universal quantifier, meaning u_1 may now take any value in $\{\omega, 1\}$. Thus the constraint may only be satisfied by setting $x = 1$ or by unifying the two variables with $x = u_1$. But the former is potentially unsound and the latter is impossible: forcing x to 1 may conflict with subsequent constraints,¹⁴ and it is outside the scope of u_1 's quantifier (this is the definition of forbidden) and thus ununifiable.

We will generalise *only where required*, thus ensuring that generalised variables represent actual dependencies. This has two consequences. Firstly, since the only communication between the constraint and subsequent usage sites is via the type of the binder, it only makes sense to generalise candidate variables: generalising an internal variable would lead to types such as $\forall u . \text{Int}^1 \rightarrow \text{Int}^\omega$ in which the generalisation serves no purpose. Secondly, since subsumption means that positive candidate variables can be minimised and negative candidate variables maximised without affecting applicability, it only makes sense to generalise variables that appear both positively *and* negatively in the binder type (see Section 4.1.3). These arise from a *dependency*, where a positive candidate variable is constrained to be greater than a negative candidate variable. In this case the monomorphic solutions are incomparable, and instead we unify the variables and generalise.

¹³It is occasionally useful to know the direction of the constraints generated by the inference. From the discussion in Section 3.8 footnote 21 (which still holds in the present context) we know that positive (negative) variables from the type environment are constrained only from above (below); further, from the explicit subtyping applied to the binder type (clause 3 of $\blacktriangleright_2\text{-LETREC}$), Figure 4.7), we know that positive (negative) variables in the binder type are constrained only from below (above) unless they occur recursively (recursive occurrences appear as variables from the type environment, and so are constrained oppositely). Internal variables may be constrained from either direction.

¹⁴The variable x is either a topmost annotation or a negative environment annotation. If x is a topmost annotation and the binder it annotates is used more than once in its scope, $x = \omega$ will be asserted by $\blacktriangleright_2\text{-LETREC}$. If x is a (necessarily negative) environment annotation, it may be constrained to lie below a positive environment annotation, and this annotation may be constrained to ω at a later point, thus forcing $x = \omega$ also. Both of these cases would lead to an insoluble constraint set.

A candidate variable may only be generalised if it may safely take all values in the annotation domain $\{\omega, 1\}$. (In the discussion that follows, recall $\omega \leq 1$, i.e., downwards means towards ω and upwards means towards 1; subsumption pulls positive candidate variables downward and negative candidate variables upward.) There are four different ways in which any given candidate variable may be constrained, and in only two of them is generalisation possible:

- | | | | |
|---|--|---|---|
| 1. If the candidate is constrained to take a <i>particular value</i> , it cannot be generalised. ¹⁵ |  | ✗ | |
| 2. If the candidate is constrained to lie above (below) <i>another candidate variable</i> , with the lower variable negative and the upper variable positive, then the constraint expresses a dependency and the candidate can only be generalised by unifying the two variables. The other candidate variable must also be generalisable. |  | ✓ |  |
| 3. If the candidate is constrained to lie above (below) an <i>internal variable</i> , then it can only be generalised if the internal variable is either assigned ω (1) or unified with the candidate variable. To ensure that generalisation is used only to express a dependency, we unify internal variables only when they lie both above <i>and</i> below a generalised variable. |  | ✓ | |
| 4. If the candidate is constrained to lie above (below) a <i>forbidden variable</i> , we cannot generalise it, since this would involve forcing the forbidden variable to ω (1). |  | ✗ | |

This is essentially an informal description of the closure algorithm given in more detail in Section 4.5.4.

For example, consider the following program, where additional usage annotations are used to indicate the usage types of subexpressions:

```

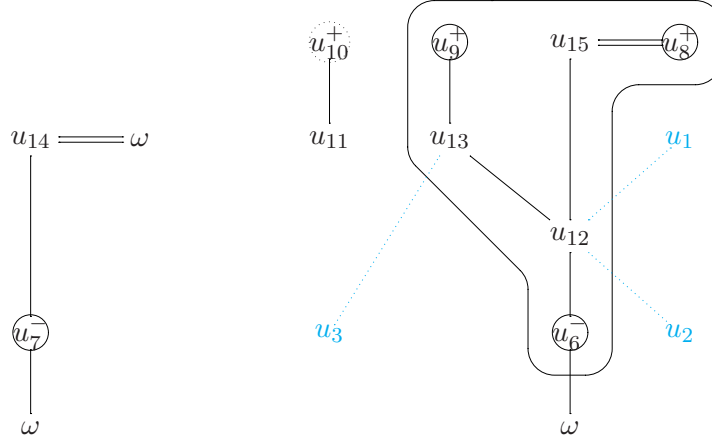
letrec sumdown : ( $\text{Int}^{u_6} \rightarrow (\text{Int}^{u_7} \rightarrow \text{Int}^{u_8})^{u_9}$ ) $^{u_{10}}$ 
      =  $\lambda^{u_{11}} s : \text{Int}^{u_{12}} . \lambda^{u_{13}} i : \text{Int}^{u_{14}} .$ 
      (if0  $i^{u_{14}}$ 
       then  $s^{u_{12}}$ 
       else (sumdown ( $s^{u_{12}} + i^{u_{14}}$ ) $^\omega$  ( $i^{u_{14}} + -1^\omega$ ) $^\omega$ ) $^{u_8}$ ) $^{u_{15}}$ 
in sumdown 100

```

The constraint generated just prior to closure of *sumdown* is as follows; candidate variables are circled, and forbidden variable u_{10} is dotted. For discussion purposes

¹⁵In our simple two-point lattice, a variable's range is either unrestricted or restricted to a single value; this is likely not to be the case for the extension discussed in Appendix C, and so this part of the definition will need to be reconsidered.

we have added three extra variables u_1, u_2, u_3 , along with the dotted constraints; these are *not* part of the constraint generated by the program above.



Ignoring the additional variables and constraints, we can see that the closure algorithm will unify variables $\{u_6, u_{12}, u_{13}, u_9, u_{15}, u_8\}$ and generalise them with a single variable. No other variables will be generalised or altered: u_{10} may not be generalised as it is a topmost annotation, and u_7 may only take the value ω (in addition to the fact that neither u_{10} nor u_7 are dependent on any other candidate variable).

If u_1 and u_2 were present as internal variables, u_1 would be forced to 1 and u_2 to ω , since respectively they lie above and below the generalised variables. If u_3 was present as a (positive¹⁶) environment variable, generalisation would be prevented.

4.5.4 The closure algorithm

The closure operator $Clos(\cdot, \cdot, \cdot)$ determines the appropriate generalisation of the binder types, based on the constraint set and the type environment. It is the core of the inference, and the major technical contribution of the present chapter. It chooses a generalisation which is ‘best’ consistent with the discussions above, approximating the constraint set where necessary.

Figure 4.12 shows the closure algorithm. Recall from Section 4.4.2 the interface of the closure operation:

$$Clos(C_0, \Gamma, \overline{\tau_i^{\kappa_i}}) = (C', \overline{u_i}, S)$$

where $\overline{\tau_i^{\kappa_i}}$ is the vector of (monomorphic) binder types to generalise, Γ is the type environment within which to generalise, and C_0 is the constraint over these types. The output is a reduced constraint set, a vector of usage variables over which to generalise, and a unifying substitution to be applied. We now discuss the algorithm step by step.

1. **Compute variable sets.** The algorithm begins by computing the sets G_0 of candidate and F_0 of forbidden variables. The candidate variables come from

¹⁶Polarities in the environment pull in opposite directions from polarities in the binder type; see footnote 13 above.

Figure 4.12 The closure operation.

$$\begin{array}{c}
\text{initial constraint} \\
\text{initial environment} \\
\text{types to generalise} \\
\text{candidate variables} \\
\text{forbidden variables} \\
\text{final constraint} \\
\text{variables to generalise} \\
\text{unifying substitution}
\end{array}$$

$$\begin{aligned}
Clos(C_0, \Gamma, \overline{\tau_i^{\kappa_i}}) &\triangleq PClos(C_0, G_0, F_0) \triangleq (C', \overline{u_i}, S) \quad \text{where} \\
G_0 &= \{u^\varepsilon \mid u \in fuv^\varepsilon(\overline{\tau_i})\} \\
F_0 &= fuv(\overline{\kappa_i}) \cup fuv(\Gamma) \\
(C, S_0) &= TransitiveClosure(C_0) \\
G &= G_0 \setminus \text{dom}(S_0) \\
F &= F_0 \setminus \text{dom}(S_0) \\
\Phi(A) &= G \cap \{u^-, v^+ \mid u \leq_C v \wedge u^- \in A \wedge v^+ \in A\} \\
&\quad \cap \{u^\varepsilon \mid u^\varepsilon \in A \wedge \neg \exists x \in (F \cup \{v \mid v^\varepsilon \in (G \setminus A)\}) . x \leq_C^\varepsilon u\} \\
G' &= gfp(\Phi) \\
(\sim) &= \{(u, v) \mid u \leq_C v \wedge u^- \in G' \wedge v^+ \in G'\}^{\pm*} \\
&\quad \text{where } R^{\pm*} \triangleq (R \cup R^{-1})^* \\
\mathcal{U} &= \{u \mid u^\varepsilon \in G'\} / (\sim) \\
\overline{u_i} &= \text{a vector containing one representative} \\
&\quad \text{from each equivalence class in } \mathcal{U} \\
S &= \{(x \mapsto u_i) \mid \exists u^- \in G' . u \leq_C x \wedge \exists v^+ \in G' . x \leq_C v \wedge u_i \in [u]_{(\sim)}\} \\
&\quad \cup \{(x \mapsto \omega) \mid \neg \exists u^- \in G' . u \leq_C x \wedge \exists v^+ \in G' . x \leq_C v\} \\
&\quad \cup \{(x \mapsto 1) \mid \exists u^- \in G' . u \leq_C x \wedge \neg \exists v^+ \in G' . x \leq_C v\} \\
&\quad \text{where } x \in (fuv(C) \setminus \text{dom}(S_0)) \\
C' &= SC
\end{aligned}$$

the binder types, and are marked with polarities from their occurrence.¹⁷ The forbidden variables come from the topmost annotations of the binder types (which are syntactically ungeneralisable) and from the free usage variables of the type environment Γ ; their polarity is irrelevant. Once these sets have been computed, the closure operation proper is performed by $PClos(C_0, G_0, F_0)$.

2. **Remove grounded variables.** According to point 1 of Section 4.5.3, we must not attempt to generalise any variable that is already constrained to take a particular value. To avoid this, we invoke an auxiliary partial constraint solver *TransitiveClosure* which forms the transitive closure of the constraints collected in C_0 . If this succeeds it returns a set of substitutions S for variables whose values are completely determined by C_0 , and a residual constraint set C constraining these variables directly to their values, and the remaining variables equivalently to C_0 . The substitution is then used to remove from consideration these determined variables, both from the set of candidate variables (lest we attempt to generalise them) and from the set of forbidden variables (lest we needlessly avoid generalising a variable because of an irrelevant constraint).

The residual constraint set C induces a partial order \leq_C over the free variables of C , the *transitive closure* of C_0 . We write $u \leq_C^+ v$ for $u \leq_C v$ and $u \leq_C^- v$ for $v \leq_C u$. If C_0 is unsatisfiable, *TransitiveClosure* and the entire translation fail; Theorem 4.4 in Section 4.6.2 states that this never occurs. The *TransitiveClosure* operation is a straightforward modification of the constraint solver of Section 3.5.4; see Section 4.5.5 for details.

3. **Find clusters.** Once grounded variables have been removed from consideration, point 2 of Section 4.5.3 directs us to find groups or *clusters* of dependent candidate variables (subsets of G). *Dependent* here means that each negative variable is constrained to lie below a positive variable in the same cluster, and *vice versa*; we require this to ensure that polymorphism is only used where needed. The clusters must be generalisable: no cluster should be constrained to lie above or below any other distinct cluster (point 2), and no cluster should be constrained to lie above or below a forbidden variable (a variable in F ; point 4).

The algorithm first takes the largest dependent subset G' of G not containing any variables constrained to lie above or below a variable in F , and then quotients it by the pairwise dependence relation. Each equivalence class thus generated is a distinct cluster, and can be unified and generalised as a distinct usage variable. The set G' is computed as the greatest fixed point (*gfp*) of function Φ , where the second conjunct of Φ requires variables in G' to be dependent on each other, and the third removes variables that would constrain forbidden variables or variables that have already been excluded from the set.¹⁸ (We

¹⁷If a usage variable u were to appear both positively and negatively in the binder types, it would appear twice in G_0 , once as u^- and once as u^+ . It can be shown, however, that with the inference algorithm as given this never occurs.

¹⁸We need only check that positive (negative) variables are not greater (less) than variables in F ; the other direction is ensured by the first clause and transitivity of \leq .

must take a fixed point because a constraint $x \leq_C v^+$ removes not just v^+ from the set, but also all u^- such that $u^- \leq_C v^+$, and for each such u^- all w^+ such that $u^- \leq_C w^+$, and so on.) The relation (\sim) is computed as the reflexive symmetric transitive closure of the dependency relation, and the set of clusters \mathcal{U} is computed as the quotient of G' by (\sim) .

4. **Perform unifications.** Having found the clusters of candidate variables to be generalised, we now perform the required unifications. Since the constraint C may contain many internal variables in addition to the candidate variables, we treat these as well, following point 3 of Section 4.5.3. In general, each cluster in \mathcal{U} constrains multiple internal variables between the candidate ones. If two candidate variables u and v in a cluster are to be unified and generalised, then all variables in between them must be so also. In addition, other internal variables may be constrained to lie above (or below) the cluster, and must be forced to 1 (or ω respectively).

This unification and forcing is formalised by the last three equations of Figure 4.12. First an arbitrary representative variable is picked from each cluster. Then a substitution is generated that unifies all variables lying within each cluster with the representative variable of that cluster, and constrains the variables above and below the cluster to 1 and ω respectively.¹⁹ Finally, the constraint is approximated by applying the substitution to it.

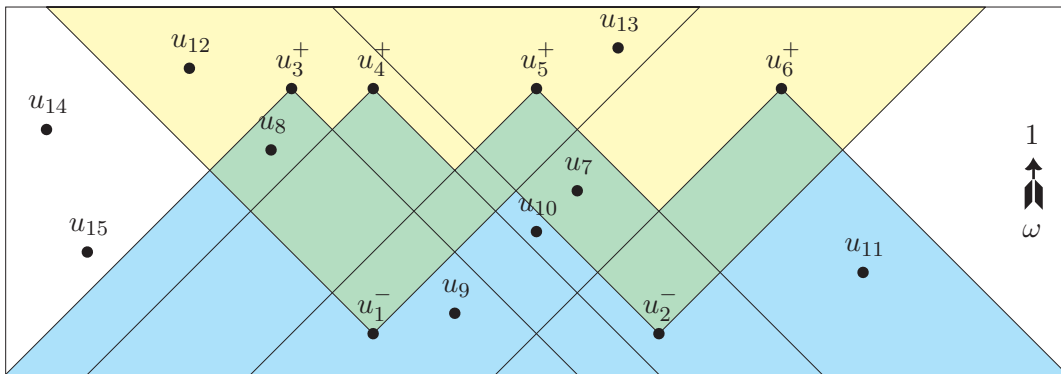
5. **Output results.** The final constraint C' , vector of variables representing the clusters to generalise $\overline{u_i}$, and substitution S are returned.

The algorithm just described never forces either forbidden or candidate variables. In some circumstances, this behaviour is not optimal. Section 4.7.2 discusses some heuristics that may be used to get better behaviour.

If desired, the behaviour of the monomorphic usage analysis of Chapter 3 may be obtained by replacing the definition of Figure 4.12 with the trivial non-closure algorithm

$$\text{TrivClos}(C, \Gamma, \overline{\tau_i^{\kappa_i}}) \triangleq (C, [], [])$$

Steps 3 and 4 of the algorithm may be a little hard to follow. We can visualise them as follows:



¹⁹The domain of the substitution will never include forbidden variables (i.e., $x \notin F$ for all $x \mapsto u_i$ in S), since the third conjunct in the definition of Φ ensures that any constraint between a forbidden variable and a cluster prevents the cluster from being generalised.

Here we visualise the constraint as a kind of abstract Hasse diagram; variables of interest are shown explicitly, and the remainder are indicated simply by regions of the plane. The diagram shows a single cluster of dependent candidate variables $\{u_1, u_2, u_3, u_4, u_5, u_6\}$, with internal variables constrained to lie above the cluster shaded in yellow, internal variables constrained to lie below the cluster shaded in cyan, and internal variables constrained to lie both above and below the cluster (and hence *within* the cluster) shaded in green.

Positive variables in the diagram pull downwards toward ω due to the subtype ordering, and negative variables pull upwards toward 1. Where a positive variable is constrained to lie above a negative variable, the two directions compete and we unify the variables, approximating the constraint between them.

All variables in the green region, including the candidate variables defining the cluster and the internal variables u_7 and u_8 , will be unified with a single variable and generalised. All variables in the cyan region, including u_9, u_{10}, u_{11} , will be forced to ω so as to permit this generalisation, and the variables in the yellow region, including u_{12}, u_{13} , will be forced to 1 for the same reason. This unification approximates away all constraints lying within these three regions. The white region unconnected with the cluster, including u_{14}, u_{15} , is unaffected.

4.5.5 Computing the transitive closure

The closure operation of Section 4.5.4 makes use of an operation *TransitiveClosure*, which partially solves a set of constraints, returning a substitution and a residual constraint. This operation also converts the constraint into a form that allows us to compute $\Phi(A)$, (\sim) , S , and C' in Figure 4.12, by exposing the partial order \leq_C .

The *TransitiveClosure* operation is implemented by extending the implementation of *CS* in Section 3.5.4, as described in detail below.

The codomain of the finite mapping is extended to allow an equivalence class to be mapped to a generalised variable; the algorithm treats this exactly as if the class was mapped to a constant. When *TransitiveClosure* is applied to a constraint set, it performs the algorithm of Section 3.5.4 on the set, building the data structure described there or failing if the constraint set is unsatisfiable. It halts once all the constraints have been considered, but before attempting to determine the optimal solution.

Operations are provided to obtain the substitution implied by the data structure (obtained by restricting the finite map to those variables mapped to a constant), to return the set of variables constrained by the data structure (the domain of the finite map), and to reify the data structure as a constraint.

In addition, an operation is provided that queries a variable constrained by the data structure, returning (unless it is mapped to a constant) its *upward* and *downward cones*, i.e., the sets of variables constrained to lie respectively above and below the variable. These sets are computed by a transitive closure algorithm for digraphs due to Nuutila [Nuu94], using the graph for which the nodes are the variables in the finite map and the directed edges from each node are the upper or lower bounds

of each variable (respectively). The algorithm is modified so as to obtain the set of nodes in the transitive closure, rather than the set of components.

Finally, an operation implements the final lines of Figure 4.12, computing S and C' , directly on the data structure, using set union and intersection on the upward and downward cones of the variables in each equivalence class to determine the sets of variables to be assigned to u , 1 , and ω .

These operations, in conjunction with those already described in Section 3.5.4, are sufficient to implement the closure algorithm of Section 4.5.4, and hence the entire inference \mathcal{IT}_2 .

4.6 Proofs

Recall from Section 3.1 that a type-based analysis must be supported by proofs that the type system and inference are sound, and preferably complete. It is also useful to know the complexity of the analysis. The proofs of these properties are presented (in sketch form) below; the full proofs appear in the appendix.

4.6.1 Well-typing rules

Once again, correctness requires that the type system be sound with respect to the operational semantics (cf. Section 3.6.1): a well-typed LIX_2 term e must never terminate in a configuration in *Wrong*, *BadBinding*, or *BadValue*; it must either fail to terminate, or terminate in a configuration in *Value* or *BlackHole*.²⁰

The operational semantics we use for LIX_2 is that of LIX_0 , with the instrumentation modified in two ways. Firstly, the types carried are polymorphic τ - and σ -types rather than t -types. Secondly, rules are added for usage abstractions and applications, and the rule for *letrec* is modified. The details are a little involved, but as they are identical to the way in which type abstractions and applications are handled in Section 5.1.4, we defer discussion to that section. As usual, an analogue of the Correspondence Lemma 2.4 states that the instrumentation is ignored, i.e., that LIX_2 and LX execute in lock-step.

Theorem 4.1 (Type soundness)

For all $e \in LIX_2$, if $\emptyset \vdash_2 e : \sigma$ and there exists a configuration C' such that $(e)^{\langle \cdot \rangle} \downarrow C'$, then $C' \in \text{Value} \cup \text{BlackHole}$.

Proof By Progress Lemma D.10 (proven by cases on C), and induction on the length of derivation of $(e)^{\langle \cdot \rangle} \downarrow C'$. The full proof is given in the appendix, Theorem D.11. \square

We further expect that its behaviour will be the same as that of the equivalent L_0 term $M = (e)^{\dagger}$: if M terminates with result V (or a black hole), then we expect e to terminate with the same result (or a black hole, respectively), and *vice versa*. The latter property can be expressed as follows:

²⁰Note that due to type erasure, *BlackHole* is extended to include some configurations of the form $\langle H; \Lambda u . x; S \rangle$.

Theorem 4.2 (Correctness)

For all LIX_2 target programs e , where $\emptyset \vdash_2 e : \sigma$, let M be the corresponding L_0 source program $(e)^\sharp$. Then we have

- (i) $(e)^{\langle \cdot \rangle} \downarrow \Leftrightarrow (\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow$
(i.e., the LIX_2 program e terminates iff the L_0 program M does); and
- (ii) If $(e)^{\langle \cdot \rangle} \downarrow C'$ and $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow C''$, then all the following hold:
 - (a) $C' \in \text{BlackHole} \Leftrightarrow C'' \in \text{BlackHole}$
 - (b) $C' \in \text{Value} \Leftrightarrow C'' \in \text{Value}$
 - (c) $C' = \langle H; n; \varepsilon \rangle \Leftrightarrow C'' = \langle H'; n; \varepsilon \rangle$

(i.e., if the two programs terminate, they both terminate in the same way, viz., black hole, non-ground value, or the same ground value).

Proof By the Correspondence Lemma we may ignore the instrumentation, and consider the two FLX terms $M_2 = (e)^\flat$ and $M_0 = (M)^\flat$. The two directions of (i) are proven separately, showing by induction on the length of the respective reduction sequence that each can simulate the other and relating terminal configurations. The full proof appears in the appendix, Theorem D.17. \square

We also must show that all source terms have a target typing:

Theorem 4.3 (Nonrestrictivity)

For all L_0 environments Γ , terms M , and types t , if $\Gamma \vdash_0 M : t$ then there exists an LIX_2 environment Γ' , term e , and type σ such that $(\Gamma')^\sharp = \Gamma$, $(e)^\sharp = M$, $(\sigma)^\sharp = t$, and $\Gamma' \vdash_2 e : \sigma$.

Proof Follows immediately from Theorem 3.3, since every LIX_1 typing is also an LIX_2 typing. \square

Together, these three results strongly support our claim that the type system models our operational notion of usage. Theorem 4.1 demonstrates that well-typed programs do not go wrong: the analysis yields operationally correct update flags. Theorem 4.2 demonstrates that the observable behaviour of a well-typed LIX_2 program is identical to that of its corresponding L_0 program: the analysis does not affect behaviour. And Theorem 4.3 reassures us that there is at least one corresponding LIX_2 program for every L_0 program: the analysis is *soft* (see Section 1.5.5), and does not reject any program. However, these results do not tell us whether or not the type system chooses *good* update flags; for this we must rely on practical experience, which we gain in Section 4.7.5.

4.6.2 Inference phase 1

The inference \mathcal{IT}_2 is intended to infer a well-typed LIX_2 annotation for a given L_0 program. We now proceed to prove that this inference algorithm is sound, and has other good properties. The formal statement of these properties should be compared

with Theorem 3.4, Section 3.6.2, where the LIX_1 inference \blacktriangleright_1 is shown to be *complete* as well as sound. The difference is in clause (iv), which for \blacktriangleright_1 states that *all* well-typed annotations of the source term are solutions of the resulting constraint but for \blacktriangleright_2 states only that there is *one* solution of the resulting constraint. We explain why this incompleteness is unavoidable below.

Theorem 4.4 (Soundness of inference phase 1)

For all Γ in LIX_2 (possibly with free usage variables) and M, t in L_0 such that $(\Gamma)^\sharp \vdash_0 M : t$ and $1 \notin \text{ann}^+(\Gamma)$,

- (i) $\blacktriangleright_2(\Gamma, M) = (e', \sigma', C, V)$ is well defined²¹
(i.e., the algorithm \blacktriangleright_2 is deterministic and does not fail);
- (ii) $(e')^\sharp = M$ and $(\sigma')^\sharp = t$
(i.e., the inference algorithm merely annotates the source term, and does not alter it or its source type);
- (iii) $\forall S. \vdash^e SC \Rightarrow ST \vdash_2 Se' : S\sigma'$
(i.e., all solutions of the resulting constraint are well-typed); and
- (iv) $\exists S. \vdash^e S(C \wedge \bigwedge_{u \in (fw(\Gamma) \cup fw(\sigma'))} \langle u = \omega \rangle)$ and $1 \notin \text{ann}^+(\sigma')$
(i.e., the resulting constraint has at least one solution, and permits all possible future uses).

Proof Proofs of (i) and (ii) are straightforward. (iii) is proved by induction over the structure of the inference derivation tree and inspection of each inference rule, comparing it with the corresponding well-typing rule.

The proof of satisfiability (iv) makes use of a lemma: if $\Gamma \blacktriangleright M_i \rightsquigarrow e_i : \sigma_i; C_i; V_i$ and all the C_i are satisfiable, then the conjunction $\bigwedge_i C_i$ is also satisfiable. We proceed by structural induction on the inference derivation tree. All the rules in Figure 4.6 can be seen by inspection to preserve the well-typedness property and the lemma, since usage variables are only ever constrained to ω or a fresh usage variable. This leaves (\blacktriangleright_2 -LETREC).

Recall that the monomorphic inference \blacktriangleright_1 can be obtained simply by providing a trivial definition for $Clos$. It is therefore unsurprising that the result is established using a Lemma 4.5 (below) describing properties of $Clos$, and is parameterised over its definition.

A full proof appears in the Appendix, Section D.14. □

Lemma 4.5 (Closure operation)

For all $C, \Gamma, \overline{\sigma_i}$ such that $\exists S'. \vdash^e S'C$ (i.e., C is satisfiable),

- (i) The closure operation $Clos(C, \Gamma, \overline{\tau_i^{\kappa_i}}) = (C', \overline{u_i}, S)$ is well-defined,

²¹It is well defined modulo the names of fresh variables; we have already noted that we are omitting the details of fresh variable management. Here this also means that the list of usage variables in scope occurs in Γ of $\Gamma \vdash_2 e : \sigma$ but not in Γ of $\blacktriangleright_2(\Gamma, M)$, since in the latter the free usage variables are managed separately.

and we have the following results, where $F_0 \triangleq (fuv(\Gamma) \cup fuv(\overline{\kappa}_i))$:

- (ii) $C' =^e SC$
(i.e., a solution of the residual constraint, applied to the substituted term, satisfies all the original constraints);
- (iii) $\exists S' . \vdash^e S' C'$
(i.e., the residual constraint is satisfiable);
- (iv) For all substitutions S', S'' such that $S'|_{U \setminus \overline{u}_i} = S''|_{U \setminus \overline{u}_i}$ (where U is the set of all usage variables), we have that $\vdash^e S' C' \Leftrightarrow \vdash^e S'' C'$
(i.e., the \overline{u}_i may safely take any value; alternatively, the residual constraint is independent of the values of the \overline{u}_i);
- (v) $\text{dom}(S) \subseteq fuv(C) \setminus F_0$, and for all $x \in F_0$ and $\kappa \in \{1, \omega\}$, if $\exists S' . \vdash^e S'(C \wedge \langle x = \kappa \rangle)$ then $\exists S' . \vdash^e S'(C' \wedge \langle x = \kappa \rangle)$.
(i.e., the substitution does not attempt to touch variables to which it is not applied, and neither does C' constrain them further); and
- (vi) $\overline{u}_i \subseteq (fuv(S\overline{\tau}_i) \setminus F_0)$
(i.e., the variables \overline{u}_i are all abstractable).
- (vii) $\forall u \in fuv^+(\overline{\tau}_i) . Su \neq 1$
(i.e., no positive annotation is forced to 1).

Proof Proofs of these properties appear in the appendix, Lemma D.13. □

Attempting to prove completeness of \blacktriangleright_2 with respect to \vdash_2 fails immediately because the inference infers rank-1 usage-polymorphic types only whereas the well-typing rules permit arbitrary usage polymorphism (see Section 4.7.4). But it fails for another reason also: the choice of which variables to generalise is made based only on C, Γ , and $\overline{\sigma}_i$ after inferring the bindings but before inferring the body or the remainder of the program. Generalising a variable can be destructive because it constrains related variables to accept any value for the generalised variable; these constraints could be relaxed if the inference knew the variable would only ever take one value in the body and the remainder of the program. Also, not generalising a variable can be destructive because it fails to insulate application sites from each other; if the inference knew that certain variables (free usage variables in Γ and topmost annotations of $\overline{\sigma}_i$) forbidden at generalisation time and related to otherwise-generalisable variables would later be constrained to a constant, the constraints preventing generalisation could be relaxed. Our inference cannot know these things, and thus is incomplete even for rank-1 usage polymorphism.

Note that this is distinct from the other incompletenesses of our analysis: the lack of most-general types for LIX_2 (Section 4.5.2), and the necessary incompleteness of any static analysis with respect to the dynamic, operational-semantic property of correct usage annotation (Section 1.4.2).

4.6.3 Inference phase 2

Phase 2 of the LIX_2 inference is identical to that of LIX_1 , and the proofs already presented in Section 3.6.3 apply unchanged.

4.6.4 Overall results

We now combine the results of the above sections into a result about the inference as a whole.

Theorem 4.6 (Inference soundness)

For all M in L_0 , if $(CS \circ \blacktriangleright_2)(\emptyset, M) = e : \sigma$ then $\emptyset \vdash_2 e : \sigma$ and $(e)^\natural = M$.

Proof Follows directly from Theorems 4.4 and 3.5, and inspection of the definition of \natural . \square

Theorem 4.7 (Inference complexity)

If we assume that nesting of conditionals and abstractions is limited to a constant depth, types annotating letrec bindings are limited to a constant size, and a linear algorithm exists for union-find, then the complexity of the inference \mathcal{IT}_2 is bounded by $O(nm^2)$, where n is the size of the program and m is the size of the largest binding group (set of expressions \bar{e}_i bound by a letrec) in the program.

Proof The dominant term comes from the algorithm which computes the transitive closure of the constraints, used to obtain the upper and lower cones for variables in G_0 in order to compute Φ , S , and C (see Section 4.5.5). Nuutila [Nuu94] gives a worst-case bound of $O(ne + n + e)$ for his algorithm, where n is the number of vertices and e the number of edges. Both n (the number of variables in the constraint) and e (the number of constraints) may be approximated in our setting by the size of the constraint C passed to $Clos$, which (under the assumptions given) is bounded by the size of the binding group being generalised. Thus the closure operation is quadratic in this parameter, and since it is invoked at each letrec this must be multiplied by the size of the program to obtain its overall contribution. This dominates remaining factors, which are essentially linear, as before (Theorem 3.9 for \mathcal{IT}_1). \square

The assumption on types of letrec bindings is a little strong, but Henglein argues in relation to ML that “Good programs have small types”; since types are in some sense abstractions of program behaviour, no reasonable program has a huge type [Hen93, §6.1]. This is well supported by the fact that for over ten years ML’s type inference was believed to be linear or possibly quadratic, whereas in fact it is doubly-exponential [KMM91]! Clearly in practice functional programmers do not write code of the kind that leads to such behaviour; the same observation restricts our exposure in this case.

4.7 Discussion

There are a number of ways in which the inference could be extended or improved; we discuss these below in Sections 4.7.2, 4.7.3, and 4.7.4. We also summarise the results obtained from our implementation of the analysis in Section 4.7.5. Firstly, however, we consider the possible wider applications of simple polymorphism.

4.7.1 Application of simple polymorphism

Simple polymorphism is not tied to usage analysis. We believe that it should have wider applications to type-based analyses. One obvious candidate is the binding-time analysis of Dussart, Henglein, and Mossin [DHM95], which presently uses constrained polymorphism over a two-point domain (static vs. dynamic). Uniqueness [BS95b, BS96] (Section 3.9.2) would be another example.

All these analyses are over two-point annotation domains. Simple polymorphism should certainly be applicable to any domain possessing a top and a bottom element. However, it is not clear whether in larger domains the approximation performed by simple polymorphism would be too great to be useful.

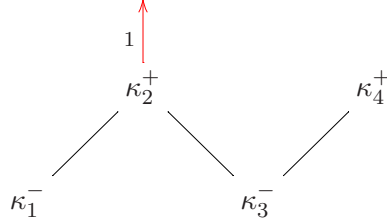
4.7.2 Better generalisation

The generalisation algorithm described in Section 4.5.4 may certainly be improved. Generalisation is presently prevented whenever there is a dependency on a forbidden variable at the time of generalisation. However, in certain cases that apparent dependency may later be removed, or steps could be taken to partially generalise anyway, or the generalisation algorithm could remove the dependency itself. All of these could improve the generality of the results of inference.

Observe that generalisation may be prevented by a variable occurring in the type environment *whenever that variable is free at generalisation time*, even if it will later be constrained to ω or 1 and thus unlinked from the candidate variables. This suggests that an inference algorithm should take care to constrain usage variables in the type environment as soon as possible, replacing fully-constrained variables with constants. Unfortunately, this is not always readily achieved: some annotations may be forced to ω early because they annotate a binder that is to be pessimised, but others will only be forced to ω based on knowledge of their occurrence or use as arguments to other functions, knowledge that is only available after inference of the body of the letrec which is performed after generalisation.

In general, no LIX_2 inference can avoid approximating (Section 4.5.2). But in specific cases, one approximation may be better than another: if *plus3* is used in a context where the second and third arguments are always supplied together, the in-general-unsafe type described by $[1, 1, u, u]$ can be used rather than $[u, u, u, u]$. If some heuristic is available to make such choices, it can be used to influence the

outcome of inference appropriately. If κ_2 is forced to 1:

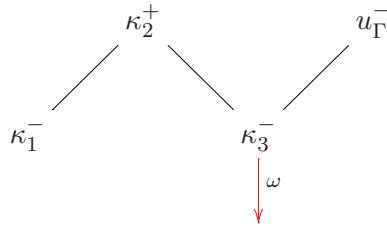


then κ_1 will be maximised to 1 also, and κ_3 and κ_4 unified, yielding the desired type:

$$(\forall u . \text{Int}^u \rightarrow (\text{Int}^1 \rightarrow (\text{Int}^1 \rightarrow \text{Int}^\omega)^1)^u)^\omega$$

Forcing to 1 requires examination of the remainder of the program to ensure that inference will not subsequently fail; forcing to ω on the other hand is always safe, although it may lead to poisoning or loss of usage information.

An interesting application for such heuristics therefore is to select a negative candidate variable to force to ω , in order to break a zig-zag link between a forbidden variable and a cluster which it is preventing from generalisation. In



the variable u_Γ occurs negatively in the environment and is therefore preventing κ_1 , κ_2 , and κ_3 from being unified and generalised. If we let $\kappa_3 = \omega$ as shown, the zig-zag is broken and κ_1 and κ_2 may be unified and generalised, at the expense of κ_3 . Similarly, longer zig-zags that would normally all be unified to a single variable may be split into multiple variables by judicious forcing. The present algorithm does not attempt these things, but heuristics controlling such behaviour can be imagined.

An even more unorthodox heuristic might determine that forcing an environment variable (or some other forbidden variable) is called for. In the example at the end of Section 4.5.3, we suggested imagining u_3 was a positive variable in the environment, preventing u_9, u_6, u_8 from being unified and generalised. If we were to force $u_3 = \omega$, these variables would be generalisable. One can imagine situations in which gaining generalisation of this type is worth the possible loss of a 1-type for u_3 : for example, if u_3 is the topmost annotation of a toplevel function, it is highly likely it will be used more than once and thus forced to ω eventually anyway.

In our presentation of the algorithm, we avoid such heuristics, and simply choose the largest possible forcing-free generalisations.

4.7.3 Usage analysis and generalisation

A significant observation, which we made after designing and implementing the closure algorithm, is that generalisation and usage analysis in fact work against each

other to a significant degree. Principal types, and our weaker maximal applicability property, aim to find a single type that describes *all possible uses* of the binder (Section 4.5.1); usage analysis seeks to discover which *actual uses* are made of the binder in order to optimise for them alone. Thus generalisation erects a barrier to usage analysis: the generalised expression is permitted no information on how it is used, and the abstract machine must pessimistically assume all possible uses (thus $u^\dagger = !$).

Henglein and Mossin, in presenting a binding-time analysis [HM94], note that their system has principal types and explain that “[t]his admits modular (‘local’) binding-time analysis of a (function) definition, independent of any of its applications.” While this may be appropriate and useful for a binding-time analysis, it is exactly the opposite of what we desire in a usage analysis. Usage types are intended precisely *to* give information about function applications. This information is inherently *not* local to the definition, but is a *global* property of the contexts in which the function is used [Gus01a, p. 4]. This suggests that it may be the case that no useful usage analysis has principal types!

This suggests that wholesale generalisation of every letrec-bound variable will lead to poor results. Contrast

```
letrec  $x = e$ 
in
...  $x$  ...
```

(where x occurs only once in the body), with the inlined version

```
...  $e$  ...
```

In the former, the generalisation procedure approximates the constraints pertaining to e , potentially leading to a less informative type; in the latter, the exact constraints are used and all available information is retained. The good behaviour of the latter example may be preserved in the former by simply not generalising: since there is only one occurrence of x , there is no potential for poisoning, and since there is no generalisation, there is no approximation. If there is only one occurrence of the binder, then, simply not generalising has the same effect as using constrained polymorphism! Even if there is more than one occurrence, they may be at sufficiently similar types that not generalising is the better choice. Therefore, it may be wise to use the monomorphic letrec in most places, and reserve the polymorphic generalising version for toplevel or exported functions only. This is borne out in our results (Section 6.8).

The fact that generalising the topmost annotation is syntactically impossible (Section 4.2.1) alleviates the problem somewhat by ensuring that the overall usage of the function as a whole, at least, is fixed statically.

Recall, however, that the benefit to the function of eschewing principal types is balanced by a disadvantage to the caller: if a function with a dependency, such as *idInt* (Section 4.1.1) is called in both contexts (or is exported), then in the absence of polymorphism argument and result must both be annotated ω . This causes those contexts in which the result is in fact used only once to be needlessly poisoned by the ω annotation on the second argument. Principal types guarantee that the function cannot needlessly poison its context.

An alternative with which we experiment in Section 6.3.3 is *specialisation*: a function $f : (\forall u . \tau)^\omega = e$ can be replaced by two functions $f_1 : (\tau[1/u])^\omega = e[1/u]$ and $f_\omega : (\tau[\omega/u])^\omega = e[\omega/u]$, and all former invocations of f 1 and f ω may be replaced respectively by invocations of f_1 and f_ω . This allows thunks within f to be given accurate update flags, at the expense of an increase in code size (for a function with n usage arguments, up to 2^n copies are required). In some cases this may be beneficial.

The Holy Grail, of course, would be to design a constraint system that could defer the decision of which variables to generalise until the end, when all available information about the program has been collected. This would yield inference completeness results analogous to those for LIX_1 and \blacktriangleright_1 , namely that the inference always yields *the best* target typing of the source program (Theorem 3.8).

4.7.4 Beyond ML-style polymorphism

Section 4.4.2 explains the decision to use ML-style polymorphism, generalising (and approximating) at letrec nodes only and using rank-1 usage polymorphism. But the well-typing rules admit more general behaviour. For example, there exist L_0 programs which can be given complete LIX_2 types only by using rank-2 polymorphism, and which will be unnecessarily approximated by the algorithm. Such programs involve passing a generalised function as argument to another function, and then using it polymorphically. Here is a contrived example:

$$\begin{aligned} applyIdInt_{L_0} &= \lambda f : (\text{Int} \rightarrow \text{Int}) . \lambda x : \text{Int} . f\ x \\ applyIdInt_{LIX_2} &= \lambda^{\omega,!} f : (\forall u . \text{Int}^u \rightarrow \text{Int}^u)^\omega . \Lambda u . \lambda^{\omega,!} x : \text{Int}^u . f\ u\ x \end{aligned}$$

With this hand-generalised function, the application $(applyIdInt\ idInt)$ is still usage-polymorphic, and may be applied both to a second argument that may be used at most once (returning a result that may be used at most once) and to one that may be used many times (returning a result that may be used many times). In contrast, the present inference algorithm yields the generalisation

$$applyIdInt_{LIX_2} = \Lambda u_1\ u_2 . \lambda^{\omega,!} f : (\text{Int}^{u_1} \rightarrow \text{Int}^{u_2})^\omega . \lambda^{\omega,!} x : \text{Int}^{u_1} . f\ x$$

in which the values of both u_1 and u_2 are fixed once the first argument is supplied, and application to second arguments of differing usage results in inference failure or poisoning.

This example is clearly contrived, but in other contexts rank-2, *a.k.a.* nested or impredicative, polymorphic types have in fact been shown to be useful (such as in the lazy state threads of [LPJ94]) and they might conceivably prove to be so here. Inference is known to be undecidable in this case [Wel94], but such types could be introduced in specific cases by explicit type signatures or a special-case algorithm. Few alterations would need to be made in order to permit their use; see [OL96], which describes the method used in GHC to implement rank-2 type polymorphism.²² One possible approach for inferring such types suggests itself from the common usage-variable-follows-type-variable pattern shown clearly by the type inferred for *compose*

²²Simon L. Peyton Jones, personal communication, 17 September, 2001.

(Figure 4.1): we could attempt to follow each nested type quantifier with a usage quantifier, possibly improving the type of functions such as *build*. We have not tried this.

It is possible to approximate at locations other than letrec nodes, as in Nordlander’s O’Haskell, discussed in Section 4.8.2.

4.7.5 Implementation

We have implemented our analysis within the Glasgow Haskell Compiler (GHC) (Section 1.2.5). Our plan of attack is that of [WPJ99, §2.3]: we perform an initial usage type inference soon after translation into Core; subsequent transformations preserve usage type soundness, but at any point we may choose to perform another inference to improve the accuracy of the types (recall that a decidable inference of any interesting operational property is necessarily an approximation). Finally, we perform an inference just prior to translation to STG, to ensure maximally-accurate usage information is available when deciding which code to generate for each thunk.

Usage information is used in two places. Firstly, it is used by the code generator in making the decision as to whether or not to generate an updatable thunk. Secondly, it is used by the optimisation passes: the usage analysis informs the optimiser about one-shot lambdas (Section 1.3.4) in order to increase the scope for code-floating transformations such as inlining and full laziness.

We have performed detailed measurements of the results of adding our analysis to the optimising compiler GHC. All standard libraries were compiled with the analysis, in addition to the program under test. For each program, the change in total bytes allocated and in run time was measured relative to the version of the compiler and libraries without usage inference. We also computed the percentage of thunks actually demanded at most once during execution that were statically identified as such by the analysis (the *effectiveness*).

Table 4.1 shows the effectiveness of the usage analysis and the improvement in run time for a representative selection of programs in the test suite. The results indicate that the simple polymorphic analysis is successful in solving the problems we identified with the monomorphic analysis. Library functions such as those in Figure 4.1 are given good types, and a significant fraction of used-once thunks are identified by the analysis. This measurably improves the run time of the programs tested. The results are presented and discussed in more detail in Section 6.8.

The comparative results of Section 6.8.2, however, clearly show that appropriate treatment of data structures is very important to the effectiveness of the analysis, and it is to this that we turn in the next chapter.

4.8 Related work

In this section we consider the large body of work on constrained polymorphism, the approach we rejected (Section 4.8.1); Nordlander’s work on pragmatic subtyping (Section 4.8.2), which uses a different approach to generate what are essentially simple-polymorphic types; and related work making use of annotation polymorphism

Table 4.1 Run time improvement and effectiveness of usage analysis.

Results of usage analysis					
Program	Run time (seconds)			Effectiveness	
	None	Usage		None	Usage
spectral/boyer	11.49	11.45	(−0.35%)	0.00%	0.00%
real/bspt	10.22	10.03	(−1.86%)	0.00%	5.60%
real/cacheprof	1.43	1.44	(+0.70%)	0.00%	4.68%
spectral/clausify	16.41	16.22	(−1.16%)	0.00%	14.24%
spectral/cryptarithm2	11.01	10.46	(−5.00%)	0.00%	93.13%
spectral/fft2	16.45	14.71	(−10.58%)	0.00%	71.48%
real/gamteb	9.22	9.13	(−0.98%)	0.00%	0.00%
imaginary/integrate	6.98	6.48	(−7.16%)	0.00%	43.54%
real/lift	0	0	—	0.00%	2.67%
spectral/mandel	15.36	14.15	(−7.88%)	0.00%	1.25%
spectral/multiplier	15.18	14.85	(−2.17%)	0.00%	22.66%
spectral/puzzle	16.69	17.08	(+2.34%)	0.00%	0.00%
imaginary/queens	14.32	14.32	(+0.00%)	0.00%	0.00%
real/reptile	0.02	0.01	—	0.00%	1.47%
spectral/simple	13.8	14.76	(+6.96%)	0.00%	42.37%
<i>Geometric mean:</i>			(−2.19%)		

(Section 4.8.3). In more depth, we attempt to relate our system to the HM(X) system of [SMZ99] (Section 4.8.4).

4.8.1 Constrained polymorphism

There has been a large amount of research on subtyping and polymorphism. A number of researchers have worked on the problem of simplifying subtyping constraints, since without some simplification effort, the constraints resulting from straightforward type inference become unmanageably large (both uninterpretable for the user, and slow to process for the machine). However, few have attempted to *approximate* constraints: the simplification that is done is usually meaning-preserving. Below we summarise work in the area.

4.8.1.1 Polymorphism and subtyping

The canonical polymorphic lambda calculi are of course the Girard–Reynolds polymorphic lambda calculus, System F [Gir72, Rey74], and its higher-order extension, F^ω [Gir72]. Extensions of these systems to include subtyping are F_\leq (“F-sub”) [CG92, CMMS94] and F_\leq^ω [Car88, SP94], respectively. These extensions use bounded quantification, as introduced by [CW85], to correctly deal with the behaviour of functions in the presence of subsumption. Types in these systems are of the form $\forall \alpha \leq \tau_1 . \tau$, where τ_1 places an *upper bound* on the types with which α may be instantiated.

Amadio and Cardelli [AC91, AC93] give an early survey of the problem of subtyping and recursive types, and discuss the meaning of subtyping, subtyping rules, type equivalence, canonical forms, and models. They give an algorithm for deciding $\tau_1 \leq \tau_2$, based on regular (but possibly infinite) trees. This can be seen as equivalent to a coinductive definition of subtyping, as observed by Pierce and Sangiorgi [PS93]; this approach was worked out in detail by Brandt and Henglein [BH98]. An excellent and accessible overview of the current state of the art is given by Gapeyev, Levin, and Pierce in [GLP00].

Two forms of subtyping are possible: *structural* subtyping [Car88], in which the subtype relation holds only of types that have the same structure, and may be decomposed into the conjunction of many applications of a *primitive* or *atomic* relation on basic types, and *non-structural* subtyping [JP99], in which the subtype relation need not follow the structure of the types, and the subtype ordering has a least element \perp and a greatest element \top . Subtyping in the present thesis is structural, and this greatly simplifies our algorithms, although it is not clear in general which form of subtyping is harder [JP99, HR98].

Because of the formal equivalence between type-based analysis and abstract interpretation (Section 1.4.1), polymorphism has a corresponding concept in flow analysis, namely *polyvariance* (or *splitting*) [Bul84, JW95, WJ98].

4.8.1.2 Type inference and constraints

ML-like languages traditionally perform type inference according to Damas–Hindley–Milner’s Algorithm \mathcal{W} (see Section 5.1.1). This infers polymorphic type *schemes* for let-bound identifiers (by means of an operation called *generalisation*), and monomorphic types elsewhere. The algorithm depends on unification [Rob65, MM82], which is invoked at application nodes $(e_1 \ e_2)$ to require that the type of e_2 is equal to the argument type of e_1 . In the presence of a subsumption rule, however, unification can no longer be used. At an application node $(e_1 \ e_2)$, the requirement is now that the type of e_2 be a *subtype* of the argument type of e_1 . This is naturally expressed as a *constraint* on the relevant types as first observed by Wand [Wan87], and type inference becomes a problem in constraint solution: one collects constraints on free type variables and finds the least solution satisfying the constraints. From here it is only a small step to *constrained polymorphism* [Cur90, Mit84, AW93, Mit91, TS96], where we allow arbitrary constraints on quantifiers rather than merely an upper bound: $\forall \alpha : C . \tau$.²³

4.8.1.3 Constraint solution

Unfortunately, constraint solution is a much harder problem in general than unification; exactly how hard depends on various parameters of the problem [LM92], but it may be as hard as nondeterministic exponential time; [Tiu97] gives a **P**TIME algorithm in one case, and [Fre97] shows that constraint solution in the structural case is **PSPACE**-complete. On the other hand, non-structural subtyping may be easier:

²³In the presence of recursive types, the two forms are in fact equivalent [PS96].

[LM92] claims linear in a very restricted case, [Hen99] gives an $O(n^2)$ algorithm for a special case derived from object-oriented programming, and [JP99] gives an $O(n^3)$ general algorithm. One may also consider the difficulty of deciding the subtype relation $C \vdash \tau_1 \leq \tau_2$ or constraint entailment $C \vdash C'$ [LM92, TS96, HR98].

4.8.1.4 Constraint simplification

In an attempt to deal with this, practical algorithms must perform *simplification* on constraint sets, as well as using clever internal representations. As explained in [AWP97], these simplifications are useful from the point of view both of the user (simpler types are easier to read, write, and understand) and of program analyses (simpler types make the relevant properties clearer, and take less time to process).

The earliest work on constraint simplification is that of Fuh and Mishra [FM90]. A good recent overview is given by Pottier in [Pot01].

The definitive work in the field is that of Trifonov and Smith [TS96], which gives an *observational* characterisation of polymorphic subtyping, proves it equivalent to a semantic one that generalises the regular trees of [AC93], and gives an efficient algorithm for a powerful decidable approximation to it. They give a general representation $\forall \vec{\alpha} . \tau \setminus C$ for constrained types that subsumes earlier representations; these representations are *closed*, which is convenient for the technical development. The observations on a constraint are the possible future constraints that may occur; types in the context may acquire only new lower bounds, and root types may acquire only new upper bounds. This leads to the notion of the *polarity* of a variable in a constraint, one which is useful in garbage-collecting unreachable constraints. The algorithm makes use of an efficient constraint set representation, the *kernel*. The algorithm subsumes Damas–Hindley–Milner type scheme instantiation, and subtyping and prenex-form union types in [AC93], and enables the soundness of various constraint garbage-collection algorithms [EST95] to be proven.

Pottier’s work [Pot01, Pot98, Pot00] is based on that of [TS96], and develops a polymorphic, subtyping type inference engine in detail. Constraints are reduced to atomic constraints, and garbage collection is performed of unreachable constraints according to polarities. Simplification is then performed by means of an algorithm based on Hopcroft’s finite state automaton minimisation algorithm [Hop71] (see also [HU79, §3.4], but this algorithm may be less efficient).

In our system, the notion of observability must be different from that of [TS96]. Internal variables, that is, those that are not mentioned in either type or context, may still be significant if they annotate a term. This is also the case for Flanagan and Felleisen’s labels annotating program points of interest [FF99], and this explains their use of a covariant goodness ordering (see Section 3.4.3) as well as Pottier’s criticism of it [Pot01, §6]. Such variables must not be simplified away, and should be treated as positive when searching for a least solution.

Flanagan and Felleisen [FF99] discuss elegantly and thoroughly the problem of performing (in practice) set-based analysis of Scheme programs. The constraint systems generated get very large (linear or quadratic in the size of the program, and solution is $kn^3 + O(n^2)$, small k but significant for 1000-plus line programs), seriously affecting performance, and so they discuss techniques for “approximate sim-

plification” – efficiently yielding simpler but not necessarily optimal constraint sets (optimal is **PSPACE**-hard), not to be confused with constraint approximation.

They give four algorithms for simplifying constraint sets, three of which are based on regular tree grammars (RTGs), in increasing order of difficulty: delete empty constraints, delete unreachable constraints, delete ϵ -constraints, and Hopcroft’s [Hop71] algorithm. Benchmarks are given showing their effectiveness (very significant) in practice. They discuss Trifonov and Scott, and Pottier, noting that they subsume their *ad hoc* simplifications.

They briefly consider conservative approximation of constraint sets, but by automatic entailment checking of human-supplied constraints, rather than automatic inference of such.

[MW97] use cycle elimination as well as other standard procedures on their constraint sets, which are represented as a *transitive kernel* D of C , where each variable in D has a set of upper and lower bounds, and D is the minimal set such that its transitive closure is C (this is a similar representation to that of [TS96]’s kernel, except that Trifonov retains the transitively-generated constraints also).

4.8.1.5 Constraint approximation

The approaches so far have all performed constraint *simplification* while preserving the observational meaning of the constraints. A few people have considered constraint *approximation*, where information is deliberately lost, preserving soundness but not completeness. Flanagan and Felleisen [FF99] consider allowing the programmer to supply approximate constraints for a type and verifying that they entail the machine-inferred constraint. Nordlander’s eagerly-approximating subtyping algorithm is considered in Section 4.8.2. Cardelli’s greedy algorithm for type-checking F_{\leq} [Car93] resolves all subtyping constraints immediately on generation (*i.e.*, at application sites), performing unification on type variables as necessary to ensure that constraints need never be propagated. Pierce and Turner [PT97] also infer unbounded-polymorphic types, although they discuss the extension to bounded polymorphism. Their approximation makes use of an unusual operation to remove particular undesirable free variables from a type, which simply promotes the type upwards until the variables disappear. This is used to avoid variables leaving their scopes. For example, $\forall Y. () \rightarrow (Y \rightarrow Y) \leq \forall Y. () \rightarrow X$ should *not* yield $Y \rightarrow Y \leq X$, but $\perp \rightarrow \top \leq X$. Obviously they work with non-structural subtyping. Amtoft [Amt94] converts strictness constraints into a normal form by means of an approximation that alters the solution set but preserves the minimal (*i.e.*, best) solution. The HM(X) system of Odersky, Sulzmann, and Wehr [OSW98], with its cylindrical constraint systems, provides an elegant setting for considering constraint approximation, and we do this in Section 4.8.4 below.

4.8.1.6 New approaches

Some recent work has addressed the problem of large constraints more directly. Rather than merely simplifying constraints, the new approach is to invent much more efficient ways of storing the constraint set, which avoid the usual exponential

blow-up in constraint size by performing more sharing. Algorithms over these efficient representations themselves become more efficient, since less work is repeated. This is reminiscent of unification in standard ML type inference, which may yield types that are exponential in size when displayed even though the internal shared digraph representation is only linear [Tiu97].

Rehof and Fähndrich [RF01] avoid the potentially-exponential blow-up of the size of the constraint set as program size increase by means of *instantiation constraints*. Instead of making a fresh copy of the constraint inferred for a function at each application site, their analysis simply stores a substitution from which the constraint may be derived. They then show that reachability techniques for context-free languages can be applied to the resulting structure, allowing the constraints to be solved dramatically faster: $O(n^3)$ in the size of the typed program, as opposed to the $O(n^8)$ previous best-known algorithm for the same problem.

Gustavsson and Svenningsson [GS00b] similarly avoid increasing the size of the constraint set by means of *constraint abstractions*, presented in a companion paper [GS01b]. Their analysis builds a *constraint abstraction* for the constraint inferred for each function, and applies it to its actual parameters at each application site. The size of these applications is bounded by the size of the type of the binder, unlike the substitution instances they replace which may be quadratic in size or worse. In the companion paper they present a constraint solver that runs in time $O(n^3)$ in the number of variables.

Each of these techniques appears to be a very promising route to avoiding the approximation inherent in our simple polymorphism while still running in reasonable time.

4.8.2 Pragmatic subtyping

Nordlander [Nor98, Nor99, Nor02] describes a *pragmatic* approach to polymorphic subtyping, arising from his implementation of the object/functional language O'Haskell. He discusses the practical difficulties of *complete* inference algorithms for polymorphic subtyping, and identifies an “inherent conflict... that is not present in the original ML type system.

“While the principal type of an ML expression is also the syntactically shortest type, the existence of subtype constraints in polymorphic subtyping generally makes a principal type *longer* than its instances.” [Nor98, §1]

He then presents an algorithm that (if it succeeds) *always infers types without subtype constraints* – exactly what we have called *simple polymorphism*! This is

“a particularly interesting compromise... since such types possess the ML property of being syntactically shortest among their instances, even though they might not be most general. We might say that the algorithm favours readability over generality.”

Nordlander's system permits the programmer to annotate her program explicitly in cases where types with constraints are actually required.

However, Nordlander’s algorithms are not applicable to the problem of usage polymorphism. He works in a setting based on [Hen96], equipped with an extensible, partially-ordered subtype relation which may have significantly more structure than our lifted annotation ordering. More importantly, his system is intended for use in the context of type inference for object-oriented languages, which turns out to be quite different from usage inference: a central assumption is that “[we] support subtyping only when the types involved are known.” [Nor98, §2]. Based on this assumption, the algorithm approximates constraints $\alpha \leq \beta$ between unknowns by *equality* constraints, unifying the variables. While in OO programming the type of either object or method is likely to be known, and functions like *twice* (Figure 4.11) are infrequent, we have identified the central contribution of polymorphism as the ability to express *dependency* constraints between input and output (Section 4.1.3), exactly the constraints Nordlander unifies away. (In our system, undetermined variables are only unified if they lie within a generalisable cluster; variables lying above or below the cluster are forced to 1 or ω respectively). In our setting, the only known annotation is ω , and it seems likely that Nordlander’s algorithm would assign this to most variables (consider his solver rule (D)), reducing the system to the monomorphic system of Chapter 3.

The system is an interesting modification to the standard efficient-unification algorithm of Martelli and Montanari [MM82], treating a constraint of a variable by a variable as a unification, but a constraint of a variable by a set of terms as a least upper or greatest lower bound operation. This approximating, eager constraint solver is then used in a modification of Milner’s Algorithm \mathcal{W} (see Section 3.9.4). As usual, types of let bindings are generalised and types of variables are instantiated (both at simple-polymorphic types), but in addition the constraint solver is invoked whenever the scope of a variable is exited, namely outside lambda abstractions and let *bodies*. This eager invocation is necessary, since if the constraint set ever gets too large the approximations may lead to failure (an example is given where three constraints is too many). It also ensures that the *non-approximating* closure operation is effective: a type variable may only be generalised if it does not occur in the constraint!²⁴ To obtain better results, Nordlander treats curried abstraction and application as uncurried, with a special rule that considers all the arguments simultaneously (as do [PT97]).

4.8.3 Annotation polymorphism

The idea of usage polymorphism itself is in no way new. It is proposed in the paper that started us off, [TWM95a], although this proposal differs from our polymorphism in that our system provides a usage-polymorphic type for a *single copy* of a function, rather than generating multiple specialised copies or variants (but see Section 6.3.3), or passing usage arguments at runtime. A similar notion of annota-

²⁴Actually, it is not surprising that generalisation can only be performed if the relevant variable is unconstrained; this is in the nature of simple-polymorphic types. However, in our system the closure operation selects the variables to generalise and then performs the required approximation to allow the selected generalisation, whereas Nordlander performs approximation elsewhere, and in closure generalises only those variables already generalisable.

Figure 4.13 Logical type rules for bounded usage quantification.

$$\begin{array}{c}
\frac{}{C; \Gamma, x : \sigma \vdash_b x : \sigma} (\vdash_b\text{-VAR}) \quad \frac{}{C; \Gamma \vdash_b n : \text{Int}^\omega} (\vdash_b\text{-LIT}) \\
\\
\frac{C; \Gamma \vdash_b e : \text{Int}^1 \quad C; \Gamma \vdash_b e_i : \sigma \quad i = 1, 2}{C; \Gamma \vdash_b \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma} (\vdash_b\text{-IF0}) \\
\\
\frac{C; \Gamma \vdash_b e_i : \text{Int}^1 \quad i = 1, 2}{C; \Gamma \vdash_b e_1 + e_2 : \text{Int}^\omega} (\vdash_b\text{-PRIMOP}) \quad \frac{C; \Gamma \vdash_b e : \text{Int}^1}{C; \Gamma \vdash_b \text{add}_n e : \text{Int}^\omega} (\vdash_b\text{-PRIMOP-R}) \\
\\
\frac{
\begin{array}{l}
C; \Gamma, x : \sigma_1 \vdash_b e : \sigma_2 \\
C \vdash^e \{ \text{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \} \\
C \vdash^e \{ \text{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \geq \kappa \} \quad \text{for all } y \in \Gamma
\end{array}
}{C; \Gamma \vdash_b \lambda^\kappa x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_b\text{-ABS}) \\
\\
\frac{C; \Gamma \vdash_b e_1 : (\sigma_1 \rightarrow \sigma_2)^1 \quad C; \Gamma \vdash_b e_2 : \sigma_1}{C; \Gamma \vdash_b e_1 e_2 : \sigma_2} (\vdash_b\text{-APP}) \\
\\
\frac{
\begin{array}{l}
C; \Gamma, \overline{x_j} : \overline{\sigma_j} \vdash_b e_i : \sigma_i \quad \text{for all } i \\
C; \Gamma, \overline{x_j} : \overline{\sigma_j} \vdash_b e : \sigma \\
C \vdash^e \{ \text{occur}(x_i, e) + \sum_{j=1}^n \text{occur}(x_i, e_j) > 1 \Rightarrow |\sigma_i| = \omega \} \quad \text{for all } i
\end{array}
}{C; \Gamma \vdash_b \text{letrec } \overline{x_i} : \overline{\sigma_i} \equiv \overline{e_i} \text{ in } e : \sigma} (\vdash_b\text{-LETREC}) \\
\\
\frac{C; \Gamma \vdash_b e : \sigma' \quad C \vdash^e \{ \sigma' \preceq \sigma \}}{C; \Gamma \vdash_b e : \sigma} (\vdash_b\text{-SUB}) \\
\\
\frac{C \wedge D; \Gamma, \overline{u_i} \vdash_b e : \tau^\kappa \quad \overline{u_i} \notin \text{fu}v(C, \Gamma, \kappa)}{C \wedge \exists \overline{u_i} . D; \Gamma \vdash_b \Lambda \overline{u_i} : D . e : (\forall \overline{u_i} : D . \tau)^\kappa} (\vdash_b\text{-UABS}) \\
\\
\frac{C; \Gamma \vdash_b e : (\forall \overline{u_i} : D . \tau)^\kappa \quad C \vdash^e \exists \overline{u_i} . D}{C \wedge D; \Gamma \vdash_b e \overline{u_i} : \tau^\kappa} (\vdash_b\text{-UAPP}) \\
\\
\frac{C; \Gamma \vdash_b e : \sigma \quad \overline{u_i} \notin \text{fu}v(\Gamma, e, \sigma)}{\exists \overline{u_i} . C; \Gamma \vdash_b e : \sigma} (\vdash_b\text{-HIDE})
\end{array}$$

tion polymorphism is also familiar in the flow analysis community, under the term *polyvariance* [Bul84, JW95, WJ98]; many flow analyses are abstract interpretations rather than type-based analyses, but it has been shown that the two are closely related [Jen91]. Polyvariance has been applied to many different analyses, notably here binding-time analysis [DHM95], and so its application to usage is unsurprising.

Christian Mossin’s work with Dussart and Henglein [DHM95, Mos93, HM94] relates to a two-point binding-time lattice ($S < D$) of annotations of lambda-calculus expressions. The system has subtyping (with explicit coercions), (bounded) annotation polymorphism, and constraints. A slightly unusual formulation is used for polymorphism and bounds (abstracting over each separately), but we believe it is equivalent to the standard presentation. Polymorphism occurs at let-nodes as usual, but *Kleene–Mycroft iteration* [DHM95, §§3.4,3.5,4] is used to decidably infer polymorphic recursion (possible due to the finite nature of the lattice).²⁵

Similarly, the Clean uniqueness typing system of Barendsen *et al.* [BS96] features constrained uniqueness polymorphism for lambda-lifted functions and data constructors, and polymorphic recursion using Kleene–Mycroft iteration [PvE98, p. 5].

Annotation, or “property”, polymorphism is considered in [GSSS01], where a generic implementation by reduction to Boolean constraints is proposed.²⁶

4.8.4 Constraint approximation in HM(X)

Rather than approaching the simple-polymorphic type system directly, it might be fruitful to consider a more general constrained-polymorphic type system, with the simple-polymorphic types as a subset. We considered this, basing our work on the HM(X) system of [SMZ99] (revised from [OSW98]). In this setting, the restriction to simple polymorphism means simply that all constraints appearing on abstracted variables must be trivial; this may be achieved by explicit approximation.

Figure 4.13 presents the well-typing rules for such a system, with explicit constraints and constrained quantification. The term and type languages are a slight extension of LIX_2 ; we have simply added constraints to quantifiers and usage abstractions. The rules define a relation $C; \Gamma \vdash_b e : \sigma$, where C is a constraint set. Note that as usual we identify types up to α -equivalence; for example in $(\vdash_b\text{-UAPP})$ we would normally choose \overline{u}_i to be fresh with respect to C . The constraint entailment relation $\cdot \vdash^e \cdot$ appearing in the rules is defined in Section 3.5.1, and other constraint operations in the appendix, Section D.3.

The usage abstraction and usage application rules are of particular interest. Rule $(\vdash_b\text{-UABS})$ abstracts over a vector of usage variables \overline{u}_i , preserving the relevant constraints D in the bound on the quantifier. Note the use here of *explicit* usage abstraction, just as in LIX_2 (Section 4.2.2). Rule $(\vdash_b\text{-UAPP})$ instantiates the vector of usage variables, adding the constraints from the bound into the constraint set of the

²⁵Polymorphic recursion is used in the function $f\ x\ y = \text{if } x == 0 \text{ then } 1 \text{ else } f\ y\ (x - 1)$ where the recursive call switches the arguments. In this case, the binding times of the arguments must be switched also.

²⁶Note that the assertion made by Glynn *et al.* that source (type) polymorphism and annotation (usage) polymorphism do not coexist in our work is false, as demonstrated by Chapter 5. Type polymorphism was elided from [WPJ00] purely in order to clarify the presentation.

consequent. Again a side condition ensures satisfiability.

An important contribution of the work of [OSW98, SMZ99] is the use of the hiding operator $\exists u . C$ (i.e., of cylindrical constraint systems) apparent in rule $(\vdash_b\text{-UABS})$: the existentially-quantified copy of D preserved in the consequent of the rule is necessary to ensure the satisfiability of D and thus the well-typedness of e even in cases where the abstraction is never applied. Trifonov and Smith’s definitive paper [TS96] (Section 4.8.1.4) does not use such an operator, and others such as Jim and Palsberg [JP99] use an *ad hoc* approach, maintaining a set of “fresh variables” which do not participate in entailment checks.

While the rules of Figure 4.13 are largely based on those of $\text{HM}(X)$, there are some significant changes.²⁷ Firstly, variables u in the present system do not scope over types, but only annotations, and so the canonical forms of [SMZ99] are not applicable; secondly, our system alters not just the constraint system X , but also the type rules, adding extra requirements on C in variable-binding rules (i.e., $(\vdash_b\text{-ABS})$ and $(\vdash_b\text{-LETREC})$). The \exists -introduction rule $(\vdash_b\text{-HIDE})$ ’s side condition reflects the fact that terms may now contain free variables u . As in the $\text{HM}(X)$ rules, an implicit side condition to each rule requires that all constraints appearing are satisfiable. Without this condition, a type derivation for a term would not necessarily imply the existence of a solution to the constraints, and nonsensical typings would be admitted for many unsound terms.²⁸

We now introduce a notion of approximation. An *approximation* to a constraint set C is a constraint set D such that $D \vdash^e C$; that is, a solution to the approximation is a solution to the original constraint set. The implicit side condition that constraint sets be satisfiable guarantees that there is still a solution.

Clearly we may already substitute (simplified but) equal constraint sets without loss of accuracy, as follows (recall $C =^e D$ iff $C \vdash^e D$ and $D \vdash^e C$):

$$\frac{C; \Gamma \vdash_b e : \sigma \quad C =^e D}{D; \Gamma \vdash_b e : \sigma} (\vdash_b\text{-EQUAL})$$

Further, we allow the removal of unnecessary variables, again without loss of accuracy, as follows:

$$\frac{C; \Gamma \vdash_b e : \sigma \quad C \vdash^e \langle u = \kappa \rangle}{\exists u . C; \Gamma[\kappa/u] \vdash_b e[\kappa/u] : \sigma[\kappa/u]} (\vdash_b\text{-MERGE})$$

Finally, if we wish to allow constraint-set approximation, possibly losing accuracy but not soundness, we may add the rule:

$$\frac{C; \Gamma \vdash_b e : \sigma \quad D \vdash^e C}{D; \Gamma \vdash_b e : \sigma} (\vdash_b\text{-APPROX})$$

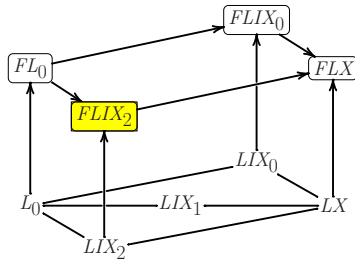
²⁷Martin Sulzmann [personal communication, 9 January, 2002] believes that this system is able to fit within the $\text{HM}(X)$ framework.

²⁸A further change is that \forall -introduction and elimination now have a (target) term representation. The rules are still “logical”, however, from the point of view of *source* terms, and thus we must still formulate a syntax-directed form of these rules for practical use.

The simple-polymorphic usage inference problem may now be seen as follows. Given M , find an annotation e typeable in this system (*i.e.*, e such that $(e)^{\sharp} = M$ and $C; \emptyset \vdash_b e : \sigma$) but such that all bounds D occurring in e are trivial; choose the best annotation of these in some appropriate sense (see Section 4.5).

The key issue is the restriction to trivial bounds. Consider the (\blacktriangleright_2 -LETREC) inference rule of Section 4.4.2. We must generalise each term M_i using (\vdash_b -UABS). If D in this rule is trivial, then the existential portion of the constraint set resulting from (\vdash_b -UABS) and the bound on the quantifiers both disappear, leaving an unbounded rule identical to (\vdash_2 -UABS) (Figure 4.4). This may be achieved in the same manner as the algorithm of Section 4.5.4, by using rule (\vdash_b -APPROX) to equate certain variables or fix them to 1 or ω , and then performing the appropriate substitutions in τ_i and e using rule (\vdash_b -MERGE).

Chapter 5.



Covering the Full Language

In the present chapter we extend our usage analysis to handle type polymorphism and user-defined algebraic data types. These essential features were omitted earlier in order to simplify the presentation; adding them justifies our claim that we deal with usage analysis for *real*, full-featured functional languages.

The exposition of this full analysis largely follows the structure of previous chapters. We begin in Section 5.1 by presenting the new features of the full source language FL_0 . We extend the target language $FLIX_2$ with the new features in the following two sections: Section 5.2 discusses usage typing for type polymorphism, which turns out to be relatively straightforward; Section 5.3 addresses the more difficult problem of usage typing for user-defined algebraic data types. The system we devise is parameterised over the choice of annotation scheme for data type declarations, and so we discuss choosing an annotation scheme separately in Section 5.4. The extended usage inference is presented in Section 5.5; it is essentially the same as that of Section 4.4. The required results are proven in Section 5.6, and we consider related work in Section 5.7.

Discussion of a number of implementation-specific language issues is deferred to Chapter 6.

5.1 The full source language

Chapter 2 described the simplified source language L_0 . We begin this chapter by extending this language with the new features, yielding FL_0 , the *full* source language. We also give an operational semantics for this language.

5.1.1 Type polymorphism

The first new language feature is *type polymorphism*. (Polymorphism was first named by Strachey [Str67, §3.6]. Algorithm \mathcal{W} , which infers polymorphic type schemes, is due to Milner [Mil78], along with the famous soundness theorem stating that “well-typed programs do not go wrong”. Damas [DM82] introduced the modern notation for type schemes and showed that Algorithm \mathcal{W} obtains *principal* type schemes, although the equivalent result had already been obtained by Hindley [Hin69] for a calculus without *let*. A tutorial is [Car87]. Lee and Yi [LY98] formally define an equivalent “folklore” Algorithm \mathcal{M} that has better error behaviour.) The source language we have been working with so far is *monomorphic*; for example, we can write the two functions

$$\begin{aligned} app_1 : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int} &= \lambda f : \text{Int} \rightarrow \text{Int} . \lambda x : \text{Int} . f\ x \\ app_2 : (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) &= \lambda f : \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) . \lambda x : \text{Int} . f\ x \end{aligned}$$

but we cannot write the general application function that takes any function and its argument and applies the one to the other. We would like to write this function and give it the *polymorphic* type $\forall \alpha, \beta . (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$.

Languages in the Hindley–Milner tradition, such as ML [MTHM97] and Haskell [PJH⁺99], perform this generalisation implicitly. That is, the programmer can write simply $\lambda f . \lambda x . f\ x$ without annotation and the language will infer the correct polymorphic type automatically. Similarly, when the function is used, the general (polymorphic) type is implicitly instantiated with the correct type parameters.

However, in a typed intermediate language it becomes necessary to record these steps explicitly. As Peyton Jones describes in [PJS98a, §3.2] this can be done most effectively using the Girard–Reynolds polymorphic lambda calculus (*a.k.a.* System F) [Gir72, Rey74]. In this calculus, the expression M is generalised with respect to the type variable α by a *type abstraction* $\Lambda \alpha . M$, and instantiated with type t by a *type application* $M\ t$. This is the technique used by the typed intermediate languages we address, and we must therefore discover how to deal with this appropriately in our usage analysis.

5.1.2 User-defined algebraic data types

The second new language feature, another crucial feature of modern functional languages, is the ability to define new *algebraic data types*. (User-defined algebraic data types were suggested by Landin in [Lan64], and their initial algebra semantics discussed in [GTW78] and many related papers. They first appeared in the language Hope [BMS80], and have subsequently become central to modern functional languages such as ML and Haskell.) A language provides a selection of primitive base types and allows more complex and structured types to be built up from them. These user-defined algebraic data types are defined recursively and expressed as tagged sums of products. A datum is constructed by applying a tag, or *constructor*, to a vector of elements of the appropriate types. Such data are taken apart (deconstructed) using a case expression, which selects an expression based on the tag and binds its variables to the component elements of the datum.

Figure 5.1 The full source language FL_0 (cf. Figure 2.1).

Terms	$M ::=$	A $ $ n $ $ $K_i \overline{t_k} \overline{A_j}$ $ $ $\lambda x : t . M$ $ $ $M A$ $ $ $\Lambda \alpha . M$ $ $ $M t$ $ $ $\text{case } M : T \overline{t_k} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow \overline{M_i}}$ $ $ $M_1 + M_2$ $ $ $\text{if0 } M \text{ then } M_1 \text{ else } M_2$ $ $ $\text{letrec } \overline{x_i : t_i = \overline{M_i}} \text{ in } M$	atom literal (integer) constructor term abstraction term application type abstraction type application case expression primop (addition) zero-test conditional recursive let binding
Atoms	$A ::=$	x $ $ $A t$	term variable atom type application
t -types	$t ::=$	$t_1 \rightarrow t_2$ $ $ Int $ $ $T \overline{t_k}$ $ $ $\forall \alpha . t$ $ $ α	function type primitive type (integers) algebraic data type type-generalised type type variable
Decls	$T :$	$\text{data } T \overline{\alpha_k} = \overline{K_i \overline{t_{ij}}}$	algebraic data type declaration

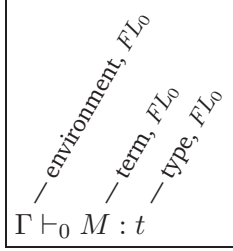
5.1.3 Language extensions

Figure 5.1 presents our full source language FL_0 , including these features. The well-typing rules are given in Figure 5.2, and should be unsurprising. They extend the language and well-typing rules of Chapter 2. Here and elsewhere, **lowlighted text** denotes material that is retained unchanged from an earlier presentation, and **highlighted text** denotes material newly introduced or altered. The full set of rules appears in Appendix B. *Type polymorphism* is represented by term forms $\Lambda \alpha . M$ (denoting abstraction of type variable α from expression M) and $M t$ (denoting application of expression M to type t), and by type forms $\forall \alpha . t$ (denoting the type t generalised over free type variable α) and α (denoting a type variable). We consider terms and types up to α -equivalence of type variables as well as term variables.

Algebraic data types are declared using the declaration syntax

$$\text{data } T \overline{\alpha_k} = \overline{K_i \overline{t_{ij}}}$$

which defines a data type with type constructor T , formal type parameters $\overline{\alpha_k}$, and constructors $\overline{K_i}$. The data type is a tagged sum of products; each constructor K_i tags the product $\prod_j t_{ij}$, where the $\overline{t_{ij}}$ may have free type variables in $\overline{\alpha_k}$. Constructors are unique within a program, and so to each data constructor K_i corresponds a unique

Figure 5.2 Well-typing rules for the full source language FL_0 (extends Figure 2.2).

All the rules for L_0 (q.v. Figure 2.2),
with the addition of:

$$\frac{\Gamma, \alpha \vdash_0 M : t \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash_0 \Lambda \alpha . M : \forall \alpha . t} (\vdash_0\text{-TYPABS}) \quad \frac{\Gamma \vdash_0 M : \forall \alpha . t_1}{\Gamma \vdash_0 M t_2 : t_1[t_2/\alpha]} (\vdash_0\text{-TYPAPP})$$

$$\frac{\begin{array}{l} t_{ij}^\circ = t_{ij}[\bar{t}_k/\bar{\alpha}_k] \quad \text{all } j \\ \Gamma \vdash_0 A_j : t_{ij}^\circ \quad \text{all } j \\ \text{where data } T \bar{\alpha}_k = \overline{K_i t_{ij}} \end{array}}{\Gamma \vdash_0 K_i \bar{t}_k \bar{A}_j : T \bar{t}_k} (\vdash_0\text{-CON})$$

$$\frac{\begin{array}{l} \Gamma \vdash_0 M : T \bar{t}_k \\ t_{ij}^\circ = t_{ij}[\bar{t}_k/\bar{\alpha}_k] \quad \text{all } i, j \\ \Gamma, x_{ij} : t_{ij}^\circ \vdash_0 M_i : t \quad \text{all } i \\ \text{where data } T \bar{\alpha}_k = \overline{K_i t_{ij}} \end{array}}{\Gamma \vdash_0 \text{case } M : T \bar{t}_k \text{ of } \overline{K_i x_{ij}} \rightarrow M_i : t} (\vdash_0\text{-CASE})$$

data declaration and type constructor T .¹ When constructing a datum, the actual type parameters \bar{t}_k instantiating the formal parameters $\bar{\alpha}_k$ must be specified.

Destruction is performed by the case statement, which scrutinises an expression, the *scrutinee*, and chooses one of the *branches* to execute on the basis of its tag. The instantiated type of the destructed datum must be specified in an annotation to resolve potential ambiguity. In the chosen branch, the specified variables are bound to the values from the scrutinee and the expression is evaluated.² We assume that all case statements are complete, *i.e.*, that they contain a branch for every constructor of the data type. Relaxing this assumption would provide an additional way in which programs could go wrong at runtime, but would not otherwise affect the analysis.

Our constructors are *lazy* rather than strict. This means that evaluation of each argument of a constructor is delayed until its value is demanded, and that once it is

¹This could be made explicit by writing K_i^T rather than just K_i , but since T is invariably clear from context it is simpler to use the latter form.

²In earlier work [WPJ99, WPJ98] we used a form of case statement that avoided variable binding, requiring the branches to be of function type and passing them the values from the scrutinee as arguments. This was intended to simplify the proofs while complicating the statement of the typing rules. In the present work we have elected to use the conventional form of the case statement, thus making the well-typing rules simpler (the proofs are not in fact made significantly more complicated).

evaluated its value is memoised in order that subsequent demands may reuse it. This laziness is made explicit by the use of an A-normal form representation (Section 2.2), which forces all constructor arguments to be letrec-bound.

5.1.4 Operational semantics

Recall from Section 2.3.2 that since a well-typed LIX_0 program does not “go wrong”, i.e., does not have a runtime type error, we may safely erase all type information, obtaining the executable language LX . Even with type polymorphism, this remains the case. This means that type abstraction and application may be removed entirely before execution; they are purely book-keeping operations intended to preserve well-typing, and have no operational meaning at runtime. This fact has significant bearing on their correct treatment by the usage analysis, described in Section 5.2 below.

This is possible only because FL_0 ’s polymorphism is *parametric* [Str67, §3.6.4]: polymorphic expressions in FL_0 behave identically at all types. A language with a typecase construct [ACPR95, ACPP91] (Simula-67’s *Inspect* [Bir84]), which permits behaviour of a function to be conditional on the type of its argument, would in general require types to be present at runtime. In addition to permitting type-erased execution, parametricity provides the programmer with “theorems for free” [Rey83, Wad89, LP96, Cra99].

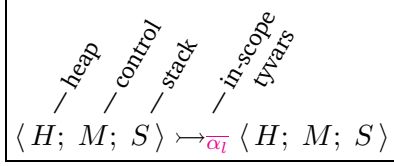
The instrumented executable language $FLIX_0$ is presented in Figure 5.3. The **instrumentation** includes types, usage annotations, and type abstractions and applications. Omitting the instrumentation yields the uninstrumented executable language FLX . The stripping and erasure functions \natural and \flat are extended in the obvious manner (Sections 2.3.1 and 2.3.2).

The operational semantics for the full language appears in Figure 5.4. The technique we use here to handle type polymorphism provides a type-erasure semantics without erasing the types, and is exactly the same as that used in Section 4.2.3 to handle usage polymorphism. The reader is referred to that section for the details. Note particularly that type applications are considered atomic, and evaluation is permitted beneath type abstractions.

Case expressions are strict in their scrutinee, as seen in the definition of shallow evaluation contexts R . Constructors take an update flag to control copying: K^\bullet denotes a constructor that must not be copied, and $K^!$ denotes one that may be copied without restriction (see Section 2.3.1). The operational semantics for constructors and case makes no reference to the data type declarations or constructor arities; $(\rightarrow_\delta\text{-CASE})$ needs only the index i of the constructor to determine which case branch to take. The function $|V|$ used by the $(\rightarrow\text{-UPDATE})$ rule is extended to inspect the update flag on constructors. The remainder of the semantics is as described in Section 2.4; a complete presentation of the combined operational semantics is deferred to the appendix, Figure B.5.

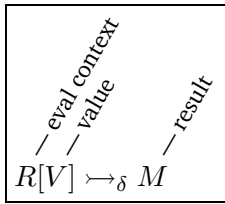
Figure 5.3 The executable language $FLIX_0$ and configurations $FLIXC_0$.
(cf. Figure 2.3).

Terms	$M ::= R[M]$ $\quad \text{letrec } x_i : \overline{t_i} =^{x_i} \overline{M_i} \text{ in } M$ $\quad \Lambda \alpha . M$ $\quad A$ $\quad V$	filled evaluation context recursive let binding type abstraction atom value
Shallow evaluation contexts	$R ::= [\cdot] A$ $\quad [\cdot] \overline{t}$ $\quad \text{case } [\cdot] : T \overline{t_k} \text{ of } \overline{K_i x_{ij} \rightarrow M_i}$ $\quad [\cdot] + M$ $\quad \text{add}_n [\cdot]$ $\quad \text{if0 } [\cdot] \text{ then } M_1 \text{ else } M_2$	term application type application case expression primop (addition) partially-saturated primop zero-test conditional
Atoms	$A ::= x$ $\quad A \overline{t}$	term variable atom type application
Values	$V ::= n$ $\quad K_i^x \overline{t_k} \overline{A_k}$ $\quad \lambda^x x : \overline{t} . M$ $\quad \Lambda \alpha . V$	literal (integer) constructor term abstraction type abstraction of value
Types	$t ::= t_1 \rightarrow t_2$ $\quad \text{Int}$ $\quad T \overline{t_k}$ $\quad \forall \alpha . t$ $\quad \alpha$	function type primitive type (integer) algebraic data type type-generalised type type variable
Update flags	$\chi ::= \bullet$ $\quad !$	not updatable/copyable updatable/copyable
Configurations	$C ::= \langle H; M; S \rangle$	where $\text{dom}(H) \not\downarrow \text{dom}(S)$
Heaps	$H ::= \emptyset$ $\quad H, x : \overline{t} =^x M$	where $x \notin \text{dom}(H)$
Stacks	$S ::= \varepsilon$ $\quad R, S$ $\quad \#x : \overline{t}, S$	where $x \notin \text{dom}(S)$

Figure 5.4 The full operational semantics (cf. Figure 2.4).

$$\begin{array}{lll}
\langle H; R[M]; S \rangle & \mapsto_{\overline{\alpha_l}} \langle H; M; R, S \rangle & (\mapsto\text{-UNWIND}) \\
\langle H; V; R, S \rangle & \mapsto_{\overline{\alpha_l}} \langle H; M; S \rangle & (\mapsto\text{-REDUCE}) \\
& \text{if } R[V] \mapsto_{\delta} M & \\
\langle H; \text{letrec } \overline{x_i : t_i =^{x_i} M_i} \text{ in } M; S \rangle & \xrightarrow{\mapsto\text{-LETREC}} & \\
& \mapsto_{\overline{\alpha_l}} \langle H, y_i : \forall \overline{\alpha_l} . t_i =^{x_i} \Lambda \overline{\alpha_l} . M_i[\phi]; M[\phi]; S \rangle & \\
& \text{where } \overline{y_i} \not\in \text{dom}(H) \cup \text{dom}(S) & \\
& \phi = [\overline{y_i} \overline{\alpha_l} / \overline{x_i}] & \\
\langle H; \Lambda \alpha . M; S \rangle & \mapsto_{\overline{\alpha_l}} \langle H'; \Lambda \alpha' . M'; S' \rangle & (\mapsto\text{-TYLAM}) \\
& \text{if } \langle H; M[\alpha'/\alpha]; S \rangle \mapsto_{\overline{\alpha_l}, \alpha'} \langle H'; M'; S' \rangle & \\
& \alpha' \text{ fresh} & \\
\langle H, x : t =^{\bullet} M; x; S \rangle & \mapsto_{\overline{\alpha_l}} \langle H; M; S \rangle & (\mapsto\text{-VAR-ONCE}) \\
\langle H, x : t =^! M; x; S \rangle & \mapsto_{\overline{\alpha_l}} \langle H; M; \#x : t, S \rangle & (\mapsto\text{-VAR-MANY}) \\
\langle H; V; \#x : t, S \rangle & \mapsto_{\overline{\alpha_l}} \langle H, x : t =^! V; V; S \rangle & (\mapsto\text{-UPDATE}) \\
& \text{where } |V| = \bullet \Rightarrow x \notin \text{fv}(H, V, S) &
\end{array}$$

$$\begin{array}{ll}
\text{where } |n| & = ! \\
|K_i^x \overline{t_k} \overline{A_j}| & = \chi \\
|\lambda^x x : t . M| & = \chi \\
|\Lambda \alpha . V| & = |V|
\end{array}$$



$$\begin{array}{lll}
(\lambda^x x : t . M) A & \mapsto_{\delta} M[A/x] & (\mapsto_{\delta}\text{-APP}) \\
(\Lambda \alpha . M) \tau & \mapsto_{\delta} M[\tau/\alpha] & (\mapsto_{\delta}\text{-TYAPP}) \\
n + M & \mapsto_{\delta} \text{add}_n M & (\mapsto_{\delta}\text{-PRIMOP-L}) \\
\text{add}_{n_1} n_2 & \mapsto_{\delta} n_3 & \text{if } n_3 = n_1 + n_2 \quad (\mapsto_{\delta}\text{-PRIMOP-R}) \\
\text{case } K_k^x \overline{A_j} \text{ of } \overline{K_i} \overline{x_{ij}} \rightarrow M_i & \xrightarrow{\mapsto_{\delta}} & \\
& \mapsto_{\delta} M_k[\overline{A_j} / \overline{x_{kj}}] & (\mapsto_{\delta}\text{-CASE}) \\
\text{if0 } n \text{ then } M_1 \text{ else } M_2 & \mapsto_{\delta} M_i & \text{if } i = (n = 0 ? 1 : 2) \quad (\mapsto_{\delta}\text{-IF0})
\end{array}$$

5.2 Accommodating type polymorphism

We now begin extending the target language LIX_2 (described in detail in Chapter 4) to the full target language $FLIX_2$. The present section adds type polymorphism to the language, and Section 5.3 adds user-defined algebraic data types.

Figure 5.5 defines the types and terms of $FLIX_2$. This figure embodies two key design decisions with respect to type polymorphism: what kind of type a generalised type should be (Section 5.2.1) and what kind of type type variables should range over (Section 5.2.2). Once these have been decided, we present the well-typing rules for type polymorphism in $FLIX_2$ (Section 5.2.3) and define subtyping over generalised types and type variables (Section 5.2.4).

5.2.1 Generalised types

In translating the type of a type abstraction $(\Lambda\alpha . e)$, we must decide both whether the type $(\forall\alpha . t)$ as a whole should be translated to a σ - or τ -type, and whether the body type t should be translated to a σ - or τ -type. Assume $e : \tau^\kappa$. There are four possibilities for this type:

		Body type	
Abstraction type	$\tau :$	$\tau :$	$\sigma :$
	$\sigma :$	(i) $(\forall\alpha . \tau)^{\kappa_1}$ (iii) $\forall\alpha . \tau$	(ii) $(\forall\alpha . \tau^{\kappa_2})^{\kappa_1}$ (iv) $\forall\alpha . \tau^{\kappa_2}$

Clearly a type abstraction must have a usage, but option (iii) has no explicit usage annotation; thus it is nonsensical (we could simply assume that the usage of a type abstraction is always ω , but this is unnecessarily coarse). We discovered in Section 5.1.4 that type abstraction and application have no operational significance (to the evaluator, e and $\Lambda\alpha . e$ look exactly the same); it follows that there should be no distinction between the usage of an expression and of its abstraction, and so $\kappa_1 = \kappa = \kappa_2$ and option (ii) is redundant.

The remaining options (i) and (iv) carry exactly the same information; the difference is purely syntactic. We choose option (i) because it fits more smoothly into the existing language, in which *usage*-generalised types $(\Lambda u . \tau)$ are τ -types, and usage annotations go outside them rather than inside (see Section 4.2.1). That is, $\forall\alpha . t$ is translated to $(\forall\alpha . \tau)^{\kappa_1}$: we add the production $\tau ::= \dots \mid \forall\alpha . \tau$.

5.2.2 Type variables

We must also independently decide: over what should type variables α (and hence type arguments) range?

Recall that the purpose of the usage typing of a function is to convey information about how that function uses its arguments. A polymorphic function uses its argument identically at any type (parametricity, Section 5.1.4), but it may certainly

choose to use it either at most once or many times. Consider the functions³

$$\begin{aligned} \text{const} &: \forall \alpha . \forall \beta . \alpha \rightarrow \beta \rightarrow \alpha = \lambda x . \lambda y . x \\ \text{const}' &: \forall \alpha . \alpha \rightarrow \alpha \rightarrow \alpha = \lambda x . \lambda y . x \end{aligned}$$

We would like to record that each of these uses its second argument not at all, and the use of its first argument is dependent on the use of its result. If type variables range over τ -types, both of these are straightforward:

$$\begin{aligned} \text{const} &: (\forall u . \forall \alpha . \forall \beta . \alpha^u \rightarrow (\beta^1 \rightarrow \alpha^u)^u)^\omega \\ \text{const}' &: (\forall u . \forall \alpha . \alpha^u \rightarrow (\alpha^1 \rightarrow \alpha^u)^u)^\omega \end{aligned}$$

However, if type variables range over σ -types we cannot express this at all. We therefore choose that type variables and type arguments range over τ -types, adding the productions $\tau ::= \dots \mid \alpha$ and $e ::= \dots \mid e \tau$.

5.2.3 Typing

The well-typing rules ($\vdash_2\text{-TYABS}$) and ($\vdash_2\text{-TYAPP}$), shown in Figure 5.6, are based on the observation that type abstraction and application have no operational significance, and thus that the usage of an expression and its type abstraction are identical. The rules simply lift and lower the usage annotation through the quantifier. We extend environments Γ to include a set of in-scope type variables in the usual way.

5.2.4 Subtyping

The subtyping relation \preccurlyeq of Figure 4.5 must be extended to accommodate the new types. The extended relation is defined in Figure 5.7.

Subtyping for generalised types and type variables is straightforward. Since $\forall \alpha . \tau$ is a binding construct, and we identify types up to α -conversion, we need only compare generalised types binding the same variable. In this simplified case, we have $\forall \alpha . \tau \preccurlyeq \forall \alpha . \tau'$ iff $\tau \preccurlyeq \tau'$. A type variable α may be instantiated to anything, and thus can only be guaranteed to be a subtype of itself: $\alpha \preccurlyeq \alpha$.

5.3 Accommodating data types

The second step in extending LIX_2 to the full language $FLIX_2$ is the addition of user-defined algebraic data types. Algebraic data types introduce additional complexity because their values, unlike primitive values, are composed of multiple other values,⁴ and the use of these values must be determined. The type system described in previous chapters is able to compute the use of a primitive value by counting syntactic occurrences and dealing appropriately with the free variables of an abstraction. These techniques apply equally well to a value of an algebraic data type, but they

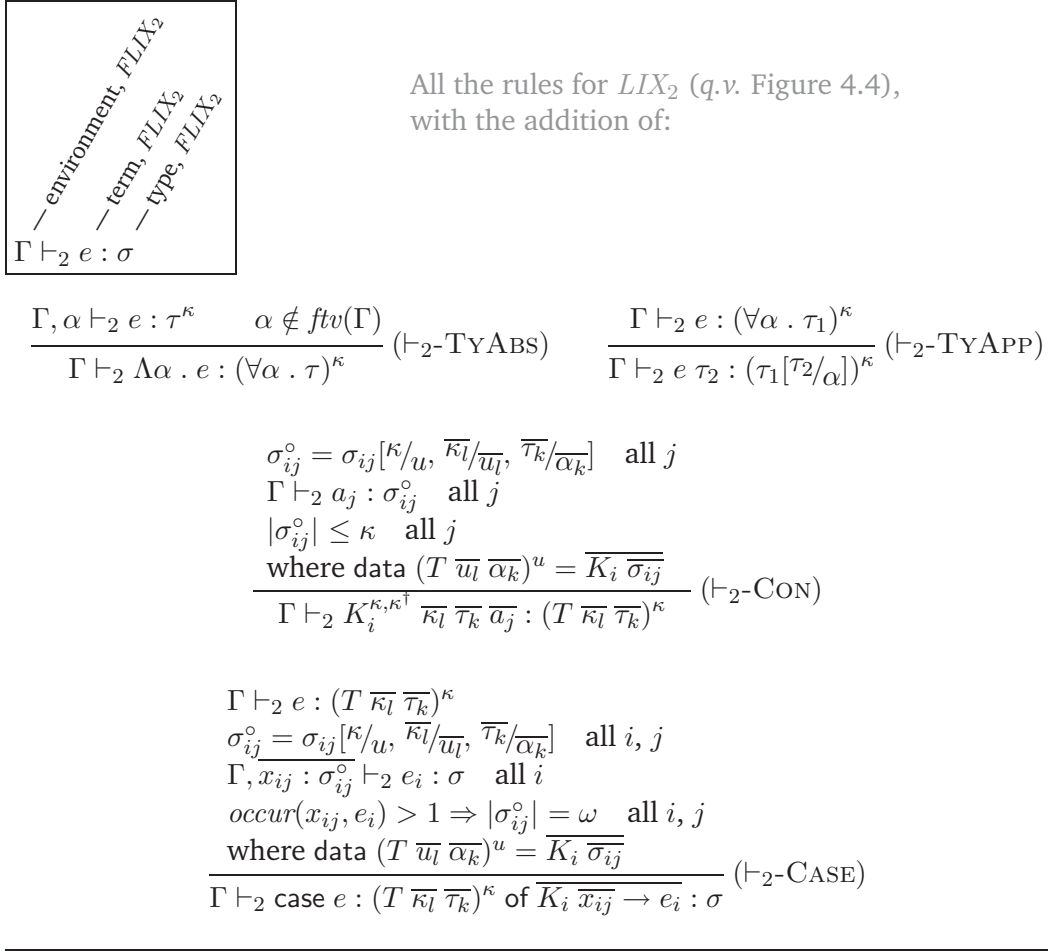
³The function const' is actually predefined in Haskell 98, under the name *astypeof*.

⁴In fact, since our constructors are lazy, an algebraic datum contains either values or *thunks* (delayed computations). We write simply “values” for brevity.

Figure 5.5 The polymorphically usage-typed language $FLIX_2$ (cf. Figure 4.2).

Terms	$e ::= a$ $ n$ $ K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j}$ $ \lambda^{\kappa, \chi} x : \sigma . e$ $ e a$ $ \Lambda \alpha . e$ $ e \tau$ $ \Lambda u . e$ $ e \kappa$ $ \text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i}$ $ e_1 + e_2$ $ \text{add}_n e$ $ \text{if0 } e \text{ then } e_1 \text{ else } e_2$ $ \text{letrec } \overline{x_i : \sigma_i} =^{\chi_i} \overline{e_i} \text{ in } e$	atom literal (integer) constructor term abstraction term application type abstraction type application usage abstraction usage application case expression primop (addition) partially-saturated primop zero-test conditional recursive let binding
Atoms	$a ::= x$ $ a \tau$ $ a \kappa$	term variable atom type application atom usage application
τ -types	$\tau ::= \sigma_1 \rightarrow \sigma_2$ $ \text{Int}$ $ T \overline{\kappa_l} \overline{\tau_k}$ $ \forall \alpha . \tau$ $ \alpha$ $ \forall u . \tau$	function type primitive type (integer) algebraic data type type-generalised type type variable usage-generalised type
σ -types	$\sigma ::= \tau^\kappa$	usage-annotated type
Decls	$T : \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}}$	data type declaration
Usage annotations	$\kappa ::= 1$ $ \omega$ $ u, v$	used at most once possibly used many times usage variable
Update flags	$\chi ::= \bullet$ $!$	not updatable/copyable updatable/copyable

Shallow evaluation contexts R , values v , configurations C , heaps H , and stacks S are defined in the same manner as for $FLIX_0$. The operational semantics for $FLIX_2$ is the same as that for $FLIX_0$, *mutatis mutandis*.

Figure 5.6 Well-typing rules for $FLIX_2$ (extends Figure 4.4).

compute only the use of the datum as a whole. The values stored *inside* the datum, occurring neither syntactically nor as free variables, must be treated differently.

We provide a general system for usage analysis of programs containing arbitrary data types. In this respect we differ from Turner *et al.* [TWM95a], who treat only a single algebraic data type, the list.

The new techniques we develop below must deal with two crucial properties of algebraic data types.

Separation between creation and use. Algebraic data types allow values to be used at a site syntactically separate from that at which they were created. The type system must ensure that information regarding the value's use is propagated back to its creation site (Section 5.3.1).

Sharing. Algebraic data may be shared. The type system must ensure that uses from multiple use sites are correctly combined (Section 5.3.2).

There is a wide design space as to exactly how this propagation is achieved. A key contribution of this chapter is a notation, introduced in Section 5.3.1, that al-

allows any particular choice to be represented precisely. This permits a clean separation between the choice (embodied in an annotated data type declaration) and the well-typing rules (which are expressed independently of any particular annotation scheme). The general well-typing rules are discussed in Section 5.3.3, but the choice of an annotation scheme is deferred until Section 5.4. Subtyping becomes a little more complicated in this setting, and we discuss it in Section 5.3.4.

5.3.1 Separation between creation and use

Consider the data type

$$\text{data } P \ \alpha = \text{MkP } \alpha \ \alpha$$

A datum of type $P \ \alpha$ consists of two components, each of type α . The following L_0 program shows such a datum being created and used.

```

letrec  $a = \dots$ 
       $b = \dots$ 
       $p = \text{MkP } a \ b$ 
in
case  $p$  of
   $\text{MkP } x \ y \rightarrow x + x + \dots$ 

```

Notice that the first component of the pair is used twice. When we annotate this program, the binding for a must be annotated with $!$. But the connection between creation and use sites here is not direct: a and p both occur syntactically only once in their scopes, and it is the multiple use of x that forces a 's $!$ annotation:

If x is used many times, so is a .

One design choice here is to consider placement into a data structure to be a (potential) multiple occurrence, and thus to require the topmost annotation of the type of all arguments to a constructor to be ω . This approach is too simple, however; it is well known that *intermediate* data structures are extremely common [GLPJ93, Wad90a], and these are often used at most once. In the absence of a perfect fusion system (*q.v.* [Gil96, LS95]), these are a key target for our analysis. If values placed in such data structures were always annotated ω despite their actual use, the analysis would be drastically weakened (Section 6.8.2).

A better design choice is to use the type of p to convey usage information from the use site (here the binding of x in the case branch) to the creation site (here the use of a as argument to MkP). To achieve this, the type $P \ \text{Int}$ must somehow be parameterised so that the usage type of an individual component (such as $\sigma_{11} = \text{Int}^\omega$) may be constrained (at the use site) or derived from it (at the creation site).

There is a wide design space as to exactly how the type should be parameterised, which we discuss in detail in Section 5.4. The final choice of parameterisation for a particular data type, however, can be represented precisely by its *annotated data type declaration*.

For example, the two options above for treating the P data type may be represented as follows:

$$\text{data } (P \ \alpha)^u = \text{MkP } \alpha^\omega \ \alpha^\omega$$

and

$$\text{data } (P \ u_1 \ u_2 \ \alpha)^u = \text{MkP } \alpha^{u_1} \ \alpha^{u_2}$$

The first adds no usage parameters to the type P , and simply assumes that all the components are annotated ω . The second adds a parameter u_i for each component, and uses it to provide the topmost annotation. Thus $P \ 1 \ 1$ is the type of pairs whose components are each used at most once; $P \ \omega \ 1$ is the type of pairs whose first components may be used many times but whose second components are used at most once; and $P \ \omega \ \omega$ is the type of pairs whose components may each be used many times.

The variable u annotates the overall usage of the datum. This can be used in handling recursion, as in the following declaration for lists:

$$\text{data } (\text{List } u_1 \ \alpha)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} \ (\text{List } u_1 \ \alpha)^u$$

Here $(\text{List } \omega \ \text{Int})^1$ is the type of lists of integers each of which may be used many times, while the list structure itself may be used at most once. The annotation of the recursive instance with u ensures that each list cell has the same usage. The overall-usage annotation is also useful for the annotation scheme (\blacktriangleright -DATA-EQUAL) discussed in Section 5.4.4.1.

In general, then, the annotated data type declaration for a source data type declaration $\text{data } T \ \overline{\alpha_k} = \overline{K_i} \ \overline{t_{ij}}$ is of the form

$$\text{data } (T \ \overline{u_l} \ \overline{\alpha_k})^u = \overline{K_i} \ \overline{\sigma_{ij}}$$

where $(\sigma_{ij})^\natural = t_{ij}$ for each i, j . This declares that the component ij of a datum of type $(T \ \overline{\kappa_l} \ \overline{\tau_k})^\kappa$ has type

$$\sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \ \overline{\kappa_l}/\overline{u_l}, \ \overline{\tau_k}/\overline{\alpha_k}]$$

The declaration adds usage parameters $\overline{u_l}$ to the type of a datum, and also names its overall usage u . The usage-annotated type of each component of the datum is then given in terms of this parameterisation. Notice that the type arguments $\overline{\alpha_k}$ range over τ -types, as discussed in Section 5.2.2. The free usage variables of each σ_{ij} must be contained in $\{u\} \cup \overline{u_l}$, and for technical reasons u must not occur negatively in any σ_{ij} (see Section 5.3.4.3).

5.3.2 Sharing

Data of algebraic type is frequently *shared*: information is often stored in a data structure for later use, and subsequently used multiple times. In this case, what happens to the usage annotations on the components?

Consider the following L_0 program, which makes use of the same data type P as above.

```

letrec a = ...
      b = ...
      p = MkP a b
in
...
case p of
  MkP x y → x + 1
...
case p of
  MkP z w → z + 2

```

Notice that in each case statement the first component of the pair is used just once; yet overall it is used twice, and so the binding for a must again be annotated !. It is not enough to do as above (Section 5.3.1) and propagate the usage type of the use site (x or z) back to the creation site (a). One must also keep track of sharing, or multiple uses of the *whole datum* (here p):

If p is used many times, so are a and b .

One design choice here would be to attempt to add up (in some appropriate sense) the usage annotations on the types of components at each use site, and to use the sum as the type for the creation site.

However, in our type rules so far variable occurrences have always been given the same type as that at the binding site, and multiple occurrences have been handled by an additional constraint. In the present chapter we maintain this approach.⁵ The ordinary syntactic occurrence machinery in (\vdash_2 -LETREC) ensures that the topmost annotation of the type of p is ω ; based on this indication of multiple use of the datum, we now make the component types appropriate for multiple use by the simple constraint that *the use of each component a, b must be at most the use of the datum p as a whole*,⁶ or equivalently that if the datum may be used more than once then so may the components: $|\sigma_{ij}^\circ| \leq \kappa$, where κ is the usage annotation on the datum. Thus the type of p is $(P \ \omega \ \omega \ \text{Int})^\omega$, which forces the type of a , x , and z to be Int^ω . In the uniqueness type system of Clean, this is called *uniqueness propagation* [BS95b] [BS96, §6].

Comparing this with the standard functional encoding of a constructor,⁷ e.g.,

$$\text{MkP } \alpha \ a \ b = \Lambda \beta . \lambda f : \alpha \rightarrow \alpha \rightarrow \beta . f \ a \ b$$

we see that this constraint is similar to that in (\vdash_2 -ABS) on the free variables of a lambda abstraction (Section 3.3.4): the usage of a and b is at most the usage of

⁵Appendix C demonstrates a more fine-grained approach, with careful explicit environment manipulation allowing a different solution to this problem.

⁶Recall that $\omega \leq 1$; thus this says that if the use of p is ω , the use of a must be also.

⁷The standard functional encoding of a constructor was given by Böhm and Berarducci [BB85], generalising Church's representation of the positive integers [Chu41, c. III]. Further references appear in [Pie91, §7.7.1].

the abstraction. The *topmost annotation only* of the component type is constrained because it is the component *as a whole* that is stored by a constructor and retrieved by a case statement. Any internal annotations that should be affected by multiple use of the component will be appropriately constrained by the well-typing rules used in the construction of that component, once multiple use is indicated by constraining the topmost annotation.

In Turner [TWM95a, §3.2], this condition is imposed as a global well-formedness condition on (list) data types;⁸ it is tidier (as here) to follow Clean [BS96, §6], and merely constrain well-typed constructor applications. Since it is the constructor applications alone that propagate the components' use back to their binding sites, this is sufficient for soundness. The fact that certain well-formed types such as $(\text{List } 1 \ \alpha)^\omega$ have no members is of no concern to us.

Gustavsson [GS00b, §3.7] [GS00b, §3.7] has a (Value) rule that constrains the free variables of any value that may be updated to themselves be updatable. This single rule elegantly has the effect of both the free-variable clause in $(\vdash_2\text{-ABS})$ (Section 3.3.4) for abstractions, and the present sharing clause in $(\vdash_2\text{-CON})$ for constructors.

Mogensen [Mog98] goes rather further in his treatment of structure sharing. His system has a fixed set of primitive type constructors (sum, product, and unit). Since recursion is excluded, the system is able to record the usage of each individual component of the data structure at all levels. Clearly such an approach must break down in the presence of general (recursive) algebraic data types.

5.3.3 Typing

Now we can give the well-typing rules for constructors and case (see Figure 5.6). Firstly, we consider the constructor application

$$K_i^{\kappa, \chi} \ \overline{\kappa_l} \ \overline{\tau_k} \ \overline{a_j}$$

That is, constructor K_i (the i th constructor of type T), given usage arguments $\overline{\kappa_l}$, type arguments $\overline{\tau_k}$, and value arguments $\overline{a_j}$, with topmost usage annotation κ and update flag χ . The value arguments are atomic, since we use A-normal form throughout as explained in Section 5.1.4. The constructor arguments record enough information to reconstruct the exact type of the datum, as well as its value. Just as for lambda abstractions (Section 3.2.2), the κ annotation forms part of the type information and is required to allow the type of a subexpression to be computed in isolation, whereas the χ flag forms part of the executable language FLX and is required to control copying of the datum. Unlike for literals n , we cannot simply assume that $(\kappa, \chi) = (\omega, !)$.

The $(\vdash_2\text{-CON})$ rule gives the type of this constructor application as

$$(T \ \overline{\kappa_l} \ \overline{\tau_k})^\kappa$$

⁸Turner *et al.*'s condition states, when translated into our notation, that the type $(\text{List } \sigma)^\kappa$ is well-formed only if $|\sigma| \leq \kappa$. Notice that the type specified here is ill-formed in our context, since we allow only τ -types as arguments to type constructors. The effect of this condition is seen in the rule for cons: $\Gamma \vdash_2 \text{cons } e_1 \ e_2 : (\text{List } \sigma)^\kappa$ holds only if $\Gamma \vdash_2 e_1 : \sigma$ with $|\sigma| \leq \kappa$.

Figure 5.7 The subtype (\preccurlyeq) and primitive (\leq) orderings over $FLIX_2$ (cf. Figure 4.5).

$\psi \preccurlyeq \psi$

$\begin{array}{c} 1 \\ \uparrow \\ \omega \end{array}$

$$\begin{array}{c}
\frac{\kappa_1 \leq \kappa_2 \quad \tau_1 \preccurlyeq \tau_2}{\tau_1^{\kappa_1} \preccurlyeq \tau_2^{\kappa_2}} (\preccurlyeq\text{-ANNOT}) \quad \frac{\tau_1 \preccurlyeq \tau_2}{\forall u . \tau_1 \preccurlyeq \forall u . \tau_2} (\preccurlyeq\text{-ALL-U}) \\
\\
\frac{}{\text{Int} \preccurlyeq \text{Int}} (\preccurlyeq\text{-LIT}) \quad \frac{\sigma_3 \preccurlyeq \sigma_1 \quad \sigma_2 \preccurlyeq \sigma_4}{\sigma_1 \rightarrow \sigma_2 \preccurlyeq \sigma_3 \rightarrow \sigma_4} (\preccurlyeq\text{-ARROW}) \\
\\
\frac{\tau_1 \preccurlyeq \tau_2}{\forall \alpha . \tau_1 \preccurlyeq \forall \alpha . \tau_2} (\preccurlyeq\text{-ALL}) \quad \frac{}{\alpha \preccurlyeq \alpha} (\preccurlyeq\text{-TYVAR}) \\
\\
\frac{\begin{array}{l} \alpha_k \in \text{ftv}^\varepsilon(\sigma_{ij}) \Rightarrow \tau_k \preccurlyeq^\varepsilon \tau'_k \text{ for all } k, \varepsilon, i, j \\ u_l \in \text{fwv}^\varepsilon(\sigma_{ij}) \Rightarrow \kappa_l \leq^\varepsilon \kappa'_l \text{ for all } l, \varepsilon, i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}} \end{array}}{T \overline{\kappa_l} \overline{\tau_k} \preccurlyeq T \overline{\kappa'_l} \overline{\tau'_k}} (\preccurlyeq\text{-TYCON}) \\
\\
\frac{}{\kappa \leq \kappa} \quad \frac{}{\omega \leq \kappa} \quad \frac{}{\kappa \leq 1} \quad \frac{}{u \leq u}
\end{array}$$

The expected types $\overline{\sigma_{ij}^\circ}$ of the components are computed by the substitution given in Section 5.3.1. Sharing is handled soundly by constraining the topmost annotations of these types as described in Section 5.3.2. Notice we need only consider the types of components of *this* constructor, not of all constructors of this type. Finally, the \cdot^\dagger function of Section 4.3.1 is used to compute the correct update flag.

Secondly, we consider the case statement

$$\text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i} \overline{x_{ij}} \rightarrow e_i$$

That is, expression e is scrutinised at type $(T \overline{\kappa_l} \overline{\tau_k})^\kappa$, binding variables $\overline{x_{ij}}$ in branch e_j at the appropriate types. Since the case statement binds variables, the corresponding rule ($\vdash_2\text{-CASE}$) must compute occurrence information $\overline{\text{occur}(x_{ij}, e_i)}$ and constrain the component types $\overline{\sigma_{ij}^\circ}$ accordingly, in the same way as ($\vdash_2\text{-LETREC}$) and ($\vdash_2\text{-ABS}$) (Section 3.3.3). The same substitution as above is used to relate these types to that of the scrutinee, and this latter type is recorded explicitly in the term. During deconstruction the scrutinee of a case statement is always used exactly once, and so it is tempting to set its annotation κ to 1 by analogy with ($\vdash_2\text{-APP}$). But this is incorrect: κ is also used to detect sharing (Section 5.3.2) and must be allowed to take any value in $\{\omega, 1\}$. Annotation κ indicates the use not just of the datum, but

also of its contents.

The arguments of Sections 5.3.1 and 5.3.2 hold regardless of whether or not the data type under consideration is recursive. In all cases a constructor combines components of known type into a datum; whether the components share the type of the datum is irrelevant to the manner in which they are created, shared, and used. Furthermore, the arguments and the resulting well-typing rules are sound no matter what parameterisation is chosen for the data type, although some parameterisations will yield more precise annotations than others, and some will fail to type all well-typed FL_0 programs. However, these issues must certainly be taken into consideration in computing the subtyping relation and in choosing an appropriate parameterisation. It is to these latter topics that we now turn.

5.3.4 Subtyping

Subtyping for algebraic data types is rather more complicated than for polymorphic types. The definition is given in Figure 5.7, and motivated below. To gain an intuition, we investigate a few typical cases first (Section 5.3.4.1) before attempting to generalise our observations. Our first attempt (Section 5.3.4.2) defines subtyping between algebraic data types in terms of subtyping between the types of the components; unfortunately this definition does not give us an algorithm, although it is of use later. Instead, our second attempt (Section 5.3.4.3) defines it in terms of subtyping between the type and usage arguments of the type constructor; this arguably less intuitive definition does yield a usable algorithm.

5.3.4.1 The intuition

Recall that the subtype relation \preccurlyeq may be read as “can be used in the place of” (Section 3.3.5). Now consider the pair type

$$\text{data } (\text{Pair } u_1 \ u_2 \ \alpha \ \beta)^u = \text{Pair } \alpha^{u_1} \ \beta^{u_2}$$

Pair is almost a Cartesian product,⁹ and it is clear that one object of this type can be used in place of another if and only if each component of the first pair can be used in place of the corresponding component of the second pair:

$$\text{Pair } \kappa_1 \ \kappa_2 \ \tau_1 \ \tau_2 \preccurlyeq \text{Pair } \kappa'_1 \ \kappa'_2 \ \tau'_1 \ \tau'_2 \iff \tau_1^{\kappa_1} \preccurlyeq \tau'_1{}^{\kappa'_1} \wedge \tau_2^{\kappa_2} \preccurlyeq \tau'_2{}^{\kappa'_2}$$

Similarly for the sum type

$$\text{data } (\text{Either } u_1 \ u_2 \ \alpha \ \beta)^u = \text{Left } \alpha^{u_1} \mid \text{Right } \beta^{u_2}$$

we have

$$\text{Either } \kappa_1 \ \kappa_2 \ \tau_1 \ \tau_2 \preccurlyeq \text{Either } \kappa'_1 \ \kappa'_2 \ \tau'_1 \ \tau'_2 \iff \tau_1^{\kappa_1} \preccurlyeq \tau'_1{}^{\kappa'_1} \wedge \tau_2^{\kappa_2} \preccurlyeq \tau'_2{}^{\kappa'_2}$$

Arrow types introduce contravariance in the usual way; e.g., for

$$\text{data } (\text{Fun } \alpha)^u = \text{Fun } (\alpha^\omega \rightarrow \text{Int}^\omega)^u$$

⁹Modulo an extra lifting.

we have

$$\text{Fun } \tau \preceq \text{Fun } \tau' \iff \tau' \preceq \tau$$

since $(\tau^\omega \rightarrow \text{Int}^\omega) \preceq (\tau'^\omega \rightarrow \text{Int}^\omega)$ iff $\tau' \preceq \tau$.

5.3.4.2 A coinductive definition

Recalling that an algebraic data type is simply a sum of products and that we may interpret a polymorphic type as a set of monomorphic instances, these examples suggest that an appropriate definition of subtyping for algebraic data types would be

$$\begin{array}{l} \sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \\ \sigma_{ij}^{\circ'} = \sigma_{ij}[\kappa'/u, \overline{\kappa'_l}/\overline{u_l}, \overline{\tau'_k}/\overline{\alpha_k}] \\ \sigma_{ij}^\circ \preceq \sigma_{ij}^{\circ'} \quad \text{for all } i, j \\ \kappa \leq \kappa' \\ \hline \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}} \quad (-) \quad (\preceq\text{-TYCON-NAÏVE}) \\ (T \overline{\kappa_l} \overline{\tau_k})^\kappa \preceq (T \overline{\kappa'_l} \overline{\tau'_k})^{\kappa'} \end{array}$$

This states that two algebraic data types with the same type constructor lie in the subtype relation if corresponding components of each data constructor all lie in the subtype relation.

However, this rule does not work as intended as an inductive definition. Algebraic data types may be recursive, and recursive types would require an infinite inductive proof tree. Consider the type

$$\text{data } (\text{List } u_1 \alpha)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} (\text{List } u_1 \alpha)^u$$

To prove the identity $\text{List } \kappa \tau \preceq \text{List } \kappa \tau$, it is necessary to prove that $\tau^\kappa \preceq \tau^\kappa$ (the head of the list) and that $(\text{List } \kappa \tau)^u \preceq (\text{List } \kappa \tau)^u$ (the tail of the list). To prove the latter, it is necessary to prove that $\tau^\kappa \preceq \tau^\kappa$ and that $(\text{List } \kappa \tau)^u \preceq (\text{List } \kappa \tau)^u \dots$ and so on. It can be seen that the resulting proof tree is infinite. In fact, the relation inductively defined by $(\preceq\text{-TYCON-NAÏVE})$, the least relation satisfying the condition, simply excludes all recursive types – clearly not what we want.

Instead, the rule must be interpreted as *coinductive* [Gor94, JR97], as pointed out by [AC91, PS93]. To indicate this we place a minus sign to the right of the implication line, following [CC92]. A coinductive definition, dual to an inductive one, denotes the greatest post-fixed point of the monotone operator associated with a set of rules. (The existence of such a greatest post-fixed point is guaranteed by the so-called [LNS82] Tarski–Knaster fixpoint theorem.) A proof that a particular pair is in the relation coinductively defined by a set of rules consists of choosing a candidate relation R containing the desired pair and showing that it is a post-fixed point of the associated monotone operator (and hence is contained in the greatest such). For R to satisfy the post-fixed point property means that each pair in R is the conclusion of one of the rules all of whose hypotheses are again in the relation R . Treating $(\preceq\text{-TYCON-NAÏVE})$ as a coinductive definition therefore allows recursive types to be

included. In our list example, the candidate relation R for $\text{List } \kappa \tau \preceq \text{List } \kappa \tau$ is simply $\{\tau^\kappa \preceq \tau^\kappa, (\text{List } \kappa \tau)^u \preceq (\text{List } \kappa \tau)^u\}$.¹⁰

The known techniques for evaluating such coinductive definitions [AC93, BH98] work only for regular data types, relying on the fact that the inductive proof tree, while infinite, has a finite representation as a digraph (*i.e.*, is regular). The nodes of this digraph form the candidate relation of the coinductive proof. Unfortunately our language, following Haskell, permits the use of *non-regular* or *nested* data types, in which a type constructor may appear recursively with different arguments from the head occurrence.¹¹ The inductive proof trees in this case may be infinite and non-regular (and thus the candidate relation infinite)¹² and therefore not susceptible to such algorithms. Instead, we must take another approach.

5.3.4.3 An inductive definition

Intuitively, we may re-express our definition above in the following form:

$$\frac{\begin{array}{l} \alpha_k \in \text{ftv}^\varepsilon(\sigma_{ij}) \Rightarrow \tau_k \preceq^\varepsilon \tau'_k \quad \text{for all } k, \varepsilon, i, j \\ u_l \in \text{fuv}^\varepsilon(\sigma_{ij}) \Rightarrow \kappa_l \leq^\varepsilon \kappa'_l \quad \text{for all } l, \varepsilon, i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}} \end{array}}{T \overline{\kappa_l} \overline{\tau_k} \preceq T \overline{\kappa'_l} \overline{\tau'_k}} (\preceq\text{-TYCON})$$

This makes use of the sets $\text{ftv}^\varepsilon(\psi)$ and $\text{fuv}^\varepsilon(\psi)$, which contain all type and usage variables (respectively) with free ε -ve occurrences in the type ψ , where ε is either $+$ or $-$, and ψ ranges over both τ - and σ -types. These sets are defined in Figure 5.8. Note that $\bar{\varepsilon}$ is sign negation, defined by $\bar{-} \triangleq +$ and $\bar{+} \triangleq -$, and $\varepsilon \cdot \varepsilon'$ is sign multiplication, defined by

\cdot	$-$	$+$
$-$	$+$	$-$
$+$	$-$	$+$

The definition also uses the convenient abbreviation $\tau \preceq^\varepsilon \tau'$, where $\tau \preceq^+ \tau'$ stands for $\tau \preceq \tau'$ and $\tau \preceq^- \tau'$ stands for $\tau' \preceq \tau$.

The inductive rule (\preceq -TYCON) above states that two algebraic data types with the same type constructor lie in the subtype relation if each pair of corresponding

¹⁰In fact this set must be closed under (\preceq -ANNOT); for clarity we omit this step here and in the subsequent footnote.

¹¹An example is the type of balanced binary trees,

$$\text{data } (\text{Bal } \alpha)^u = \text{BZero } \alpha^u \mid \text{BSucc } (\text{Bal } (\alpha, \alpha))^u$$

See Section 5.4.5.5 for references and further discussion.

¹²For example, a candidate relation R for $\text{Bal } \tau \preceq \text{Bal } \tau$ is

$$\{\tau R \tau, \text{Bal } \tau R \text{Bal } \tau, \text{Bal } (\tau, \tau) R \text{Bal } (\tau, \tau), \\ \text{Bal } (\tau, \tau, \tau, \tau) R \text{Bal } (\tau, \tau, \tau, \tau), \dots\}$$

or, more concisely expressed,

$$\{\tau R \tau\} \cup \{\text{Bal } \tau^{2^n} R \text{Bal } \tau^{2^n} \mid n \in \mathbb{N}\}.$$

Figure 5.8 Positive and negative free occurrences.

$ftv^\varepsilon(\psi)$	$fuw^\varepsilon(\psi)$
$ftv^\varepsilon(\tau^\kappa) = ftv^\varepsilon(\tau)$	$fuw^\varepsilon(\tau^\kappa) = fuw^\varepsilon(\kappa) \cup fuw^\varepsilon(\tau)$
$ftv^\varepsilon(\text{Int}) = \emptyset$	$fuw^\varepsilon(\text{Int}) = \emptyset$
$ftv^\varepsilon(\sigma_1 \rightarrow \sigma_2) = ftv^\varepsilon(\sigma_1) \cup ftv^\varepsilon(\sigma_2)$	$fuw^\varepsilon(\sigma_1 \rightarrow \sigma_2) = fuw^\varepsilon(\sigma_1) \cup fuw^\varepsilon(\sigma_2)$
$ftv^\varepsilon(\forall \alpha . \tau) = ftv^\varepsilon(\tau) \setminus \{\alpha\}$	$fuw^\varepsilon(\forall \alpha . \tau) = fuw^\varepsilon(\tau)$
$ftv^\varepsilon(\forall u . \tau) = ftv^\varepsilon(\tau)$	$fuw^\varepsilon(\forall u . \tau) = fuw^\varepsilon(\tau) \setminus \{u\}$
$ftv^+(\alpha) = \{\alpha\}$	$fuw^\varepsilon(\alpha) = \emptyset$
$ftv^-(\alpha) = \emptyset$	$fuw^+(u) = \{u\}$
	$fuw^+(\kappa) = \emptyset, \quad \kappa \in \{1, \omega\}$
	$fuw^-(\kappa) = \emptyset$
$ftv^\varepsilon(T \overline{\kappa_l} \overline{\tau_k}) = \left\{ \alpha \mid \bigvee_{k, \varepsilon'} \left(\alpha \in ftv^{\varepsilon \cdot \varepsilon'}(\tau_k) \wedge \alpha_k \in \bigcup_{ij} ftv^{\varepsilon'}(\sigma_{ij}) \right) \right\}$	
$fuw^\varepsilon(T \overline{\kappa_l} \overline{\tau_k}) = \left\{ u \mid \bigvee_{l, \varepsilon'} \left(u \in fuw^{\varepsilon \cdot \varepsilon'}(\kappa_l) \wedge u_l \in \bigcup_{ij} fuw^{\varepsilon'}(\sigma_{ij}) \right) \vee \bigvee_{k, \varepsilon'} \left(u \in fuw^{\varepsilon \cdot \varepsilon'}(\tau_k) \wedge \alpha_k \in \bigcup_{ij} ftv^{\varepsilon'}(\sigma_{ij}) \right) \right\}$	

where data $(T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}}$

type arguments lies in the subtype relation covariantly (respectively contravariantly) if that argument occurs positively (negatively) in a component of a data constructor. An argument may occur both positively and negatively, in which case both cases apply. We have dispensed with references to u (and κ, κ') by ensuring that all occurrences of u in the σ_{ij} are positive (Section 5.3.1), so the consequent test $\kappa \leq \kappa'$ will already have been performed by (\preceq -ANNOT).

This definition is clearly inductive in the structure of the type, and can be implemented efficiently. The definitions of $fuw^\varepsilon(\psi)$ and $ftv^\varepsilon(\psi)$, the free usage and type variables of a type, are also inductive, and may be readily computed in advance (when processing data type declarations) by means of iteration to a fixpoint. Termination is guaranteed for all finite sets of data types (even non-regular, mutually recursive ones), since there are a finite number of variables α , a finite number of types ψ occurring as subterms of constructor argument types, and two values for ε , ensuring a finite maximum size of the relation.

It seems ‘obvious’ that the two definitions are equivalent, but I have been unable to find a proof. That (\preceq -TYCON) implies (\preceq -TYCON-NAÏVE) is relatively easily proven, but the reverse direction is somewhat harder: proving that a coinductively-defined set is contained within an inductively-defined one seems not a very common thing to do! It is possible that techniques from [BH98], which gives an inductive presentation of another coinductively-defined subtyping relation, may apply. Luckily, we only need the former result for our proof of soundness. This is a trivial consequence

of a result proven in the appendix, Lemma D.15.

5.4 Annotation of data type declarations

In Section 5.3.1 we introduced the form of the $FLIX_2$ data type declaration, and showed how it could encode a variety of design choices regarding the precise parameterisation of a data type. In Section 5.3.3 we saw that soundness is not dependent on the particular parameterisation; all parameterisations lead to sound well-typing rules. However, not all parameterisations lead to a good analysis: there is a tension between maximising precision and minimising the cost of the analysis, and certain parameterisations are incomplete, yielding an analysis that cannot type all programs.

In this section we discuss how best to annotate FL_0 data type declarations. Some typical data structures that we use as examples are listed in Section 5.4.1. We begin the discussion by considering two extreme approaches to the problem (Section 5.4.2). In the light of these examples, we discuss what exactly are the factors to be considered in designing or selecting an annotation scheme (Section 5.4.3). Finally, we present four workable schemes (Section 5.4.4). Recursive data types introduce additional difficulty in design, and we consider these issues separately (Section 5.4.5). The final choice of scheme will depend on the results of experiment, which we defer to Chapter 6.

5.4.1 Some typical data structures

These typical FL_0 data type declarations will be used to illustrate aspects of the annotation schemes below.

```
data List  $\alpha$  = Nil | Cons  $\alpha$  (List  $\alpha$ )
```

```
data Tree  $\alpha$  = Leaf | Node (Tree  $\alpha$ )  $\alpha$  (Tree  $\alpha$ )
```

```
data Customer = MkCustomer Int String String (List String) Bool
```

```
data Term  $\alpha$  = Var  $\alpha$  | App (Term  $\alpha$ ) (Term  $\alpha$ ) | Lam  $\alpha$  (Term  $\alpha$ )
```

```
data Skew = SLeft | SNone | SRight
```

```
data AVLTree  $\alpha$  = ALeaf | ANode (AVLTree  $\alpha$ )  $\alpha$  Skew (AVLTree  $\alpha$ )
```

```
data Rose  $\alpha$  = RLeaf  $\alpha$  | RNode (List (Rose  $\alpha$ ))
```

We assume our language contains types `Int`, `String`, and `Bool` as primitive. AVL-trees (a form of balanced binary tree) are due to [AVL62].

We add one somewhat contrived data type declaration intended to illustrate the

behaviour of the various schemes as concisely as possible:

$$\text{data } R \alpha \beta = R1 \alpha \mid R2 (\text{Int} \rightarrow \beta)$$

5.4.2 Two extreme approaches

We are required to translate an FL_0 data type declaration into an $FLIX_2$ (annotated) data type declaration, as follows:

$$\blacktriangleright \text{data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \rightsquigarrow \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}}$$

This involves (i) choosing how many usage parameters $\overline{u_l}$ to add, and (ii) choosing how to annotate each type t_{ij} using the parameters $\overline{u_l}$ and topmost usage annotation u of the declared type.

The simplest approach is to add no parameterisation at all, fixing all annotations statically. To permit all possible patterns of construction and use, it is necessary that the annotations are all ω , as in the first alternative of Section 5.3.1. This seems reasonable if we assume that one usually places data in a data structure in order that it may be used repeatedly. We may write this

$$\frac{\sigma_{ij} = [t_{ij}]_{\sigma}^{\omega} \quad \text{all } i, j}{\blacktriangleright \text{data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \rightsquigarrow \text{data } (T \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}}} (\blacktriangleright\text{-DATA-MANY})$$

That is, for each component type t_{ij} place a ω annotation on top and at every usage annotation position within the type. This ensures that each component can be used as many times as desired, and that any appropriately- FL_0 -typed expression can be placed in the data structure.

The example data type R becomes

$$\text{data } (R \alpha \beta)^u = R1 \alpha^{\omega} \mid R2 (\text{Int}^{\omega} \rightarrow \beta^{\omega})^{\omega}$$

Notice especially that ω annotations are essential in negative positions as well as positive:¹³ if $R2$'s argument were to be typed $(\text{Int}^1 \rightarrow \beta^{\omega})^{\omega}$ then a function using its argument more than once could not be placed in the data structure.

This scheme clearly has the least possible overhead (none at all), but it also has very low precision (anything that goes into or comes out of a data structure is annotated ω).

At the other end of the spectrum, we may consider parameterising over every annotation position of every component type of the constructor, thus:

$$\frac{\begin{array}{l} \sigma_{ij} = [t_{ij}]_{\sigma}^{fresh} \quad \text{all } i, j \\ \overline{u_l} = fuv(\overline{\sigma_{ij}}) \end{array}}{\blacktriangleright \text{data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \rightsquigarrow \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}}} (\blacktriangleright\text{-DATA-ALL-BAD})$$

¹³Positive (covariant) and negative (contravariant) positions in a type are defined in Section 3.2.1; see also Figure 5.8.

This certainly yields the maximum possible precision, but at the expense of huge types. It is also ill-defined for recursive data types (an infinite number of usage arguments would be required). Our example becomes

$$\text{data } (R \ u_1 \ u_2 \ u_3 \ u_4 \ \alpha \ \beta)^u = R1 \ \alpha^{u_1} \mid R2 \ (\text{Int}^{u_2} \rightarrow \beta^{u_3})^{u_4}$$

Even this simple example has four usage arguments; more realistic ones may have ten or twenty. Values for these must be given at every constructor application, and constraints on them must be generated wherever the type is used.

5.4.3 The issues

The two examples above have introduced the crucial issues to consider in choosing an annotation scheme.

Precision. We want sufficient precision in the types of constructor components that in realistic programs, usage inference is not significantly affected by intervening data structures.

For example, a very common pattern (found, *e.g.*, in queens from the NoFib suite, Section 6.7.1) is the intermediate list. One function lazily generates a list consumed by another; the structure of the list is used at most once (and each cell is garbage-collected once the consumer has used it), but each of the stream of elements contained in the list may be used multiple times in processing. A good annotation scheme should permit usage inference to detect that each list cell is used at most once, a fact that the optimiser can use to avoid ever building the list explicitly.

However, the lack of constrained quantification in our type language restricts the useful degree of precision that can be introduced. Constructor application has a special rule ($\blacktriangleright_2\text{-CON}$), which includes an explicit constraint on the top-most annotations of the element types that cannot be represented within the type system. This means that constructor application cannot be replaced with function application – the exact type of a constructor cannot be abstracted, because we cannot give it a principal type. Thus a simple wrapper function (*e.g.*, $wR2 = \Lambda \alpha . \Lambda \beta . \lambda f : \text{Int} \rightarrow \beta . R2 \ \alpha \ \beta \ f$) necessarily loses precision, and there is some maximum useful degree of precision. We investigate this question experimentally in Section 6.8.2, since it clearly depends (at least in part) on typical coding style.

Cost. We want the additional cost of processing usage-annotated programs, in terms of compilation time, intermediate file size, and (un)readability of error messages and internal compiler state information, to be kept as low as possible.

In practice, this means we want as few parameters as possible. The vector of usage arguments appears in every constructor application, and constraints on them are generated at least at every usage site. The vector also appears in every type signature or annotation. All of these increase processing time and file size, and make usage-annotated terms less readable.

Typeability. We must be able to type every well-typed FL_0 program.

An example of an annotation scheme violating this requirement was given in Section 5.4.2: if a constructor argument were to be typed $(\text{Int}^1 \rightarrow \beta^\omega)^\omega$ then functions using their argument more than once could not be placed in the data structure.

In general, (i) given a suitable instance of the data type we must be able to use the components of the datum more than once, and indeed in any well- FL_0 -typed way we choose; and (ii) we must also be able to build a (suitably parameterised) datum from any well- FL_0 -typed components we choose. This is *maximal applicability*, as discussed in Sections 1.5.6, 3.8, and 4.5.1.

In practice, this simply means that no 1-annotations may appear anywhere in the types $\overline{\sigma}_{ij}$ of the components. 1-annotations may not appear in positive positions for reason (i), and may not appear in negative positions for reason (ii) (as in the example). Variable and ω -annotations are permitted anywhere. This constraint is sufficient to achieve typeability, as we show in Theorem 5.7.

One *non-issue* is *inference*. The well-typing and inference rules given in Sections 5.3 and 5.5 treat all annotation schemes uniformly, and can be proven sound independent of the particular scheme used.

5.4.4 Some workable approaches

We now give four points on the cost–precision spectrum, namely (\blacktriangleright -DATA-EQUAL), (\blacktriangleright -DATA-FULL), (\blacktriangleright -DATA-CLEAN), and (\blacktriangleright -DATA-RESTR). All four annotation schemes satisfy the typeability requirement. For now we consider only the simplest form of recursion, direct self-recursion, deferring the treatment of more complex forms to Section 5.4.5.

5.4.4.1 Equal

It is possible to increase the precision of the (\blacktriangleright -DATA-MANY) rule of Section 5.4.2 without adding any extra usage parameters. The idea is to equate the usage of components of the data structure with usage of the data structure itself:

$$\frac{\sigma_{ij} = (\lceil t_{ij} \rceil_\tau^\omega)^u \quad \text{all } i, j}{\blacktriangleright \text{ data } T \overline{\alpha}_k = \overline{K_i \overline{t_{ij}}} \rightsquigarrow \text{ data } (T \overline{\alpha}_k)^u = \overline{K_i \overline{\sigma_{ij}}} \quad (\blacktriangleright\text{-DATA-EQUAL})}$$

Our example data type R becomes

$$\text{data } (R \alpha \beta)^u = R1 \alpha^u \mid R2 (\text{Int}^\omega \rightarrow \beta^\omega)^u$$

To avoid complications, only the topmost usage annotations of the components are made equal to u ; any annotations lying deeper inside the component types are given the pessimistic annotation ω .¹⁴

¹⁴Note that if (\blacktriangleright -DATA-EQUAL) has been used for all the data types on which the present one depends, deeper annotations can arise only from the arrow type constructor as other type constructors will have no usage parameters.

This annotation scheme is ideal in terms of cost: usage-annotated data types take no extra parameters at all, and so compilation times, intermediate file size, and usage-annotated terms are all completely unaffected. In fact, this annotation scheme may be implemented with only minimal modifications to the compiler, since the syntax of terms and types has changed so little.

In terms of precision, however, this annotation scheme is inadequate. The results of our experiments are discussed in Section 6.8; here it suffices to note that while it may often be reasonable to identify the usage of components of different constructors or of the same constructor, very often the *structure* of a data type is used differently from its *contents*, and identifying these is not reasonable. This is the case for example in the intermediate list pattern discussed in Section 5.3.1. Using (►-DATA-EQUAL) leads to the unification of too many usage variables, and the multiple use of a single component leads to all being equated with ω to the detriment of the resulting types.

5.4.4.2 Full

To further increase the precision we must add additional usage parameters. An obvious way to do this is to give a distinct topmost usage annotation to each component of the data type. As with (►-DATA-EQUAL), for simplicity we give pessimistic deeper annotations, except in the case of direct self-recursion where we give the recursive instance the same type as that being declared:

$$\frac{\sigma_{ij} = \begin{cases} (T \overline{u_l} \overline{\alpha_k})^u, & \text{if } t_{ij} = T \overline{\alpha_k} \\ ([t_{ij}]_\tau^\omega)^{u'}, & \text{otherwise, } u' \text{ fresh} \end{cases} \quad \text{all } i, j}{\overline{u_l} = fuv(\overline{\sigma_{ij}})} \quad (\text{►-DATA-FULL})$$

$$\text{► data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \rightsquigarrow \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}}$$

Our example data type R becomes

$$\text{data } (R \ u_1 \ u_2 \ \alpha \ \beta)^u = R1 \ \alpha^{u_1} \mid R2 \ (\text{Int}^\omega \rightarrow \beta^\omega)^{u_2}$$

This annotation scheme gives excellent precision for many common data types, including all those defined in Section 5.4.1 except *Rose*. It does not deal well with data types involving mutual recursion or indirect recursion; the usage parameters of the inner type constructor are pessimistic and the topmost annotation is in practice soon unified with u or ω .

The cost however can be significant. For a data type with n constructors each of m arguments, a total of $n \cdot m$ usage parameters will be generated. Medium to large data types (such as for the abstract syntax of a language) may yield twenty or thirty parameters, with consequent types such as $(Expr \ \kappa_1 \ \kappa_2 \ \kappa_3 \ \kappa_4 \ \kappa_5 \ \kappa_6 \ \kappa_7 \ \kappa_8 \ \kappa_9 \ \kappa_{10} \ \kappa_{11} \ \kappa_{12} \ \kappa_{13} \ \kappa_{14} \ \kappa_{15} \ \kappa_{16} \ \kappa_{17} \ \kappa_{18} \ \kappa_{19} \ \kappa_{20} \ \kappa_{21} \ \tau)^\kappa$. This is probably not a good idea in terms of analysis and type-directed optimiser efficiency.

5.4.4.3 Clean

A well-motivated restriction of (►-DATA-FULL) is (►-DATA-CLEAN), so named because it is based on the treatment of data structures in the language Clean ([BS93a,

def. 8.6], referenced in [BS96, §6]). One observes that frequently a data structure of type (say) $T \alpha \beta$ is comprised of some kind of relatively uninteresting framework, along with components having the parameter types, α and β . The two types of component must in general represent different classes of object, but one assumes that components of the same type will be treated similarly. Thus (\blacktriangleright -DATA-CLEAN) gives the same usage annotation to every component of type α , and another usage annotation to every component of type β , *etc.*

The problem arises, how to annotate components of type not α or β . Evidently direct self-recursion should recursively use the same type being declared; for ground types we choose to annotate with ω since 1 would not be sound; and for other types such as $(\alpha \rightarrow \beta)$ or $(\text{List } (T \alpha \beta))$ we bail out and annotate freely with ω . The assumptions motivating the (\blacktriangleright -DATA-CLEAN) annotation scheme assume that these cases do not occur frequently. The resulting rule is as follows:

$$\frac{\sigma_{ij} = \begin{cases} \alpha^{u_k}, & \text{if } t_{ij} = \alpha \\ (T \overline{u_k} \overline{\alpha_k})^u, & \text{if } t_{ij} = T \overline{\alpha_k} \\ \lceil t_{ij} \rceil_\sigma^\omega, & \text{otherwise} \end{cases} \quad \text{all } i, j}{\blacktriangleright \text{ data } T \overline{\alpha_k} = K_i \overline{t_{ij}} \rightsquigarrow \text{ data } (T \overline{u_k} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}}} \quad (\blacktriangleright\text{-DATA-CLEAN})$$

Note that the “propagation” requirement of [BS93a] [PvE98, §4.5.3] does not arise here; it is handled instead by the constraint in the (\vdash_2 -CON) rule. This makes our rule simpler than the Clean type attribution rule.¹⁵ The (\blacktriangleright -DATA-CLEAN) rule can also be seen as a generalisation of the rule used by Turner *et al.* in [TWM95a] to describe lists.

Our example data type R becomes

$$\text{data } (R \ u_\alpha \ u_\beta \ \alpha \ \beta)^u = R1 \ \alpha^{u_\alpha} \mid R2 \ (\text{Int}^\omega \rightarrow \beta^\omega)^\omega$$

This annotation scheme has excellent precision for common data types Pair, Either, List, and Tree. But it does not work so well for Customer, AVLTree, or Term. There are two problems. Firstly, ground components (such as the fields of Customer or the skew parameter of AVLTree) or components of non-variable type (such as the List (Rose α) parameter of Rose) automatically get the pessimistic annotation ω , even if usage information would contribute to the precision of the analysis. Secondly, different components that happen to have the same variable type (such as the binders and the bound variables of Term) are not distinguished, even if they are used differently (consider a renaming function, which uses each binder multiple times but each bound variable once).

The cost is tightly bounded: there are exactly as many usage parameters as there are type parameters, and so types at worst double in size. Thus using the scheme (\blacktriangleright -DATA-CLEAN) gives predictable and small cost, and good precision on some common data types, but for many other data types the results are poor. The fact that essentially this rule is used for uniqueness typing in the language Clean suggests that it probably works fairly well in practice.

¹⁵The Clean rule also descends into nontrivial types, which we do not attempt.

5.4.4.4 *Restricted*

The (►-DATA-CLEAN) annotation scheme may be viewed as an attempt to limit the cost of (►-DATA-FULL) by placing a restriction on the number of usage parameters and choosing annotations for each component, according to some rule, from the parameters or ω . We may generalise this approach by restricting the number of parameters in some other way, say to an arbitrary small integer N . This approach will have the cost advantages of (►-DATA-CLEAN), but will deal well both with ground component types as found in, e.g., *Customer* and with distinct components sharing a type as found in, e.g., *Term*.

The rule is the same as that for (►-DATA-FULL), but if the resulting number of parameters is greater than the limit, we perform unification until it is within the limit. For example, we may take the original parameters in order and map them to the cyclic list of available parameters, or we may start the cycle again for each constructor, or we may assign all annotations for a single constructor the same usage parameter. We hope that experimentation will suggest effective schemes.

As an optimisation, we may observe that one need not distinguish the use of a data type with a single component (*i.e.*, one unary constructor) from the use of its component. Thus we may annotate its declaration data $(T \alpha)^u = K \tau^u$ rather than data $(T u_1 \alpha)^u = K \tau^{u_1}$.

$$\sigma_{ij} = \begin{cases} ([t_{ij}]_\tau^\omega)^u, & \text{if } i = j = n = m = 1 \\ (T \overline{u_l} \overline{\alpha_k})^u, & \text{if } t_{ij} = T \overline{\alpha_k} \\ ([t_{ij}]_\tau^\omega)^{u'}, & \text{otherwise, } u' \text{ fresh} \end{cases} \quad \text{all } i, j$$

fresh $\overline{u_l}$
 S some appropriate mapping from $fu\overline{v}(\overline{\sigma_{ij}}) \setminus \{u\}$ onto $\overline{u_l}$
 $\sigma'_{ij} = S\sigma_{ij} \quad \text{all } i, j$

$$\text{► data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \rightsquigarrow \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma'_{ij}} \quad (\text{►-DATA-RESTR})$$

For simple data types, this approach is identical to (►-DATA-FULL), with its excellent precision; but we have the added assurance that the cost cannot expand beyond a fixed bound. With an appropriate bound, this approach is probably the best to choose.

5.4.4.5 *Discussion*

Of the four annotation schemes considered above, (►-DATA-FULL) (or, for efficiency, some restriction of it) is probably the best for use in practice.

In terms of precision, the implementation experience (Section 6.8.2) suggests that while (►-DATA-FULL) is sufficient for most programs there are several cases where even more precision is required. This is likely to be because of the use of dictionaries, which are data structures that store functions. For good results the usage annotation structure on these functions should not be lost when placed in the dictionary; this entails an annotation scheme more along the lines of (►-DATA-ALL-BAD), although of course dealing appropriately with recursion.

In terms of efficiency, it is clear that some restriction must be imposed to avoid ludicrous numbers of usage arguments. The restriction we used is described in Sec-

tion 6.4.11, but this is rather *ad hoc* and further investigation would be well worthwhile.

In the next section we discuss how we might better deal with recursive data types, another issue which in the implementation we deal with in a fairly *ad hoc* manner.

5.4.5 Recursion

The rules above deal only with the simplest form of recursion, namely direct self-recursion. It is possible to treat more complex forms of recursion in a principled way, and we attempt to outline one such method in this section. We do not work out all the details, or deal with all expressible forms of recursion. This is an area of current research, and needs further work to yield a good general annotation scheme.

Much of the power of user-defined algebraic data types comes from their unrestrained ability to refer (potentially recursively) to each other and themselves. Any annotation scheme must take this into account, at the very least ensuring well-definedness on all legal data type declarations, and preferably yielding good solutions with respect to the issues of Section 5.4.3 even for complex recursive patterns.

The annotation scheme developed in this section is intuitively reasonable, and well-defined for all expressible data type declarations, but it is not the only possible scheme. Other schemes may do better in specific cases or even in general, but this one is sufficient for our purposes.

5.4.5.1 Direct self-recursion

Consider one of the simplest, *self-recursive* data types:

$$\text{data List } \alpha = \text{Nil} \mid \text{Cons } \alpha \text{ (List } \alpha \text{)}$$

For a first attempt at parameterising this along the lines of (\blacktriangleright -DATA-ALL-BAD), we might place one annotation, u_1 , on the α and another, u_2 , on the $(\text{List } \alpha)$. Thus List will take two usage arguments. Or will it? The recursive occurrence of List must take two usage arguments also, u_3 and u_4 ; now List must take *four* usage arguments. But this means the recursive occurrence must take *six*... and so on:

$$\text{data (List } u_1 \ u_2 \ u_3 \ u_4 \ u_5 \ u_6 \ \dots)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} \text{ (List } u_3 \ u_4 \ u_5 \ u_6 \ \dots \ \alpha)^{u_2}$$

We are attempting here to give a separate usage annotation to every possible list element and list cell; since there are an infinite number of these we are doomed to fail. In order to stop this infinite unfolding, we must identify the usage arguments of the recursive instance of List with those of the List being defined, thus:

$$\text{data (List } u_1 \ u_2)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} \text{ (List } u_1 \ u_2 \ \alpha)^{u_2}$$

This is now a well-defined fixed point. However, if we unfold the definition a few times we can see an asymmetry:

$$\begin{array}{lcl}
 (\text{List } u_1 \ u_2)^u & \ni & \text{Nil}^u \\
 & | & \text{Cons}^u \ \alpha^{u_1} \ \text{Nil}^{u_2} \\
 & | & \text{Cons}^u \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ \text{Nil}^{u_2}) \\
 & | & \text{Cons}^u \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ \text{Nil}^{u_2})) \\
 & | & \text{Cons}^u \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ (\text{Cons}^{u_2} \ \alpha^{u_1} \ \text{Nil}^{u_2}))) \\
 & | & \dots
 \end{array}$$

Notice that while u_1 is used to annotate every list element, u_2 annotates every list cell *but the first*, for which u is used. It seems more natural and symmetrical to use u (“the usage of the list”) for *all* list cells, and so we revise the declaration as follows:

$$\text{data } (\text{List } u_1)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} \ (\text{List } u_1 \ \alpha)^u$$

This also saves a usage argument, reducing the cost by around half, by losing precision only in cases where the first cell of a list is treated distinctly from all other cells, presumably an unlikely case.

A similar pair of arguments applies to the data type of trees

$$\text{data } \text{Tree } \alpha = \text{Leaf} \mid \text{Node } (\text{Tree } \alpha) \ \alpha \ (\text{Tree } \alpha)$$

which would otherwise lead to a veritable sorcerer’s apprentice¹⁶ of usage arguments. It would not be especially useful to have different usage annotations for the left and right halves of the tree while within those halves annotations were all identical, and so we translate the declaration to:

$$\text{data } (\text{Tree } u_1 \ \alpha)^u = \text{Leaf} \mid \text{Node } (\text{Tree } u_1 \ \alpha)^u \ \alpha^{u_1} \ (\text{Tree } u_1 \ \alpha)^u$$

Another way of seeing that this is the natural annotation pattern is to consider the declarations of the above data types by explicit recursion:

$$\text{data } (\text{List } u_1 \ \alpha)^u = \mu\sigma . 1 + \alpha^{u_1} \times \sigma$$

$$\text{data } (\text{Tree } u_1 \ \alpha)^u = \mu\sigma . 1 + \sigma \times \alpha^{u_1} \times \sigma$$

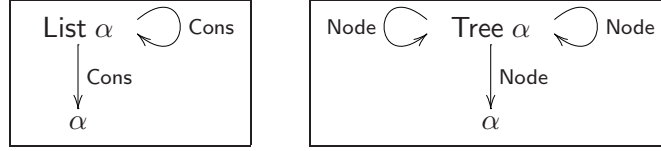
This view extends to some of the more complex forms of recursion considered below, but not to all – the algebraic data type declaration mechanism is more general than this form of explicit recursion.

In general, then, for data types involving direct self-recursion we recurse at exactly the type being declared (the type to the left of the equals sign). This means that:

Isomorphic portions of the data structure
are given equal usage annotations.

¹⁶This expression derives “from Goethe’s *Der Zauberlehrling* via Paul Dukas’s *L’apprenti sorcier* in the film *Fantasia*.” [R⁺01]

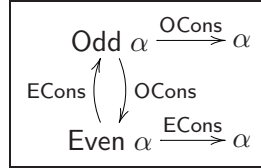
We can visualise this in graphical form:



Each type is represented by a node, and each component of each constructor is represented by an arc. Each constructor maps the constructed data type to the types of each component. The unfolding of this finite graph is the regular tree containing one node for each element of the data type.

5.4.5.2 Direct mutual recursion

The same intuition can be extended to directly *mutually-recursive* data type declarations, such as the types of odd and even lists:¹⁷

$$\begin{aligned} \text{data Odd } \alpha &= \text{OCons } \alpha \text{ (Even } \alpha) \\ \text{data Even } \alpha &= \text{ENil} \mid \text{ECons } \alpha \text{ (Odd } \alpha) \end{aligned}$$


The naïve approach would use four usage parameters, one for each arg of OCons and ECons; but this suffers from a similar asymmetry to that we encountered when we gave two usage parameters to List above. On the other hand, annotating all instances of types in the mutually-recursive group with the same topmost annotation u , leaving just two usage parameters, arguably loses more precision than necessary. Following again the dictum “isomorphic portions of the data structure have equal usage annotation” we obtain:¹⁸

$$\begin{aligned} \text{data (Odd } u_1 \ u_2 \ u_3 \ \alpha)^u &= \text{OCons } \alpha^{u_1} \text{ (Even } u_1 \ u_2 \ u \ \alpha)^{u_3} \\ \text{data (Even } u_1 \ u_2 \ u \ \alpha)^{u_3} &= \text{ENil} \mid \text{ECons } \alpha^{u_2} \text{ (Odd } u_1 \ u_2 \ u_3 \ \alpha)^u \end{aligned}$$

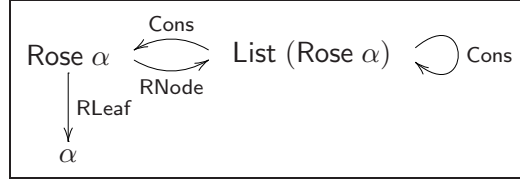
The name of the topmost annotation is arbitrary; we have named that of Even u_3 rather than u for clarity.

¹⁷Of course, we must extend the annotation scheme to translate an entire mutually recursive *group* of data type declarations ([BB85] suggests the term “data system”) simultaneously, rather than one declaration at a time.

¹⁸To obtain exactly this, when computing the isomorphic portions we must distinguish the two α occurrences; yet we must not distinguish the left and right subtrees in the declaration of Tree. This is achieved by distinguishing each component of each constructor of each data type (by means of a tag, (T, i, j)), *except* in the case that the type is one of those being defined in the present mutually-recursive group (in which case we omit the tag). We can identify isomorphic portions of the infinite expansion of the data structure by using an algorithm similar to that of [AC93], or use techniques from [BH98].

5.4.5.3 Indirect recursion

Up to this point we have considered only *direct* recursion, *i.e.*, that in which the type being recursed upon appears as one of the t_{ij} s. But *indirect* recursion is also possible, in which the type being recursed upon appears as a strict subterm of one of the t_{ij} s. A standard example is the type of trees of variable arity, *a.k.a.* rose trees:

$$\text{data Rose } \alpha = \text{RLeaf } \alpha \mid \text{RNode (List (Rose } \alpha))$$


Each node of the rose tree contains a List of the zero or more subtrees of that node; each element of the list is itself a rose tree. Thus the recursion passes through another type constructor, that of lists. In annotating this data type, we must provide usage annotations for the intervening List type constructor as well as for the recursive instance and the leaf component α .

Our guiding principle tells us we wish the subtrees (that is, the elements of the list) to have exactly the same type as the tree itself. Since the usage argument to List annotates the elements, this requires that argument to be u ; the enclosed Rose type constructor will take the same arguments as the defined type. The list as a whole will be given a distinct usage argument, as will the leaf component. These considerations yield:

$$\text{data (Rose } u_1 \ u_2 \ \alpha)^u = \text{RLeaf } \alpha^{u_1} \mid \text{RNode (List } u \ (\text{Rose } u_1 \ u_2 \ \alpha))^{u_2}$$

It is also possible for the type being recursed upon to appear under the arrow type constructor, in which case similar considerations apply. For example, consider a formulation of lazy lists (these are not useful in Haskell, but become so in strict languages such as ML):

$$\text{data LazyList } \alpha = \text{LNil} \mid \text{LCons } \alpha \ (\text{Unit} \rightarrow (\text{LazyList } \alpha))$$

The corresponding usage annotation would be:

$$\text{data (LazyList } u_1 \ u_2 \ u_3 \ \alpha)^u = \text{LNil} \mid \text{LCons } \alpha^{u_1} \ (\text{Unit}^{u_2} \rightarrow (\text{LazyList } u_1 \ u_2 \ u_3 \ \alpha)^{u_3})^{u_3}$$

This distinguishes the usage of the list itself, the head element, and the function yielding the tail; the tail itself has the same usage as the whole list. It also unfortunately distinguishes the usage of the unit component (which is certainly not used in any reasonable implementation); this is an unfortunate consequence of the generality of our system.

One potential problem with this form of recursion occurs when the type being recursed upon appears in a negative position in one of the t_{ij} . Negative recursion arises in a number of places.¹⁹ Parsers and other continuation-passing styles are one

¹⁹Thanks for these examples to members of the Haskell list haskell@haskell.org, 4 July, 2001: Sebastien Carlier, Koen Claessen, Ralf Hinze, Lars Mathiesen, and Mark Shields.

example: an LR parser might use a data type like the following:

$\text{data State} = \text{MkState} \begin{array}{c} \nearrow \text{next} \\ (\text{Sym} \rightarrow \text{List} (\text{Pair State Sym}) \rightarrow \text{List Sym} \rightarrow \text{Sym}) \end{array}$

Another example is `Fix`, used to encode the Y-combinator in Haskell [MH95]:

$\begin{aligned} \text{data Fix } \alpha &= \text{F } (\text{Fix } \alpha \rightarrow \alpha) \\ \text{fix} &:: (\alpha \rightarrow \alpha) \rightarrow \alpha \\ \text{fix } f &= (\lambda(F\ x) . f\ (x\ (F\ x)))\ (F\ (\lambda(F\ x) . f\ (x\ (F\ x)))) \end{aligned}$

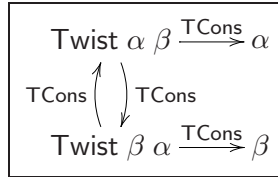
Other examples come from object encodings, higher-order abstract syntax, and domain encodings.

In all these cases, a naïve analysis will result in all usage arguments becoming bivariant (both covariant and contravariant), and hence non-subtypeable. The resulting poisoning will almost invariably force its usage arguments to ω . Following [MH95], such cases could be recognised and the usage arguments duplicated, with one set used for positive occurrences and others negatively.

5.4.5.4 Non-uniform recursion

Another standard tricky example is twist-lists, with elements of two alternating types. Here the recursive instance of the type has different parameters from the initial instance, but the resulting tree is still regular:

$\text{data Twist } \alpha\ \beta = \text{TNil} \mid \text{TCons } \alpha\ (\text{Twist } \beta\ \alpha)$



Once again, this is straightforwardly handled by our dictum; observe that we alternate between types `Twist $\alpha\ \beta$` (with head type α) and `Twist $\beta\ \alpha$` (with head type β), and hence may give each a different usage annotation:

$\text{data } (\text{Twist } u_1\ u_2\ u_3\ \alpha\ \beta)^u = \text{TNil} \mid \text{TCons } \alpha^{u_1}\ (\text{Twist } u_2\ u_1\ u\ \beta\ \alpha)^{u_3}$

A particularly elegant combination of negative and non-uniform recursion occurs in *hyperfunctions*, used in [LKS00] for zip fusion:

$\text{data H } \alpha\ \beta = \text{Cont } (\text{H } \beta\ \alpha \rightarrow \beta)$

Here α occurs only negatively and β only positively, thus sidestepping the usual problems of bivariate in negatively-recursive data types.

5.4.5.5 Non-regular recursion

Unfortunately, there is one form of recursion where our techniques fail. All the examples we have considered so far have involved *regular* recursion: the tree obtained by unfolding the declaration, while infinite, has only finitely many different subtrees (i.e., it is regular). However, algebraic data types may also be *non-regular*, having a tree with infinitely many different subtrees. Such data types (also known as *nested* data types [BM98]) were not thought useful until recently, and indeed while they are definable in Standard ML it is not possible to use them, due to the absence of polymorphic recursion [Myc84, Hen93]. There is no such restriction in Haskell, however, and Okasaki's work in particular [Oka98, Oka97, Oka99] has shown that such data types are indeed very useful.

To understand the difficulty of translating such a data type, consider the type of balanced binary trees (we here assume a type $(P \ \alpha)$ of uniform pairs with elements $(MkP \ \alpha \ \alpha)$, which we abbreviate as (α, α)):

$$\text{data Bal } \alpha = \text{BZero } \alpha \mid \text{BSucc } (\text{Bal } (P \ \alpha))$$

The unfoldings of this data type are as follows:

$$\begin{array}{lcl} \text{Bal } \alpha & \ni & \text{BZero } \alpha \\ & & \mid \text{BSucc } (\text{BZero } (\alpha, \alpha)) \\ & & \mid \text{BSucc } (\text{BSucc } (\text{BZero } ((\alpha, \alpha), (\alpha, \alpha)))) \\ & & \mid \text{BSucc } (\text{BSucc } (\text{BSucc } (\text{BZero } (((\alpha, \alpha), (\alpha, \alpha)), ((\alpha, \alpha), (\alpha, \alpha)))))) \\ & & \mid \dots \end{array}$$

The subterms of these components all have distinct types, which get larger with each unfolding; hence there is no regular tree representing this type. Naïve usage annotation (a distinct usage annotation for each distinct subtree) would give an infinite vector of usage parameters.

It is clear that at this point our analysis must bail out somehow. Pending further results on the behaviour of such data types, on detecting non-regular recursion we bail out and use a simpler annotation scheme for the data type (in fact, $\blacktriangleright\text{-DATA-EQUAL}$), defined in Section 5.4.4.1).

5.5 Usage inference in the full language

Once again, the final part of our type-based analysis is the *inference algorithm* \mathcal{IT}_2 , which must compute the best valid $FLIX_2$ typing for any FL_0 program. That is, when presented with an FL_0 program, the algorithm must infer an equivalent $FLIX_2$ program which is well-typed according to the rules of Sections 5.2 and 5.3; and if there is more than one such, it should choose the one that is the ‘best’ in some appropriate sense.

Since $FLIX_2$ is merely an extension of the previous language LIX_2 , the inference algorithm \mathcal{IT}_2 is also an extension of the previous algorithm described in Section 4.4. Four new rules are added in order to handle the four new language constructs. Only phase 1 of the inference, \blacktriangleright_2 , is extended, as described below in Section 5.5.1. The

Figure 5.9 Type inference rules from FL_0 to $FLIX_2$ (extends Figures 4.6 and 4.7).

<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <div style="display: flex; justify-content: space-around; text-align: center;"> <div>— environment, $FLIX_2$</div> <div>— source term, FL_0</div> <div>— target term, $FLIX_2$</div> <div>— target type, $FLIX_2$</div> <div>— constraint</div> <div>— occurrence multiset</div> </div> <div style="margin-top: 10px;"> $\Gamma \blacktriangleright_2 M \rightsquigarrow e : \sigma; C; V$ </div> </div>	<p>All the rules for LIX_2 (q.v. Figures 4.6 and 4.7), with the addition of:</p>
$\frac{\Gamma, \alpha \blacktriangleright_2 M \rightsquigarrow e : \tau^\kappa; C; V \quad \alpha \notin ftv(\Gamma)}{\Gamma \blacktriangleright_2 \Lambda \alpha . M \rightsquigarrow \Lambda \alpha . e : (\forall \alpha . \tau)^\kappa; C; V} (\blacktriangleright_2\text{-TYABS})$	
$\frac{\Gamma \blacktriangleright_2 M \rightsquigarrow e : (\forall \alpha . \tau_1)^\kappa; C; V \quad \tau_2 = [t]_\tau^{fresh}}{\Gamma \blacktriangleright_2 M t \rightsquigarrow e \tau_2 : (\tau_1[\tau_2/\alpha])^\kappa; C; V} (\blacktriangleright_2\text{-TYAPP})$	
$\begin{array}{ll} \tau_k = [t_k]_\tau^{fresh} & \text{for all } k \quad \text{fresh } v, \overline{v_l} \\ \sigma_{ij}^\circ = \sigma_{ij}[v/u, \overline{v_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] & \text{for all } i \\ \Gamma \blacktriangleright_2 A_j \rightsquigarrow a_j : \sigma'_j; C_1^j; V_j & \text{for all } j \\ C_1 = \bigwedge_j (C_1^j \wedge \{\sigma'_j \preceq \sigma_{ij}^\circ\}) \\ C_2 = \bigwedge_j \{ \sigma_{ij}^\circ \leq v\} \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} \end{array}$	
$\Gamma \blacktriangleright_2 K_i \overline{t_k} \overline{A_j} \rightsquigarrow K_i^{v, v^\dagger} \overline{v_l} \overline{\tau_k} \overline{a_j} : (T \overline{v_l} \overline{\tau_k})^v; C_1 \wedge C_2; \biguplus_j V_j \quad (\blacktriangleright_2\text{-CON})$	
$\begin{array}{l} \Gamma \blacktriangleright_2 M \rightsquigarrow e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa; C_1; V \\ \sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{for all } i, j \\ \Gamma, x_{ij} : \sigma_{ij}^\circ \blacktriangleright_2 M_i \rightsquigarrow e_i : \sigma_i; C_2^i; V_i \quad \text{for all } i \\ C_2 = \bigwedge_i C_2^i \quad (C_3, \sigma) = \text{FreshLUB}(\overline{\sigma_i}) \\ C_4 = \bigwedge_{ij} \{V_i(x_{ij}) > 1 \Rightarrow \sigma_{ij}^\circ = \omega\} \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} \end{array}$	
$\frac{\Gamma \blacktriangleright_2 \text{case } M : T \overline{t_k} \text{ of } \overline{K_i \overline{x_{ij}}} \rightarrow \overline{M_i}}{\rightsquigarrow \text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}}} \rightarrow e_i : \sigma; C_1 \wedge C_2 \wedge C_3 \wedge C_4; V \uplus (\biguplus_i V_i \setminus \{\overline{x_{ij}}\})} (\blacktriangleright_2\text{-CASE})$	

pessimisation step described in Section 4.4.3 is trivially altered in the obvious way to handle type-polymorphic types and type constructors, and phase 2 of the inference remains exactly as described in Section 4.4.4. As before, the combination of these three parts yields the complete inference:

$$\mathcal{IT}_2(\Gamma, M) \triangleq (\mathcal{CS} \circ \mathcal{P}_{\text{ess}} \circ \blacktriangleright_2)(\Gamma, M) = (e, \sigma)$$

where $e : \sigma$ is the best well-typed $FLIX_2$ term corresponding to M .

Note that for the purposes of the present section we assume that all data type declarations have been annotated already, possibly along the lines suggested in Section 5.4. The inference algorithm is parameterised on these annotations.

5.5.1 Inference phase 1

The first phase of the inference algorithm, \blacktriangleright_2 , takes an FL_0 term and yields an equivalent $FLIX_2$ term, along with an appropriate constraint on its free variables. This phase is defined in Figure 5.9. The figure defines a relation

$$\Gamma \blacktriangleright_2 M \rightsquigarrow e : \sigma; C; V$$

which may be read “In the $FLIX_2$ type environment Γ , the FL_0 term M translates to $FLIX_2$ term e , which has type σ , generated constraints C , and free term variables V .”

We obtain an inference algorithm for $FLIX_2$ exactly as before (Sections 3.5.2 and 4.4.1); subsumption is restricted to the arguments of (\blacktriangleright_2 -CON) and the branches of (\blacktriangleright_2 -CASE), and the inference rules follow directly from the corresponding well-typing rules of Figure 5.6. The side condition in (\blacktriangleright_2 -TYABS) may be satisfied by α -conversion if necessary. Rule (\blacktriangleright_2 -CASE), like (\blacktriangleright_1 -IF0), makes use of the *FreshLUB* operator defined in Section 3.5.2. The result is a syntax-directed rule set, and hence an algorithm.

5.6 Proofs

Below we give the various proofs that support the full-language usage analysis. The results we prove are all analogous to those of Chapters 2 and 4, and so we simply list them without comment, referring the reader to their earlier expositions.

5.6.1 Well-typing rules

Firstly, the results about $FLIX_0$ and \vdash_0 follow those for LIX_0 and \vdash_0 in Section 2.5. \mathcal{IT}_0 (Section 2.3.3) is extended to the full language in the obvious way, setting update flags on constructors also to $!$. \mathcal{T}_0 is again defined by type erasure: $\mathcal{T}_0 = \flat \circ \mathcal{IT}_0$. The well-typing rules \vdash_0 are sound for $FLIX_0$ with respect to the operational semantics.

Lemma 5.1 (Progress)

For all $FLIX_0$ configurations C , if $\vdash_0 C : t$ then either (i) $C \in \text{Value} \cup \text{BlackHole}$ or (ii) $\exists C' . C \rightsquigarrow C'$ and $C \rightsquigarrow C' \Rightarrow \vdash_0 C' : t$.

Lemma 5.2 (Instrumented soundness for $FL_0, \vdash_0, \mathcal{IT}_0$)

For all FL_0 terms M , if $\emptyset \vdash_0 M : t$ then

- (i) $(\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle}$ is well-defined.
- (ii) $\vdash_0 (\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} : t$.
- (iii) If $(\mathcal{IT}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C'$ then $C' \in \text{Value} \cup \text{BlackHole}$.

Lemma 5.3 (Correspondence)

The functions $\langle ; ; \rangle$ and \rightsquigarrow are well-defined on FLX and $FLXC$ (as well as $FLIX_0$ and $FLIXC_0$), and make the following two diagrams commute:

$$\begin{array}{ccc}
 FLIX_0 & \xrightarrow{\langle ; ; \rangle} & FLIXC_0 \\
 \downarrow b & & \downarrow b \\
 FLX & \xrightarrow{\langle ; ; \rangle} & FLXC
 \end{array}
 \qquad
 \begin{array}{ccc}
 FLIXC_0 & \xrightarrow{\rightsquigarrow} & FLIXC_0 \\
 \downarrow b & & \downarrow b \\
 FLXC & \xrightarrow{\rightsquigarrow} & FLXC
 \end{array}$$

Theorem 5.4 (Soundness for $FL_0, \vdash_0, \mathcal{T}_0$)

For all FL_0 terms M , if $\emptyset \vdash_0 M : t$ then

- (i) $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle}$ is well-defined.
- (ii) If $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C'$ then $C' \in \text{Value} \cup \text{BlackHole}$.

5.6.2 Well-typing rules

The operational semantics we use for $FLIX_2$ has been presented in Section 5.1.4. Using this semantics, we prove the soundness and correctness results analogous to those for LIX_2 in Section 4.6. We list these below. The well-typing rules \vdash_2 are sound for $FLIX_2$ with respect to the operational semantics, and on well-typed programs the operational semantics is correct with respect to the FL_0 operational semantics. All FL_0 programs have an $FLIX_2$ typing.

Theorem 5.5 (Type soundness)

For all $e \in FLIX_2$, if $\emptyset \vdash_2 e : \sigma$ and there exists a configuration C' such that $(e)^{\langle ; ; \rangle} \downarrow C'$, then $C' \in \text{Value} \cup \text{BlackHole}$.

Theorem 5.6 (Correctness)

For all $FLIX_2$ target programs e , where $\emptyset \vdash_2 e : \sigma$, let M be the corresponding FL_0 source program $(e)^{\natural}$. Then we have

- (i) $(e)^{\langle ; ; \rangle} \downarrow \Leftrightarrow (\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow$
(i.e., the $FLIX_2$ program e terminates iff the FL_0 program M does); and
- (ii) If $(e)^{\langle ; ; \rangle} \downarrow C'$ and $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle ; ; \rangle} \downarrow C''$, then all the following hold:
 - (a) $C' \in \text{BlackHole} \Leftrightarrow C'' \in \text{BlackHole}$

- (b) $C' \in \text{Value} \Leftrightarrow C'' \in \text{Value}$
- (c) $C' = \langle H; n; \varepsilon \rangle \Leftrightarrow C'' = \langle H'; n; \varepsilon \rangle$

(i.e., if the two programs terminate, they both terminate in the same way, viz., black hole, non-ground value, or the same ground value).

Theorem 5.7 (Nonrestrictivity)

For all FL_0 environments Γ , terms M , and types t , if $\Gamma \vdash_0 M : t$ then there exists an $FLIX_2$ environment Γ' , term e , and type σ such that $(\Gamma')^\natural = \Gamma$, $(e)^\natural = M$, $(\sigma)^\natural = t$, and $\Gamma' \vdash_2 e : \sigma$.

The proof of nonrestrictivity (Theorem 5.7) is an extension of that for Theorem 3.3, but it requires a certain property of the annotation scheme used for data type declarations. While the well-typing rules ($\vdash_2\text{-CON}$) and ($\vdash_2\text{-CASE}$) are sound no matter what annotation scheme is used for each data type declaration, not all annotation schemes permit all FL_0 programs to be typed. As seen in Section 5.4.3, *typeability* requires that no 1-annotations appear anywhere in the types of the components in the data type declaration. This requirement, satisfied by all the annotation schemes of Section 5.4.4, is sufficient to establish nonrestrictivity. It is sufficient because it guarantees that the translation $(K_i \bar{\omega} \overline{[t_k]_\tau^\omega} \overline{[e_j]^\omega})$ of a constructor application will permit any use of its components, which after deconstruction by case will have types $\sigma_{ij}^\circ = [t_{ij}]_\sigma^\omega$.

5.6.3 Inference phase 1

Phase 1 of the $FLIX_2$ inference is sound with respect to the well-typing rules.

Theorem 5.8 (Soundness of inference phase 1)

For all Γ, e, σ in $FLIX_2$ such that $\Gamma \vdash_2 e : \sigma$,

- (i) $\blacktriangleright_2 (\Gamma, (e)^\natural) = (e', \sigma', C, V)$ is well defined²⁰
(i.e., the algorithm \blacktriangleright_2 is deterministic and does not fail);
- (ii) $(e')^\natural = (e)^\natural$ and $(\sigma')^\natural = (\sigma)^\natural$
(i.e., the inference algorithm merely annotates the source term, and does not alter it or its source type);
- (iii) $\forall S. \vdash^e SC \Rightarrow \Gamma \vdash_2 Se' : S\sigma'$
(i.e., all solutions of the resulting constraint are well-typed); and
- (iv) $\exists S. \vdash^e SC$
(i.e., the resulting constraint has at least one solution).

5.6.4 Inference phase 2

Phase 2 of the $FLIX_2$ inference is identical to that of LIX_2 and LIX_1 , and so the proofs already presented in Section 3.6.3 apply unchanged.

²⁰It is well defined modulo the names of fresh variables; we have already noted that we are omitting the details of fresh variable management.

5.6.5 Overall results

Combining the above results, we have that the entire $FLIX_2$ inference is sound with respect to the well-typing rules, and the inference has complexity bounded by $O(nm^2)$ under the same assumptions as previously.

Theorem 5.9 (Inference soundness)

For all M in FL_0 , if $(CS \circ \blacktriangleright_2)(\emptyset, M) = e : \sigma$ then $\emptyset \vdash_2 e : \sigma$ and $(e)^\sharp = M$.

Theorem 5.10 (Inference complexity)

If we assume that nesting of conditionals, abstractions, and case statements is limited to a constant depth, data type declarations are limited to a constant size, types annotating letrec bindings are limited to a constant size, and a linear algorithm exists for union-find, then the complexity of the inference \mathcal{IT}_2 is bounded by $O(nm^2)$, where n is the size of the program and m is the size of the largest binding group (set of expressions \bar{e}_i bound by a letrec) in the program.

Proof The proof directly extends that of Theorem 4.7, under the new assumptions on case (treated like if0) and data type declarations. \square

5.7 Related work

Most related work has already been considered, either in the relevant sections or in previous chapters. Below we address the relationship between our treatment of algebraic data types and that of Gustavsson (Section 5.7.1), and we examine work related to our definition of subtyping (Section 5.7.2).

5.7.1 Gustavsson

Gustavsson rightly observes [Gus99, §7.1] that the type system of our paper [WPJ99] is less expressive for data structures than his or that of [TWM95a]. However, this is merely a result of the particular annotation scheme used there, one equivalent to $(\blacktriangleright\text{-DATA-EQUAL})$. In the present thesis we parameterise the type system, allowing us to use different schemes with greater expressive power such as one equal to the treatment of lists by Gustavsson. He briefly suggests a general scheme for user-defined algebraic data types [Gus99, §8.2.2] which seems to be equivalent to one of the schemes we work out in detail in Section 5.4, namely $(\blacktriangleright\text{-DATA-FULL})$. This scheme is the one Gustavsson and Svenningsson describe in their later (constrained-polymorphic) analysis [GS00b] as “the most aggressive” of those listed in our earlier work [WPJ98].

Gustavsson and Svenningsson are however mistaken, we believe, in their assertion that

[I]n a system with simple usage polymorphism the usage of the spine [of a list] *must be* unified with the usage of the elements... [GS00b, §4, emphasis mine]

from which they conclude that our system cannot discover that an intermediate list has a spine used at most once but elements each used many times. In [WPJ98] we certainly rejected a system capable of just this; but in fact as we have seen in the present chapter the lack of bounded or constrained polymorphism in our type language does not prevent the use of bounds or constraints in the well-typing rules for particular syntactic forms, such as (\vdash_2 -CON) (the relevant constraint is discussed in Section 5.3.2). Thus their criticism that this is likely to “have a significant effect on the accuracy of the analysis” refers only to a system using (\blacktriangleright -DATA-EQUAL), not one using (\blacktriangleright -DATA-FULL) or one of the other annotation schemes discussed in Section 5.4.

The idea of describing the treatment of user-defined data types by providing a way to translate source data type declarations into target data type declarations appeared as a suggestion for future work in [Gus99, §8.2.2], but the parameterisation of the type system is not worked out and the idea of varying the annotation scheme does not appear.

5.7.2 Subtyping of data types

The only work addressing the problem of subtyping algebraic data types of which we were aware at the time of writing down the subtype relation of Section 5.3.4 (in [WPJ98, WPJ99]) was that of Hosoya, Pierce, and Turner [HPT98]. Their work is based on kernel F_{\leq}^{ω} [SP94], and includes the following rule for type-constructor application:

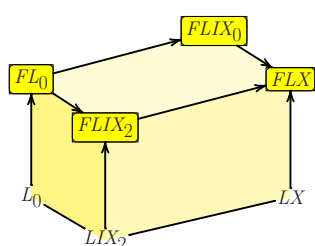
$$\frac{\Gamma \vdash R \in K \quad \Gamma \vdash T \in K \quad \Gamma \vdash_L R \preccurlyeq T \quad \Gamma \vdash_L S \preccurlyeq T \quad \Gamma \vdash_L T \preccurlyeq S}{\Gamma \vdash_L R(S) \preccurlyeq T(U)} \text{ (S-APP)}$$

This states that, *e.g.*, $C \tau \preccurlyeq D \tau'$ iff $C \preccurlyeq D$ and $\tau \simeq \tau'$; that is, subtyping applies to the *data type constructors* but not to its arguments. This is in direct contrast with the system we develop, which requires the data type constructors to be equal and subtypes the *arguments*. For a truly “full account... of the integration of data types with... subtyping” [HPT98], both systems should be combined.

Much later we encountered the work of Nordlander on O’Haskell [Nor98], discussed in Section 4.8.2. Nordlander encountered the same issue, namely how to deal with subtyping of Haskell user-defined algebraic data types, and gave the same solution [Nor98, §§3.1, 5], which he calls *depth subtyping*. His rule (DEPTH) corresponds to (\preccurlyeq -TYCON) (Figure 5.7). Nordlander also points out that (\preccurlyeq -ARROW) is subsumed by this rule; \rightarrow is just a binary type constructor whose first argument is negative and whose second is positive. A similar subtyping rule appears in [BM97] [BM96, §2], motivated by object oriented programming.

The infinite-tree model of subtyping used in Section 5.4.5 and implicitly in Section 5.3.4 originated in the seminal paper of Amadio and Cardelli [AC91, AC93], and was reinterpreted in a coinductive framework by Brandt and Henglein [BH98]. An excellent and accessible overview of the current state of the art is given by Gapeyev, Levin, and Pierce in [GLP00]. All of these algorithms apply only to the case of regular data types, and the extension to non-regular data types is an important area for future research.

Chapter 6.



Implementation

In the preceding chapters we have developed a usage analysis which is capable of handling the wide range of constructs found in real functional languages, and which we claim is powerful enough to yield good results in practice. This chapter presents evidence to support our claim. We have implemented the analysis in the Glasgow Haskell Compiler, and performed an extensive series of tests to determine its performance. The results are encouraging, but future work remains.

The implementation is a key contribution of this thesis. Firstly, we have implemented the analysis developed in earlier chapters in a *production compiler*, and *measurements* have been obtained of its effectiveness on a large set of real programs. Secondly, in scaling up the analysis we encountered a number of interesting *implementation issues*, and we discuss these below.

We begin in Section 6.1 with a brief discussion of the Glasgow Haskell Compiler, in which the analysis was implemented. After outlining the implementation in Section 6.2, we examine some design choices in Section 6.3, and then explain how the many non- FL_0 language constructs were handled in Sections 6.4 and 6.5. Section 6.6 discusses how we made use of the results of the analysis, Section 6.7 describes the ways in which we measured its performance, and in Section 6.8 we present the results we obtained. The results are examined through two case studies in Section 6.9, and we conclude in Section 6.10.

6.1 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) has been described in Section 1.2.5. We chose to implement our analysis in GHC for a number of reasons.

- GHC is a *typed* compiler, with a typed intermediate language, and so should readily support a type-based analysis.
- GHC is an *optimising* compiler, implementing most known optimisations for Haskell, and so we can easily use our analysis to guide existing optimisations as well as obtaining realistic information on the benefits of using usage analysis in conjunction with existing analyses, not just in place of them.
- GHC is designed in part as a *testbed* compiler for new analyses, and so it is structured with modularity and extensibility in mind.
- GHC is *open source*, and *well-supported* by an active development group. In fact, the author's supervisor is the chief architect, and most of the development team are nearby.

Core, the intermediate language of GHC, is essentially a System F_ω polymorphic lambda calculus with letrec and case. (The term and type languages are given at the end of the chapter, in Figures 6.13 and 6.14 respectively). Identifiers are internally annotated with types (and type variables with kinds). Variables may be identifiers or type variables, and the Lam and App productions are shared between types and terms, as in the lambda cube [PJM97, Bar92]. Case statements bind the result of evaluating the scrutinee to a variable that scopes over all the alternatives, and may have a *DEFAULT* pattern. Constructors are simply special identifiers, and are not represented in the grammar. Notes may be placed on a term at any point. Types are straightforward F_ω types with type constructors, but classes [WB89, PJJM97], implicit parameters [LSML00], and newtypes [PJH⁺99, SM01] each have a special representation.

Most of this is a reasonably close fit with the language FL_0 we have considered thus far. Points of difference will be considered as they are encountered.

6.2 Implementing the analysis

The implementation was a major effort, involving a number of important design decisions. After a chronology of the implementation, we consider the overall structure of our implementation. Specific treatment of the various constructs of Core is deferred to Sections 6.4 and 6.5.

6.2.1 A chronology

The present implementation of the analysis is the last and by far the most successful of several implementations performed over the course of this research.

Prototype. Initially, a prototype of the monomorphic analysis of Turner, Mossin, and Wadler [TWM95a] was written, and its behaviour observed for small terms. From this experience the poisoning problem became obvious, as well as the need to deal with type polymorphism and more general data types.

Monomorphic analysis. A version of the monomorphic analysis was then implemented in GHC (then version 4.02). This was the analysis described in our POPL paper [WPJ99], using a rule equivalent to (\blacktriangleright -DATA-EQUAL) for algebraic data types, and incorporating subsumption to control poisoning, and type polymorphism. The final results were extremely disappointing, however: just two thunks in the entirety of the standard libraries was annotated \bullet . The reason for this severe scaling problem was soon discovered to be the lack of usage polymorphism, as described in Sections 3.7 and 4.1.

Polymorphic analysis. To address this, the polymorphic analysis of Chapter 4 was designed and refined. A new implementation effort was embarked upon, and this time the analysis was properly wired into the type system of the compiler. The monomorphic version had simply used the intermediate language’s optional `TyNote` type annotations (see Figure 6.14) to carry usage annotations, but it was hard to ensure the annotations were correctly preserved and manipulated, and it seemed unwise to have a binder, the `usage-forall`, in such a fragile location. Directly implementing the σ - and τ -types in the compiler in place of the existing t -types meant that their correct use was statically checked, but it also meant that essentially the whole compiler had to be examined, and coercions $\llbracket \cdot \rrbracket_{\sigma}^{\omega}$, $\llbracket \cdot \rrbracket_{\tau}^{\omega}$, $\llbracket \cdot \rrbracket$, $(\cdot)^{\omega}$, and $\tau^{\kappa} \mapsto \tau$ inserted in hundreds of places. These modifications were particularly fiddly in the typechecker, where imported types (with true usage annotations that had to be preserved) jostled with user-supplied source types (with dummy usage annotations inserted to pacify the static checking). Although the analysis is defined on Core, the modification of the compiler’s data type of types had effects much more far-reaching than the intermediate language, from the renamer, typechecker, and desugarer right through to the translation to STG and interface-file generation. Classes and dictionaries, type synonyms, rules, foreign functions, and many other features had to be dealt with in detail.

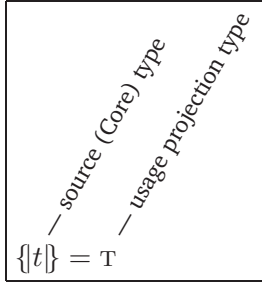
Unsurprisingly, this effort became bogged down: initial results were obtained, but many bugs remained, and extending the analysis further was out of the question. And extension seemed necessary, for the results that *were* obtained (and published in extremely preliminary form in [WPJ00]) appeared to show that, again, very little performance was being gained. This time the results were not *all* zero, but they were very nearly so, despite the encouragement of finding all the library functions being given the expected, very general usage types. Detailed analysis of the analysis output suggested that usage specialisation and finer treatment of algebraic data types should together provide better results. Another implementation was begun.

Data structures. For this implementation, intended to feature usage specialisation and general annotation schemes along the lines of Section 5.4, usage types re-

Figure 6.1 Usage projection types for GHC.

Usage schemes	$S ::= (\forall \overline{u_k}; \overline{u_l} . T)^\kappa$	
Usage-annotated types	$U ::= T^\kappa$	
Unannotated types	$T ::= U \rightarrow U$	function type
	$\mid \overline{T \kappa_l T_k}$	type constructor
	$\mid \star$	base type

Usage annotations κ and usage variables u as before.

Figure 6.2 The default projection – safely annotating with ω .

$\{t_1 \rightarrow t_2\}$	$= \{t_1\}^\omega \rightarrow \{t_2\}^\omega$	
$\{T \overline{t_k}\}$	$= T \overline{\omega \{t_k\}}$	(may be unsaturated)
$\{\alpha\}$	$= \star$	
$\{\forall \alpha . t\}$	$= \{t\}$	(type generalisation ignored)
$\{t_1 t_2\}$	$= \star$	(higher kinds beyond scope of analysis)
$\{t^{tynote}\}$	$= \{t\}$	(annotations)

mained embedded in the type language of the compiler, but a slightly closer fit to the existing code was chosen. Instead of new productions for usage lambdas, foralls, variables, and applications, usage variables were treated as type variables of usage kind. This greatly simplified the implementation, as kinded type variables are already handled correctly throughout, allowing the new features to be added. Usage specialisation was added, manually generating specialisations but using the rules mechanism (Section 6.5.3) to invoke them. Preliminary measurements showed little benefit of specialisation alone, and so we pressed on.

Adding finer treatment of data types proved to be another major alteration. Type constructors now took usage arguments as well as type arguments, not a major difficulty, but the corresponding data constructors often took different arguments, violating an assumption made many times in the compiler. Dealing correctly with this proved to be another quagmire. Correctly computing the annotations of recursive data type declarations also proved nontrivial. This implementation was never completed.

Projection types. The approach of directly implementing the σ - and τ - types of the analysis had proven too ambitious several times. For the final implementation, we decided instead to use a much simpler scheme, *usage projection types*. The final implementation is based on GHC version 5.03.20020220, and is described in the remainder of this chapter.

6.2.2 Usage projection types

The final implementation stores a *projection* of the usage information as an annotation alongside the source type; the combination of these can be used to recover the intended usage type. The impact of this system upon the rest of the compiler is much smaller (existing type-manipulation code remains unchanged), but the accuracy of the usage information is no longer enforced (so the results may be less reliable). Based on our experience, this tradeoff is a wise one.¹

As an example of usage projection types, consider the type of *map* from Figure 4.1:

$$\text{map} : (\forall u_1, u_2 . \forall \alpha, \beta . (\alpha^{u_1} \rightarrow \beta^{u_2})^\omega \rightarrow ((\text{List } \alpha)^{u_1} \rightarrow (\text{List } \beta)^{u_2})^\omega)^\omega$$

The source-type and usage-type projections of this type are as follows:

$$\begin{aligned} & \forall \alpha, \beta . (\alpha \rightarrow \beta) \rightarrow (\text{List } \alpha) \rightarrow (\text{List } \beta) \\ & (\forall u_1; u_2 . (\star^{u_1} \rightarrow \star^{u_2})^\omega \rightarrow ((\top u_1 \star)^1 \rightarrow (\top u_2 \star)^{u_2})^\omega)^\omega \end{aligned}$$

It is clear that the two representations convey exactly the same information, with only a small amount of redundancy in the the projection-type representation.

The projection types are described in full in Figure 6.1, and Figure 6.2 describes how to obtain a pessimistic usage projection, analogous to $\lceil \cdot \rceil_\tau^\omega$, from a source type. The inference as described in Chapter 5 infers only rank-1 usage polymorphism, and it is convenient to enforce this syntactically by defining a sort of *usage schemes* s . The generalised variables are divided into two classes, the *interesting* and the *boring* usage variables (defined in Section 6.3.3.3), for the benefit of the specialisation algorithm. Usage-annotated types U are as before. For unannotated types T , no type information is stored, but we duplicate the minimum amount of type structure necessary to provide a framework for the annotations. Thus for function types, the arrow remains; for type constructors we store the usage arguments and the usage-projection of the type arguments, but not the type constructor itself;² and for type variables and primitive types only a placeholder \star is stored. Type generalisation $\forall \alpha . t$ has no representation in the usage projection.

Notice that since type variables are a type-level entity and not a usage-level one, substitution of a (projection) type for a type variable in a (projection) type cannot be performed using only the projections, but must refer also to the source type, as in

¹An example of a transformation which will frequently invalidate usage information is *common subexpression elimination*, which transforms $\text{letrec } x = e \text{ in letrec } y = e \text{ in } e'$ to $\text{letrec } x = e \text{ in } e'[y/x]$.

²In fact the implementation *does* store the type constructor here also, but purely for efficiency; the external format omits it and it is reconstructed again on input.

the following:

$$\begin{aligned} (\alpha \rightarrow \delta)[\beta \rightarrow \gamma/\delta] &= \alpha \rightarrow \beta \rightarrow \gamma \\ (\star^\omega \rightarrow \star^1)[\star^1 \rightarrow \star^\omega/\delta] &= \star^\omega \rightarrow (\star^1 \rightarrow \star^\omega)^1 \end{aligned}$$

The placeholder \star is also used to represent types that are not directly handled by the analysis, such as type applications $(t_1 \ t_2)$. In GHC this requires some special handling during manipulation, as we describe in Section 6.5.1.

6.2.3 Implementing the inference itself

The inference itself essentially follows the algorithm developed in Chapters 4 and 5 and summarised in Figure B.11. It comprises around 600 lines of code for the inference itself (not including comments or blank lines), 180 lines for the specialiser, 500 lines for the constraint solver, and 1300 lines of support code. Significant code was also required for elaborating the usage annotation scheme for each data type declaration, and for interfacing the inference with the remainder of the compiler. By way of comparison, the compiler as a whole (not including the RTS *etc.*) is around 70 000 lines.

Implementation in Haskell was straightforward, using a monad to thread the fresh name supply, annotated-variable environment, constraint set, call sites map and interesting variable set (Section 6.3.3), and compiler options through the code. The monad definition is as follows:

```
newtype USPInfM a = USPInfM (UIMInp -> UIMOut a)

data UIMInp    = UIMInp { usi    :: UniqSupply,    -- unique supply in
                          sub    :: VarEnv Var,      -- unannot->annot varenv
                          subfuv :: UVarSet,        -- fuv(type(rng(sub)))
                          dflags :: DynFlags        -- command-line options
                        }
data UIMOut a = UIMOut { res     :: a,              -- result
                        uco      :: UConSet,        -- final constraints
                        uso      :: UniqSupply,      -- unique supply out
                        cso      :: CallSites,       -- callsites of annot vars
                        ivo      :: UVarSet          -- set of interesting uvars
                      }

instance Monad UspInfM where ...
```

An action in the monad takes an input fresh name supply `usi`, a substitution from unannotated to annotated bound variables `sub` (variables are annotated at their binding site, and bound occurrences share this same copy of the variable), a set of the free usage variables of the range of this substitution `subfuv` (this is $fuv(\Gamma)$, used in computing the closure), and a set of compiler flags `dflags` (used to control generalisation *etc.*). It returns a result `res`, a constraint set `uco` (constraint sets from each action are combined together and propagated upwards), the output fresh name supply `uso` (after use by the action), a set of discovered call sites `cso` and a set of interesting usage variables `ivo` (see Sections 6.3.3.1 and 6.3.3.3 respectively).

Given this, implementation was largely a matter of transcribing the inference rules of Figure B.11. The deviations almost all relate to differences between Core and the source language FL_0 studied in this thesis.

6.3 Design choices

We now consider two important design choices: the location of the inference pass in the optimiser pipeline, and the restriction of generalisation in view of the trade-off between generalisation and accurate annotation. We also discuss an additional means of addressing the difficulties of generalisation: specialisation.

6.3.1 Locating the inference

Usage inference is treated as a Core-to-Core optimisation pass, along with other major passes such as strictness analysis, specialisation, floating out, floating in, occurrence analysis, common subexpression elimination, and so on. The list of these is quite long, and sequencing can be critical. The optimum location in this list for usage inference can only be determined through experience, so our present choices can be only provisional.

Usage inference, as we shall see below, informs the rest of the compiler in two ways. Firstly, it causes generated thunks to be marked `•` or `!` by the code generator as appropriate (Sections 6.6.1). Secondly, it marks certain lambdas as used once, allowing inlining and other optimisations to be more aggressive (Section 6.6.2). For the first, it is absolutely essential that the usage information is correct at code generation time, or else the generated code might enter a black hole. For the second, usage information must be available to as many of the optimisations as possible, although its accuracy is not quite so critical.

The minimum useful approach, therefore, is to run the usage inference twice: once fairly early on in the Core-to-Core sequence, and once as late as possible, certainly after any code motion transformations. We perform the earlier inference after the desugarer, initial simplification, specialisation of overloaded operations, initial floating outwards and inwards, and next simplification have done their job. This is before two further simplifications, strictness, worker-wrapper generation, full floating outwards and inwards, and the final cleanup simplification, leaving plenty of opportunity for the results to inform the optimiser.

We also tried running the inference much more frequently, to see whether adding extra passes is worthwhile. This is controlled by the `-fusage-heavy` flag. The results appear in Sections 6.8.2 and 6.8.3 below.

6.3.2 Restricting generalisation

We observed in Section 4.7.3 that generalisation and usage analysis in fact work against each other to a significant degree. There is a tradeoff between using a most-general type so as not to poison use sites, and using a monomorphic type so as to maximise the number of thunks in the function itself that can be marked used-once. We introduced a flag `-fusagesp-moden` to investigate several different choices:

<code>-fusagesp-mode0</code>	bindings are never generalised
<code>-fusagesp-mode1</code>	only exported bindings are generalised
<code>-fusagesp-mode2</code>	only toplevel bindings are generalised
<code>-fusagesp-mode3</code>	all bindings are generalised

Based on the results obtained in Section 6.8, the default is **-fusagesp-mode1**. Another way of addressing this problem is by usage specialisation.

6.3.3 Usage specialisation

Usage specialisation is an attempt to address the problem that while generalisation of a function reduces poisoning at call sites, it greatly reduces the benefit of usage analysis in the function itself, since thunks annotated with a generalised variable must be marked updatable !. If we generate several cloned versions of a generalised function, each specialised to a specific vector of usage arguments, then a call site may choose the appropriate one and not be poisoned, and yet code may be generated for the (clone of the) function knowing that certain thunks will be used at most once and can be marked non-updatable •. Thus we hope to achieve the best of both worlds.

Usage specialisation is performed immediately after the final usage inference, just before interface files are generated and the Core program is tidied and translated into STG. The algorithm used is due entirely to Simon Peyton Jones, although the implementation is the author's.

6.3.3.1 Call sites

The algorithm is centred around a notion of *call sites*. During usage inference, whenever a usage-polymorphic term variable v_{orig} is instantiated with a fresh vector $\overline{u_l}$ of usage variables by rule (\blacktriangleright_2 -VAR), the occurrence of the variable is replaced by a fresh, uniquely-named clone v_{site} of the variable and the triple $(v_{orig}, v_{site}, \overline{u_l})$ is recorded (recall that in the projection-based implementation, usage applications are not recorded in the term itself, so without this step this information would be lost). The variable binding for v_{orig} is left untouched. For example,

$$\begin{aligned} \text{let } g : (\forall u_1, u_2 . \text{Int}^{u_1} \rightarrow (\text{Int}^{u_2} \rightarrow \dots)^{u_1})^\omega = \dots \\ \text{in } \dots g \ 6 \ 7 \dots g \ (1 + 2) \dots \end{aligned}$$

where the first occurrence of g is instantiated at $\langle u_3 \ 1 \rangle$ and the second at $\langle \omega \ 1 \rangle$, is transformed to

$$\begin{aligned} \text{let } g : (\forall u_1, u_2 . \text{Int}^{u_1} \rightarrow (\text{Int}^{u_2} \rightarrow \dots)^{u_1})^\omega = \dots , \quad \{(g, g', \langle u_3 \ 1 \rangle), (g, g'', \langle \omega \ 1 \rangle)\} \\ \text{in } \dots g' \ 6 \ 7 \dots g'' \ (1 + 2) \dots \end{aligned}$$

Once inference is complete and the final usage substitution decided upon, this substitution is applied to the set of triples as well as to the term; for example, if $S(u_3) = 1$ then the set above becomes $\{(g, g', \langle 1 \ 1 \rangle), (g, g'', \langle \omega \ 1 \rangle)\}$.

6.3.3.2 Using specialisations

We now have access to the actual usage arguments passed to each instance of every variable used in the program. If usage specialisation is disabled, we may restore the program to coherence by using the set of call-site triples to substitute the correct original variable for each site variable. However, if for a particular v_{site} the corresponding v_{orig} has one or more usage specialisations available, it may be the case

that one of them matches the particular vector of usage arguments used at this site (if more than one specialisation match, the best one – that with the most 1s – is chosen). If there is a match, the occurrence of v_{site} is replaced by an occurrence of the appropriate v_{spec} . In our example, if there is a version g_1 of g specialised to the arguments $\langle 1\ 1 \rangle$, the call-site set contains enough information to compute the substitution $S = \{g' \mapsto g_1, g'' \mapsto g\}$, which makes use of the specialisation for the first call site and the original version for the second.

6.3.3.3 Interesting variables

Not all usage parameters of a usage-generalised function are worth specialising. Only those that will lead to the generation of a used-once thunk are worth the effort of duplicating the code for the function body. We therefore partition the parameters of a usage type scheme s into *interesting* and *boring* variables.

- *Interesting* usage variables are those that appear as the topmost annotation on the type of a let-binding, or as the topmost annotation of the type of an argument that will likely be let-bound by the A-normal form conversion, or as an interesting argument to another function.
- *Boring* usage variables are those that are not interesting.

It is only the interesting portion of the vector of usage arguments supplied at a call site that we record in the call-site triple and consider further, and it is only the interesting usage variables that we consider generating specialisations for. Furthermore, we only consider specialising them to 1 – for the purposes of execution, and hence of choosing a specialisation to use, the constant usage ω and a variable usage u are equivalent, requiring in both cases an updatable thunk to be generated.

6.3.3.4 Generating specialisations

It remains to *generate* the specialisations we have used above. Certainly some investigation is needed into how many and which specialisations should be generated, and for which variables. For the present, we generate one specialisation for every top-level variable that stands to benefit from it, *i.e.*, that has at least one interesting usage argument. The specialisation we generate is simply the one that sets all (interesting) usage arguments to 1. We have not tried the alternative of generating all possible specialisations, which would appear likely to lead to an exponential code explosion; several people have suggested to us that in practice the number of specialisations would be manageable.

A specialisation is generated as follows. First we create a fresh clone of the original variable, and give it a new name: f at $1, 1, u_3, 1$ (*i.e.*, with arguments 1, 2, and 4 forced to 1 and argument 3 left free) would be named $\$U11u1f$. Then we make a copy of its right-hand side, making the appropriate substitutions throughout

for the arguments we are instantiating:

$$\begin{aligned}
 f & : (\forall u_1, u_2, u_3, u_4 . \tau)^\omega \\
 f & = \Lambda u_1, u_2, u_3, u_4 . e \\
 \$U11u1f & : (\forall u_3 . \tau[1/u_1, 1/u_2, 1/u_4])^\omega \\
 \$U11u1f & = \Lambda u_3 . e[1/u_1, 1/u_2, 1/u_4]
 \end{aligned}$$

Thunks in f annotated by u_1 , u_2 , or u_4 will be given the annotation 1 in $\$U11u1f$.

A subtlety is that specialisations may *cascade*; that is, call sites in the right-hand side of a specialisation may themselves be able to be specialised due to being passed arguments that have been instantiated: if f invokes g at $\langle u_1 \ u_2 \rangle$ then $\$U11u1f$ should invoke $\$U11g$ instead. We take special care when generating a clone body to select specialisations for call sites based on an extended substitution incorporating the actual known values of the usage arguments.

A further subtlety is that if a specialisation is required for a function in a mutually-recursive binding group, we must in fact specialise the entire group as one, replacing recursive calls in the specialised RHSs with references to the specialised variables. Since the resulting group is mutually recursive within itself but not with the original group, the two groups may be added to the list of binds (Section 6.4.1) in either order.

6.3.3.5 Discussion and related work

To guide future investigations into which specialisations should be generated, we generate trace output noting when a specialisation is generated, when a specialisation is used, and when a specialisation could usefully have been used had it been available.

It is important that specialisations are only generated and used at the very end of the optimisation cycle, when there is no chance of the terms being manipulated in any way that might invalidate the results of the analysis. It is not allowed even to run the usage analysis again afterwards, as the right-hand sides of specialised versions differ from those of the original functions only in usage annotations, and these are thrown away when re-inferring – the result would be two identical functions with identical, fully-general usage type schemes, not the desired behaviour at all!

Interestingly, Goldberg, presenting the earliest usage analysis of which we are aware (Section 1.3.5) advocates specialisation, generating a different set of super-combinators for each application of the function. He believes that

[a]lthough this creates a potential for code explosion, it is probably a reasonable thing to do, for two reasons. First, the sharing properties typically do not vary much, and thus code explosion is not a problem. Second, in some sense it is unreasonable to penalize a programmer's use of a function in one place because of a use of the same function somewhere else. [Gol87, p. 424]

Gustavsson also proposes usage specialisation (“annotation polyvariance”) as future work, in [Gus98, §11.1] [Gus99, §8.2.5], as do Turner *et al.* [TWM95a, §4.5]. We however believe we are the first to actually implement it.

6.4 Inference for Core constructs

There are a number of constructs in Core that go beyond the FL_0 language we have considered in previous chapters. In implementing the analysis in GHC we have had to consider how to perform inference for all of these. Many of these constructs are likely to occur in other compiler intermediate languages, and so we describe our solutions in this section. Those that seem to us more GHC-specific are deferred to Section 6.5.

6.4.1 Binds

A Core let expression has the form `let bind in e`, where *bind* is either a nonrecursive binding for a single variable, or a set of mutually recursive bindings for one or more variables. Binds are broken into strongly-connected components and nested in dependency order on translation into Core from Haskell, and the simplifier acts to ensure these properties hold throughout compilation. The inference rule for nonrecursive let is trivially derived from that for recursive `letrec`.

6.4.2 Modules

A *module* is simply a list of bindings b_1, b_2, \dots, b_n which are interpreted as nested in sequence: `let b_1 in let b_2 in ... let b_n in $[\cdot]$` . A program is made up of one or more modules, and the bindings from each are concatenated; the hole $[\cdot]$ of the resulting expression is filled by the variable *main* from module *Main*, yielding a Core expression to be evaluated. Apart from allowing the code of a program to be divided into separate files, the only significance of modules is that each has a separate namespace, and provision is made in the source language for hiding and exposing names from other modules.

6.4.3 Unsaturated constructors

In Core it is not necessarily the case that constructor applications are saturated, although the action of the simplifier ensures that they often are. For the present we do not attempt to yield sensible results for unsaturated constructor applications, but instead return the safe but uninformative default projection of the returned source type, with ω -annotations everywhere (Figure 6.2).

6.4.4 Beyond A-normal form

Expressions in Core are not restricted to A-normal form (this was relaxed in GHC version 4.01), and so constructor and application arguments need not be atomic. When Core is converted into STG prior to code generation, nontrivial arguments are bound by fresh let expressions, thus converting to A-normal form (in fact, arguments that are demanded strictly are bound by case expressions for efficiency).

Usage inference is not affected by this: in rules (\blacktriangleright_2 -APP) and (\blacktriangleright_2 -CON), the recursive inference of the arguments can as well be of *e* as of *a*. But it is critical

that the fresh let bindings are given accurate usage annotations, since very often it is these anonymous arguments (the thunk for $(g\ x)$ in $f\ (g\ x)$, for example) that are used just once. Thus when inferring an application of either a constructor or some other function, we annotate each nontrivial argument with a new *Note*, a *TopUsage*, containing the topmost usage of that argument. This is inspected by the A-normal form converter and used to give the fresh let binder a usage projection type of \star^κ , where κ is the annotation from the *TopUsage* note. In fact STG requires the *function* (if not a constructor) to be atomic also, but in this case inspection of the type rule for $(\vdash_2\text{-APP})$ reveals that the function is *always* used at most once. By building this into the converter (*i.e.*, always using the projection \star^1 for such binders) we avoid having to place a *TopUsage* note on nontrivial functions.

6.4.5 Case expressions

For case expressions, Core adds an optional *DEFAULT* pattern which matches if none of the other patterns match, and binds the result of evaluating the scrutinee to a variable that may be used in any branch of the case:

$$\begin{array}{l} \text{case } e_0 \text{ of } x_0 : t_0 \langle \\ \quad \text{DEFAULT} \rightarrow e' ; \\ \quad K_i \overline{x_{ij}} \rightarrow e_i ; \\ \quad \dots \rightarrow \dots ; \\ \rangle \end{array}$$

To accommodate this, we treat *DEFAULT* as a nullary constructor, and occurrences of x_0 in the alternatives are counted as additional uses of the scrutinee. If x_0 is used more than once, including its implicit use for deconstruction by the case statement itself, then the topmost annotation of the type of x_0 (and the scrutinee) must be ω . We omit the full rule.

A special case of this form gives Core a form of strict let, known as *polymorphic case*. This is a case expression of the form

$$\begin{array}{l} \text{case } e_0 \text{ of } x_0 : t_0 \langle \\ \quad \text{DEFAULT} \rightarrow e; \\ \rangle \end{array}$$

where e_0 can have any type at all, even a polymorphic or function type. The operational meaning is to evaluate e_0 to WHNF and then bind the result to x_0 in e . The inference rule for this construct is

$$\frac{\begin{array}{l} \sigma_0 = \lceil t_0 \rceil_\sigma^{\text{fresh}} \\ \Gamma \blacktriangleright_2 M_0 \rightsquigarrow e_0 : \sigma'_0; C_1; V_1 \\ C'_1 = C_1 \wedge \{\sigma'_0 \preceq \sigma_0\} \\ \Gamma, x_0 : \sigma_0 \blacktriangleright_2 M \rightsquigarrow e : \sigma; C_2; V_2 \\ C_3 = \{V_2(x_0) + 1 > 1 \Rightarrow |\sigma_0| = \omega\} \end{array}}{\Gamma \blacktriangleright_2 \text{case } M_0 \text{ of } x_0 : t_0 \langle \text{DEFAULT} \rightarrow M \rangle \rightsquigarrow \text{case } e_0 \text{ of } x_0 : \sigma_0 \langle \text{DEFAULT} \rightarrow e \rangle : \sigma; C'_1 \wedge C_2 \wedge C_3; V_1 \uplus (V_2 \setminus \{x_0\})} \quad (\blacktriangleright_2\text{-POLYCASE})$$

where even though no deconstruction is performed on the scrutinee, we must still count one demand for the case statement itself, since at runtime it makes the demand that reduces it to WHNF. Note that here and in the general form of case above, the type σ_0 on the scrutinee binder refers only to the usage made of x_0 itself by the case body; this will usually differ from the usage made of scrutinee, as recorded by rule (\blacktriangleright_2 -CASE) in Figure 5.6.

Two further Core features make case statements interesting: primitive types and existential constructors. A case performed at a primitive type rather than an algebraic one has literals as patterns; these are treated as nullary constructors and the primitive type constructor is treated as a nullary algebraic data type constructor. Existential constructors are treated in Section 6.5.4.

6.4.6 Type coercions

To support Haskell newtype declarations,³ and the *bottom-propagation* transform that floats upwards *bottoming expressions* such as (*error* “fail”) that always yield \perp , e.g.,

$$\text{case } \text{error} \text{ “fail” of } \text{alts} \implies \text{error} \text{ “fail”}$$

in a type-safe way, the Core language includes *coercions*. The expression

$$\text{Note } (\text{Coerce } t_{\text{to}} \ t_{\text{from}}) \ M$$

represents the term M , but treated as if it had type t_{to} , even though it actually has type t_{from} . Such coercions break the flow of usage information for the present type-based analysis, since they direct the compiler to ignore the existing type for an expression and use instead an arbitrarily unrelated one. We must therefore handle them carefully to avoid introducing unsoundness.

The overall (topmost) usage annotation can safely be propagated over the coercion, since it is not dependent on the structure of the term’s type. The outside type (t_{to}) is annotated everywhere with ω : the positive annotations since we have no reason to restrict the term’s use, and the negative annotations since we do not know how the term will use its arguments. The inside type (t_{from}) is pessimised, since we do not know how it will be used, but the negative annotations need not be touched (and indeed should not be, since this would break the property that result types are only ever constrained via the subtype relation, see footnote 13 on page 99). Thus the inference rule is

$$\frac{\begin{array}{l} \Gamma \blacktriangleright_2 M \rightsquigarrow e : \tau_{\text{from}}^{\kappa}; C_1; V \\ C_2 = \text{Pess}^+(\tau_{\text{from}}) \\ \tau_{\text{to}} = \lceil t_{\text{to}} \rceil_{\tau}^{\omega} \end{array}}{\Gamma \blacktriangleright_2 \text{Note } (\text{Coerce } t_{\text{to}} \ t_{\text{from}}) M \rightsquigarrow \text{Note } (\text{Coerce } \tau_{\text{to}}^{\kappa} \ \tau_{\text{from}}^{\kappa}) e : \tau_{\text{to}}^{\kappa}; C_1 \wedge C_2; V} \ (\blacktriangleright_2\text{-COERCE})$$

³Coercions are in fact used only for recursive newtypes; the mechanism for non-recursive newtypes is described in Section 6.5.1.

6.4.7 Unboxed data types

In GHC, primitive data types are *unboxed*; this means that they can reside in registers and on the stack, but cannot reside in the heap because they have no tag to identify them. This means that an unboxed value can never be represented by a thunk; its value is always computed strictly. In consequence, usage annotations on such types are irrelevant: an unboxed value is never entered, so whether it is used at most once or not is irrelevant. We build this into our implementation of the subtyping relation, by simply ceasing comparison whenever a type of the form $(T \dots)^\kappa$, T unboxed, is encountered.

In fact the system in GHC is a little more complicated. A type is *unlifted* if it does not have \perp as an element. All unboxed types are unlifted; but some types are boxed but unlifted, for example byte arrays, which must live in the heap for space reasons but are otherwise just values. For these too we ignore usage annotations. On the other hand, unboxed pairs are themselves unlifted but may contain lifted objects inside. Ignoring these usage annotations would be therefore be unsound. We define a *deeply unlifted type* to be one which does not contain an object of lifted type, *i.e.*, that cannot be or contain a pointer to a thunk.

In one specific case we must cheat. A *stable pointer* gives an integer name to a Haskell pointer. The name is itself unboxed, and may be freely passed in and out of the Haskell system, yet it can be dereferenced to obtain the original Haskell pointer. Since we cannot control the usage of an unboxed integer (indeed, it may be duplicated outside the Haskell system entirely), the only way to preserve soundness is to define stable pointer creation in such a way as to ensure that all objects having stable pointers are annotated ω . This allows us to treat stable pointers as deeply unlifted.

6.4.8 Primops

The world may rest on the back of four elephants, standing on an infinite chain of turtles each supporting the one above,⁴ but compilers cannot rely on such infinite regress. The fundamental operations of arithmetic and logic, as well as interaction with other programs and the outside world, are implemented by *primitive operations*, or *primops*, the irreducible building blocks from which programs are built by means of functional glue. These primops mediate between the typed functional intermediate language on one side and the untyped imperative implementation language on the other; each has a type describing its low-level imperative behaviour in high-level functional terms. Since the type system now includes usage annotations, we must extend these to include correct usage projection types for each operation.

For the vast majority of primops, usage annotation is trivial. Primops are al-

⁴“A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: ‘What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise.’ The scientist gave a superior smile before replying, ‘What is the tortoise standing on?’ ‘You’re very clever young man, very clever,’ said the old lady, ‘but it’s turtles all the way down!’” [Haw88, p. 1].

ways saturated (partial applications are handled by providing eta-expanded wrapper functions), and usually take unlifted arguments and yield unlifted results. Thus their types are of the form

$$+\# : (\text{Int}\#\mathbf{X} \rightarrow (\text{Int}\#\mathbf{X} \rightarrow \text{Int}\#\mathbf{X})^1)^\omega$$

where \mathbf{X} stands for a don't-care annotation (see Section 6.4.7), in practice replaced by 1 in negative positions and ω in positive ones.

The exceptions are primops that handle lifted arguments: the mutable array operations, mutable variables, MVars, exception raising, catching, blocking, and unblocking, weak and stable pointers and stable names, fork and kill, *seq#* and *par#*, *touch#*, and *dataToTag#* and *tagToEnum#*. For these, sensible annotations must be provided: if the primop may demand the argument more than once, or store it such that it may later be retrieved, then the argument must be annotated ω or usage polymorphism used to ensure the usage propagates appropriately; if it enters its argument at most once, or not at all (e.g., *sameMutVar#*), then it should be annotated 1.

The statefulness of some primops can present traps for the unwary: one of our earlier implementations gave *writeMutVar#* the incorrect type

$$\text{writeMutVar}\# : (\forall u . (\text{MutVar}\# s a)^u \rightarrow (a^u \rightarrow ((\text{State}\# s)^1 \rightarrow (\text{State}\# s)^\omega)^1)^1)^\omega$$

in an attempt to unify the usage of the object with the usage of the *MutVar#*. This type is unsound, because all occurrences of u are in negative positions, meaning that it is equivalent to

$$\text{writeMutVar}\# : ((\text{MutVar}\# s a)^1 \rightarrow (a^1 \rightarrow ((\text{State}\# s)^1 \rightarrow (\text{State}\# s)^\omega)^1)^1)^\omega$$

which has as one of its instances the clearly incorrect type

$$\text{writeMutVar}\# : ((\text{MutVar}\# s a)^\omega \rightarrow (a^1 \rightarrow ((\text{State}\# s)^1 \rightarrow (\text{State}\# s)^\omega)^1)^1)^\omega$$

(this would allow an object of type a^1 , permitted to be used at most once, to be written into a *MutVar#* permitted to be used many times). This error was only detected during testing. The correct type has ω -annotations on both *MutVar#* and a , pessimistically ensuring soundness.

6.4.9 Variance analysis

The variances (co- and/or contra-) of the usage and type arguments of each data type constructor are computed when processing the declaration, and stored with it for later use by the subtyping algorithm. The algorithm computes the fixpoint of the equations in Figure 5.8 by iteration, as described in Section 5.3.4.3. A few primitive types have the variance of their arguments hardwired into the compiler.

6.4.10 Unnecessary constructor usage arguments

In Core, data type constructors are merely special identifiers, and do not have their own production in the grammar. Constructor arguments, both types and terms, are

supplied by normal function application. For the usage analysis, we add a vector of usage arguments to these, representing both the topmost annotation κ and the type constructor arguments $\bar{\kappa}$. Thus the FL_2 constructor instance $K_1^{u_1} 1 u_1 \text{Int } (1 + 2)$ would be represented in Core by $K_1 \langle u_1 1 u_1 \rangle \text{Int } (1 + 2)$. In fact, the final implementation elides usage applications and abstractions from terms and types, but they are still implicit in the inference design.

The full story is a little more complicated, however, since we want to minimise the number of usage arguments to each constructor for efficiency reasons (to minimise the number of usage variables introduced at each instantiation site, and to minimise the size of printed terms). Firstly, the initial usage argument may be omitted if the constructor has no other usage arguments; second, only the tycon usage arguments that are relevant to this particular constructor need be supplied. Consider the data type of lists:

$$\text{data } (\text{List } u_1 \alpha)^u = \text{Nil} \mid \text{Cons } \alpha^{u_1} \text{List } u_1 \alpha^u$$

There is no need to write $\text{Nil } u u_1 \alpha : (\text{List } u_1 \alpha)^u$; simply $\text{Nil } \alpha : (\text{List } \omega \alpha)^\omega$ is sufficient. The constructor is manifestly a value containing no arguments that might not be values, thus its usage is irrelevant and may be assumed ω , and the u_1 argument restricting the usage of elements of the list is also irrelevant because there are no elements.

If usage variables were supplied for these arguments they would simply assume whatever values were required by the context; we therefore let them take the most general values, here ω and ω because both occur only positively. For any data type declaration, the topmost annotation is required always to occur only positively, and so is always ω if omitted; the remaining annotations may occur with any variance and should be given appropriate values when omitted: ω if positive, 1 if negative, either if nonvariant, and never omitted if bivariate.

Recall from Section 5.4.3 that only variable and ω annotations are permitted anywhere in the $\overline{\sigma_{ij}}$ of the data type declaration. Thus if a constructor has no usage arguments other than the topmost, it follows that all relevant annotations are ω . In this case, the constructor cannot contain any single-entry thunks, and so may be freely duplicated; thus there is no need to restrict its usage and the topmost annotation may be omitted also.

In fact the implementation so far makes the (false) assumption that all usage arguments are positive, and thus assumes ω everywhere for irrelevant arguments. This can cause the result types of case expressions to be excessively restrictive, as the lub is taken of each branch; it is not unsound but can lead to untypeability of programs involving operations such as primops lacking maximally-applicable types. Removing this assumption would be straightforward.

6.4.11 Elaborating data type annotation schemes

Elaborating data type annotation schemes and computing the correct vector of usage arguments for each type and data constructor is not trivial in the presence of mutually-recursive groups of declarations.

The algorithm proceeds as follows. For each argument of each constructor of

each data type in each group, an *annotation category* is chosen. This is one of the following, where u is the overall-usage annotation of the data type:

Category	Operation on argument t
<i>AnnMany</i>	$\lceil t \rceil_\sigma^\omega$
<i>AnnOver</i> *	$(\lceil t \rceil_\tau^\omega)^u$
<i>AnnOver</i> n	$(\lceil t \rceil_\tau^{\text{fresh}})^u$, n fresh usage variables required
<i>AnnFresh</i> *	$(\lceil t \rceil_\tau^\omega)^{u_1}$, u_1 fresh
<i>AnnFresh</i> n	$\lceil t \rceil_\sigma^{\text{fresh}}$, $n + 1$ fresh usage variables required

That is, *AnnMany* places ω everywhere on the type of the constructor argument; *AnnOver* uses the overall-usage annotation on top of the argument and optionally annotates deeply inside it; and *AnnFresh* uses a fresh annotation on top of the argument and optionally annotates deeply inside it.

Occurrences of type constructors in the same recursive group are not annotated with fresh usage variables, but with a recursively-supplied copy of the same vector as that computed for the whole group. If they occur at the top of a constructor argument they should be given the category *AnnOver* 0 so that the topmost usage of the recursive instance is the same as the usage of the original.

The four annotation schemes we provided are as follows:

-fusagesp-dconmode1 Use *AnnMany* for all arguments.

Corresponds to (►-DATA-MANY).

-fusagesp-dconmode2 Use *AnnOver* * for all arguments.

Corresponds to (►-DATA-EQUAL), as used in [WPJ99].

-fusagesp-dconmode3 Use *AnnOver* 0 for directly (mutually) recursive arguments, *AnnMany* for deeply-unlifted arguments, and otherwise *AnnFresh* n as appropriate. Corresponds roughly to (►-DATA-ALL-BAD) (Section 5.4.2), but well-defined.

-fusagesp-dconmode4 As **-fusagesp-dconmode3**, but if n is greater than a limit m set by **-fusagesp-dconmaxcount** m , use *AnnFresh* * instead of *AnnFresh* n . With $m = 0$ corresponds to (►-DATA-FULL), and with other settings allows some control over the number of annotations used. With $m = 10$ we refer to this as (►-DATA-ALL).

Control is essential: with **-fusagesp-dconmode3**, the dictionary data type for class RealFloat is given 1780 usage arguments, and a data type in one of the libraries (Edison’s Assoc.FiniteMap) was given 8941! Such large numbers of usage variables greatly slow down the inference, presumably without commensurate impact on the effectiveness of the analysis. By using **-fusagesp-dconmode4** with **-fusagesp-dconmaxcount**10, the default settings, these are reduced to just 57 and 5 respectively, with a consequent dramatic improvement in inference time.

Once the annotation category has been chosen for each constructor argument, the number of usage arguments required for the whole constructor can be computed. Each constructor then computes its initial and final variable indices in the vector for

the whole group: if it is the first constructor of the first data type in the group, the initial index is zero; if it is the first constructor of a data type other than the first, the initial index is one greater than the final index of the final constructor of the previous data type; otherwise the initial index is one greater than the final index of the previous constructor. In all cases the final index is simply one less than the sum of the initial index and the number of usage arguments for this constructor.

The final index of the final constructor of the final data type in the group is then examined, and a vector of the appropriate length passed in recursively to the constructors. The constructors each select out the appropriate subvector to use in annotating their arguments, and elaboration is complete.

The recursion involved is a little tricky, but in a lazy language quite manageable. Care must be taken not to examine the supplied usage arguments too early; it turned out to be easiest to obtain the fresh variable counts required by the annotation category in a separate counting pass, rather than attempting to perform just a single pass over the arguments. This should not be too inefficient relative to the remainder of the compiler.

6.4.12 Intermodule analysis

The Glasgow Haskell Compiler performs separate compilation of modules, using *interface files* to convey type and pragma information from each module to those importing from it. The pragmatic information includes information on strictness, arity, specialisations, unfoldings, rules, and so on.

To support separate compilation in the usage analysis, little more is required than to pessimise functions that are exported as described in Section 4.4.3 and to add usage projection types to the pragmatic information attached to binders in interface files. Care must be taken that all binders appearing in the interface file are pessimised, not just those the user has declared as exported. Identifiers occurring in unfoldings (function bodies, used to perform cross-module inlining) may be inlined and then the call context transformed arbitrarily during optimisation of the importing module, potentially leading to usage different from that appearing in the exporting module; similarly for identifiers appearing in the right-hand side of a rule.

On the other hand, pessimisation is to be avoided where possible, and so the number of exported binders should be kept to a minimum. By default in Haskell, if an export list is not declared explicitly all top-level binders are exported; we deviated from this for module *Main*, exporting by default only the binding *main*. This improved the performance of the analysis on a number of single-module benchmarks, by not pessimising auxiliary functions that were never actually used by another module. It is important to remember that to get the best performance from the usage analysis, export lists should be regularly pruned to the minimum necessary.

6.5 Inference for GHC-specific constructs

A number of the constructs we dealt with are GHC-specific. The interested reader may discover how we dealt with these below; others may skip to Section 6.6.

6.5.1 Substitution and Type invariants

The data type of Types in Core is not a free algebra. Certain type applications (functions and newtypes) have special forms when saturated, and the integrity of the compiler depends upon these special forms being used when they are applicable. It is therefore necessary to take extreme care when performing operations such as substitution that may cause type applications to become saturated, as the shape of the type may change: we must ensure the shape of the projection changes to match. Consider the following two examples:

$$\begin{aligned} (\alpha \gamma)[(\rightarrow) \beta / \alpha] &= \beta \rightarrow \gamma \\ \star[\star / \alpha] &= \star^\omega \rightarrow \star^\omega \\ \\ (\alpha \beta)[\text{ST } \sigma / \alpha] &= \text{State} \# \sigma \rightarrow (\# \text{State} \# \sigma, \beta \#) \\ \star[\star / \alpha] &= (\text{T } \star)^\omega \rightarrow (\text{T } \omega \omega (\text{T } \star) \star)^\omega \end{aligned}$$

In the first example an application of the function type constructor $(\rightarrow) \beta$ becomes saturated and turns into an arrow type $\beta \rightarrow \gamma$, and in the second an application of a nonrecursive newtype constructor becomes saturated and is expanded out.⁵ That this is the desired behaviour is not immediately obvious!

The general rule is this (reverting to the notation of Figure 6.14): if the function of an AppTy is substituted with a TyConApp, and the tycon of the TyConApp is FunTyCon, and there would now be two arguments, then it becomes a function type, FunTy. If the tycon is a *nonrecursive* newtype type constructor and it would now become saturated, then it becomes transparent and we must use its expanded form. In both cases, the new, unknown usage annotations are assumed safely to be ω . Otherwise we simply add the new argument to the list of arguments of the TyConApp.

6.5.2 Storing usage projection types

It is sufficient for the purposes of the analysis to store usage projections only of the types of all identifiers; we do not store usage projections of other types, such as those appearing in type applications. In GHC an identifier is represented by a rather complicated record, into which we added an extra field of UsageInfo. This comprises three parts: the *usage signature*, or usage scheme projected from the type of the variable; the *lambda usage*, the usage annotation of the lambda binding this variable, if this variable is bound by a lambda; and any available *usage specialisations* for this variable.

It turns out to be much more convenient to record the usage of the lambda on the variable it binds, rather than on the lambda. In many places the compiler takes apart a function by collecting its value and type binders and returning these in a list along with the remaining body; the original term can be reconstructed simply by inserting appropriate value and type lambdas, since the binders all carry their own type or kind as an annotation. By recording the lambda usage in the binder also, we preserve

⁵In fact the type becomes SourceTy (NTy ST $[\sigma, \beta]$). We show instead the expansion of this into a normal type, since usage projections look through SourceTys when determining the shape of a type.

the ability to reconstruct a function from its body and a list of binders, without any additional information. Whenever we want to inspect the topmost annotation of a lambda, it is relatively simple to descend an extra node into the term to inspect the appropriate info field of the lambda's binder.

The list of specialisations of a variable (Section 6.3.3.4) is stored as part of the usage info, as a list of pairs of usage argument vectors and corresponding specialised-variable names, and this is written out to the interface file for use by importing modules; an occurrence in this list is sufficient to cause the signature of each mentioned specialised variable to be exported in the interface file.

6.5.3 Rules

A novel feature of the Glasgow Haskell Compiler is *rules* [PJTH01]. These allow the programmer to specify rewrites to be applied to the program by the simplifier. A rule has the form

$$\text{“rule name” (after phase } p\text{): } \forall \alpha \beta \dots \gamma x y \dots z . f \ e_1 \ \dots \ e_n \Longrightarrow e'$$

where $\alpha, \beta, \dots, \gamma$ are type variables, x, y, \dots, z are term variables, $(f \ e_1 \ \dots \ e_n)$ is the LHS or rule head made up of the specialised identifier f and the template arguments e_i , and e' is the RHS. If the simplifier is running in phase $p' > p$ and encounters an application of f to at least n arguments, and the first n arguments can be unified with the e_i (given the type and term variables specified), then the instance of the LHS is replaced by the corresponding instance of the RHS. Amongst other things, this feature can be used to implement the *foldr/build* rule of Gill *et al.* [GLPJ93, Gil96].

Earlier attempts at a usage analysis in which types carried usage information directly had to handle rules with some care, to ensure that usages matched sufficiently often that rule matching was not adversely affected, and to ensure that usage typing was preserved after a rule firing. The present implementation is not so ambitious; after simplification the result is a different program, and usage analysis of the program before simplification provides no guarantees that usage information in the program afterwards remains correct.

Since rules are merely guides to the simplifier, and do not themselves generate code, it is not necessary to perform usage inference on the rules themselves. However, to guarantee typeability in the presence of separate compilation, we must allow for occurrences that may be introduced by rules when computing variable occurrences. To this end, we consider all the rules within the scope of all the binds in scope in the current module, and count the free variable occurrences in the right-hand side. From this we subtract the occurrences on the left-hand side, and delete any occurrences of the parameter variables; this is then doubled (equivalent to multiplying by ω) and added to any other occurrences in scope for the purposes of computing whether or not to annotate a variable on top with ω .

6.5.4 Existential constructors

An existential data constructor [OL96] has type arguments as well as value arguments; these can simply be ignored when inferring the usage type of the constructor arguments, since they are statically unknown and hence approximated by \star .

6.6 Using the results of the analysis

The results of the usage analysis are used to guide the compiler in two ways: to tell the code generator when to avoid generating code to update a thunk, and to tell the simplifier (and other optimisation passes) when a lambda is applied at most once.

6.6.1 Update avoidance

Using the results for update avoidance is straightforward. The STG machine has three kinds of closure: re-entrant, updatable, and single-entry. Re-entrant closures are used for functions, which are already values and are never updated. The remaining two are for thunks: updatable thunks push an update frame when they are entered and thereby update themselves when evaluated, whereas single-entry thunks do not. We must simply mark all non-reentrant, used-1 thunks as single-entry, and the runtime system will do the rest.

Achieving this is a matter of a few lines in `CoreToStg`, the module that translates from A-normal-form-converted Core into STG code. When a closure is created, we test if it should be marked re-entrant. If not, but the binder has a usage projection type attached with topmost annotation 1, and the closure is not at the top level, we mark it single-entry; otherwise, we mark it updatable. The RTS appears to have some difficulty with single-entry CAFs (top-level thunks), and so we avoid generating them. The cost of this omission should not be significant, as there can only be a small finite number of CAFs in a program (unlike dynamically-allocated thunks, of which an unbounded number may be created).

This pass depends on the modification already discussed (Section 6.4.4) to the A-normal-form converter, ensuring that binders created in this process bear the correct usage information.

6.6.2 Informing the simplifier

Santos' thesis [San95] lists several optimisations that can benefit from usage information. In almost all cases, the actual information that is required is the number of times a particular lambda expression is applied; *i.e.*, the usage annotation on the lambda. As we have seen, the latter is stored in the binder as part of the *lambda usage* portion of the usage information. To enable this information to propagate to the optimiser (primarily the simplifier), we provide a boolean function *isOneShotLambda* that tests whether a binder is bound by a lambda that is used (applied) at most once. This function is used by the optimiser in the following places:

Eta-expansion. When the simplifier is eta-expanding an expression, it takes care

not to duplicate work: let $z = e$ in $\lambda x . e'$ and $(\lambda x . e') e$, which perform allocation, and case e of $\langle K_1 \rightarrow \lambda x . e_1; K_2 \rightarrow \lambda x . e_2 \rangle$, which evaluates e , are not normally expanded if e is non-trivial. However, if the lambda is known to be one-shot, then in these cases the whole expression must be one-shot, and even after eta-expansion e will be evaluated at most once. Thus in this case the above are translated (assuming e has arity 1) to, respectively, $\lambda y_1 . \lambda y_2 . \text{let } z = e \text{ in } (\lambda^1 x . e) y_1 y_2$ and $\lambda y_1 . (\lambda^1 x . e') e y_1$ and $\lambda y_1 . \lambda y_2 . \text{case } e \text{ of } \langle K_1 \rightarrow (\lambda^1 x . e_1) y_1 y_2; K_2 (\lambda^1 x . e_2) y_1 y_2 \rangle$. Subsequent β -reduction will bring all the lambdas in these expressions to the front, making their arity manifest; this makes for efficient STG code, and also enables further optimisations.

Floating. The floating-in transformation floats bindings in through one-shot lambdas, but not through ordinary lambdas (or type lambdas, for reasons of the interaction with floating out).

The floating-out transformation does not count one-shot lambdas or type lambdas as major levels for the purposes of selecting drop points for floating; instead bindings float right past them without noticing their presence.

Inlining. Variable occurrences under a one-shot lambda do not count as occurrences that are “inside a lambda and therefore dangerous to duplicate”. This causes both ordinary conditional inlining (*callSiteInline*) and unconditional inlining (*preInlineUnconditionally*) to inline more eagerly – in the latter case, a variable that occurs just once and not “inside a lambda” is *always* inlined, even if the occurrence is under a one-shot lambda.

However, our usage analysis is not the only source of one-shotness information. A number of other sources exist:

Manifest functions. The occurrence analyser (which informs the inliner) knows that in applications of manifest functions, such as $(\lambda x . \lambda y . e) a_1 a_2$, the lambdas are one-shot.

Join points. When the simplifier builds a case *join point*, it sets all the lambdas of its right-hand side to one-shot, since the join point will always be saturated, and invoked at most once.

The *build* hack. The occurrence analyser also knows about four very special functions, chiefly related to the *foldr/build* fusion machinery of Gill *et al.* [GLPJ93, Gil96]. It knows that the sole argument of *build* is fully applied, just once, to two arguments (and so both lambdas are one-shot), that the first argument of *augment* is ditto, that the first argument of *foldr* is always fully applied to two arguments (and so the argument’s second lambda is one-shot), and that the sole argument of *runSTRep* is applied just once (and so the argument’s lambda is one-shot).

The ST hack. A particularly gruesome hack is that any lambda whose binder is of type $\text{State} \# \sigma$ for some σ is treated as one-shot. (This type indicates the use

of *state threads* [LPJ95], which are used amongst other things to implement I/O.) This is very often true; such lambdas often arise from monadic code of the form

$$\begin{array}{lcl} a & \gg= & \lambda s . \\ e & \gg= & \lambda s' . \\ e' & \dots & \end{array}$$

which contains many one-shot lambdas.

The configuration option **-fusagesp-oneshotmode_n** controls the source of information for *isOneShotLambda*:

-fusagesp-oneshotmode0	always false
-fusagesp-oneshotmode1	uses the <i>build</i> and ST hacks only
-fusagesp-oneshotmode2	uses usage analysis result only
-fusagesp-oneshotmode3	true if the hack <i>or</i> the analysis say so

6.6.3 Configuration

To enable us to perform a range of measurements, and to tune the implementation without having to rebuild the compiler each time, a number of compiler flags have been introduced. We summarise them here.

-fusagesp	turns the analysis on and off.
-fusagespec	turns usage specialisation on and off (both generation of new specialisations and use of existing ones), as described in Section 6.3.3.
-fusagesp-mode_n	selects the generalisation mode used by the analysis, as described in Section 6.3.2. This allows the L_1 (monomorphic) analysis to be simulated, and permits investigating the tradeoffs involved in generalising bindings.
-fusage-heavy	turns on several extra passes of the usage analysis, as described in Section 6.3.1.
-fusagesp-dconmode_n -fusagesp-dconmaxcount_n	select the annotation scheme to use for data type declarations, as described in Section 6.4.11.
-fusagesp-oneshotmode_n	controls the function <i>isOneShotLambda</i> , as described in Section 6.6.2.

6.7 Measuring performance

We measured performance on the NoFib test suite, both by profiling the number of used-once thunks identified and by recording run times, allocations, and other standard metrics.

6.7.1 The NoFib test suite

NoFib⁶ is a suite of around seventy Haskell programs designed to be used for compiler benchmarking and testing. Most of them were written to solve real problems before being donated to the suite. The programs are mostly medium in size (between a few hundred and two thousand lines) and computationally intensive, and have been collected over a period of over ten years from a wide range of problem domains. The exception is a number of tiny traditional benchmarks such as the n -queens problem and Eratosthenes sieve, which are included in the “imaginary” category. Overall the suite exercises the compiler fairly thoroughly, and presents an analysis with a range of styles of code and levels of complexity, and is arguably a representative sample from the somewhat intangible category of all reasonable programs.

For the purposes of collecting a wide range of results, the NoFib suite is too large to use in its entirety. For most of our results, therefore, we use a subset of fifteen programs. These were chosen as follows. We collected ticky-ticky profiles (see Section 6.7.2 below) on a single run of each program in the NoFib suite, with the usage analysis enabled and the default choice of settings. Then we ruled out any programs that allocated less than 10 000 thunks during execution as being too trivial for consideration. After ranking the remaining 57 programs in order of effectiveness of the analysis, we selected every fifth, yielding a set of twelve. Finally, graphing effectiveness against opportunity (Figure 6.4) suggested that our sampling had missed some significant regions of the space, and so we chose three more by inspection of this graph. The chosen programs are described briefly below, with authors where known.

spectral/cryptarithm2. Solves the cryptarithm THIRTY + TWELVE + TWELVE + TWELVE + TWELVE + TWELVE = NINETY. Andy Gill.

spectral/fft2. Performs a 512-point FFT of a ramp waveform in two different ways, and then performs a Slow Cosine Transform of the same. Floating-point benchmark, Rex Page, Amoco Production Research.

imaginary/integrate. Extremely naïve numerical integration.

spectral/simple. Some kind of hydrodynamic simulation (weather?). Automatically translated to Haskell from Id, with hand annotations. Kattamuri Ekanadham and R. Paul. (additional selection)

spectral/multiplier. Simulation of a 16-to-32-bit binary multiplier circuit, run for 2000 clock cycles. John O’Donnell. (additional selection)

⁶NoFib [Par93] is available from <http://cvs.haskell.org/cgi-bin/cvsweb.cgi/fptools/nofib/> as part of the Glasgow fptools suite.

spectral/claify. Reduces $(a = a = a) = (a = a = a) = (a = a = a)$ to clausal form, seven times. Colin Runciman and David Wakeling.

real/cacheprof. Simulates execution and cache behaviour of Intel i386 processor. Julian Seward.

real/bspt. BSP tree geometric modeller and renderer.

real/lift. Lambda-lifting program. Performs lambda-lifting, constant folding, and full-laziness on a program with nine bindings. Simon L. Peyton Jones.

spectral/mandel. Renders the Mandelbrot set to a 150x150 pixel PPM file, with limit 75 iterations. (additional selection)

real/reptile. Interactive program for designing and printing potato prints. Text manipulation. Sandra Foubister and Colin Runciman.

real/gamteb. Monte-Carlo particle simulation. Patricia Fasel, Los Alamos National Laboratory.

spectral/puzzle. Brute-force solution to a puzzle (crossing the river?). Stephen Eldridge.

imaginary/queens. 10-queens.

spectral/boyer. Rewriting-based theorem prover. Bob Boyer.

We examine two of these programs, `queens` and `boyer`, in more detail in Section 6.9.

6.7.2 Ticky-ticky profiling

The primary goal of the analysis is to identify those thunks that are demanded at most once, so that they can be marked as single-entry rather than updatable. We modified the runtime system so that it would collect statistics allowing us to determine whether the analysis succeeded in this goal.

The Glasgow Haskell Compiler has a feature, *ticky-ticky profiling*, that is intended to collect statistics of this nature. When compiling with ticky-ticky profiling enabled, the runtime system maintains a large set of counters, which are incremented appropriately whenever the program does something: enters a thunk, pushes an update frame, squeezes an update frame, allocates memory for a 3-word constructor, performs a garbage collection, returns an unboxed 5-tuple, performs a fast entry to a function, etc..⁷

It was relatively straightforward to modify this system to collect the information we desired.

Detecting single-entry violation. On entering a single-entry thunk, we blackhole the thunk with a new kind of black hole, the `SE_BLACKHOLE`. If this black hole

⁷The name *ticky-ticky* alludes to the hypothetical sound of all the counters rapidly ticking over as the program runs, not unlike a knitting machine or one of the early relay computers.

is ever entered (*i.e.*, the thunk is entered more than once), the RTS aborts with the message “SE_BLACKHOLE at %p entered!”; for single-entry CAF thunks (not generated; see Section 6.6.1) an SE_CAF_BLACKHOLE is used instead. To ensure all such invalid enters are detected, eager blackholing is turned on.

Counting entries. Normally, when the RTS encounters an update frame on the stack (or the code generator knows the present result is to be used to update a thunk) it updates the relevant thunk with an indirection node pointing to the new value. Subsequent accesses to the thunk will follow the indirection to reach the desired value.

In order to count the number of thunks entered more than once, we alter this process slightly. When we update a thunk with an indirection, we use not a normal indirection but another kind of indirection, a *permanent indirection*.⁸ We then arrange that when a permanent indirection is entered, it mutates into a normal indirection node before performing the indirection. Thus the second demand on the thunk will go via a permanent indirection, while all subsequent demands will go via a normal indirection.

Obtaining accurate results. We disable a number of RTS optimisations that would otherwise invalidate our statistics: update-in-place, update squeezing, and indirection shorting (avoided by the use of permanent indirections). We also observe the number of selector thunks that are shorted, and verify it is not large.

Implementing and checking. Several new counters had to be added, and a number of cross-checks between them were used to ensure the validity of the results (they were certainly invalid at first!).

Given these modifications, it is possible from the counters to compute the numbers of single-entry and updatable thunks that are entered zero times, once, and more than once (the number of single-entry thunks entered more than once is always zero, since a second entry causes the program to abort). From these it is then possible to compute two interesting measures: the *opportunity*, the percentage of all thunks that are actually entered at most once, and the *effectiveness*, the percentage of all thunks entered at most once that are identified as single-entry by the analysis.

Of course, the latter statistic is only interesting if the analysis has not yielded its benefit by causing the simplifier to omit allocation of used-once thunks entirely, through the one-shot lambda mechanism. Comparisons show (Section 6.8.4) that in fact this mechanism sadly has little effect on the number of allocations made by the program. We therefore did not attempt to include this in the effectiveness score, although we do consider the two effects qualitatively in discussing the results.

⁸These are intended to assist the cost-centre profiling mechanism in its bookkeeping; this unrelated use of them means that ticky-ticky profiling and cost-centre profiling cannot now coexist. If desired, it would be fairly simple to create a new form of indirection and use it instead of permanent indirections.

6.7.3 Timings

The NoFib infrastructure, in conjunction with the compiler, shell, and runtime system, is capable of collecting module and executable binary size, compilation time, bytes allocated, bytes garbage-collected, garbage collection time, mutator time,⁹ and overall run time. We used this system to discover the bottom-line impact of the usage analysis in practice.

6.8 Results

We now measure the performance of the analysis, in terms both of its effectiveness at identifying used-once thunks and of its impact on run time and allocation. We compare its performance with that of our other analyses, and we investigate the effect of varying the analysis parameters. We also consider the costs of the analysis.

6.8.1 Effectiveness of the analysis

The results of the analysis are summarised in Figure 6.3. The programs in the NoFib suite run across the x -axis. For each program, the *opportunity*, *i.e.*, the percentage of all thunks that are in fact entered at most once, is marked with a triangle, and the *effectiveness*, *i.e.*, the percentage of these thunks that are marked single-entry by the analysis, is marked with a diamond. The effectiveness points are joined by a line, and the area under the line corresponds to the average effectiveness of the analysis on all programs in the suite. The programs have been ranked in order of decreasing effectiveness for clarity, and to allow percentiles to be obtained.¹⁰

The effectiveness of the analysis in terms of identifying used-once thunks is remarkably variable, but clearly significant for a significant proportion of programs tested. For half of the programs tested, the analysis identifies over 3% of the thunks entered at most once; in over a third of the programs over 10% are identified, and for around 15% of the programs over half are identified. Sadly, in more than a third of the programs less than 1% of the thunks entered at most once are identified, even though in all of these programs more than 45% of thunks are in fact single-entry.

It is striking that Figure 6.3 does not show a direct correlation between effectiveness and opportunity. To investigate this further, we plotted effectiveness against opportunity using the same data, in Figure 6.4. The x -axis (opportunity) starts at 40% because the minimum opportunity, for `gcd`, is 46.9%. The fifteen programs chosen for more detailed analysis are circled.¹¹

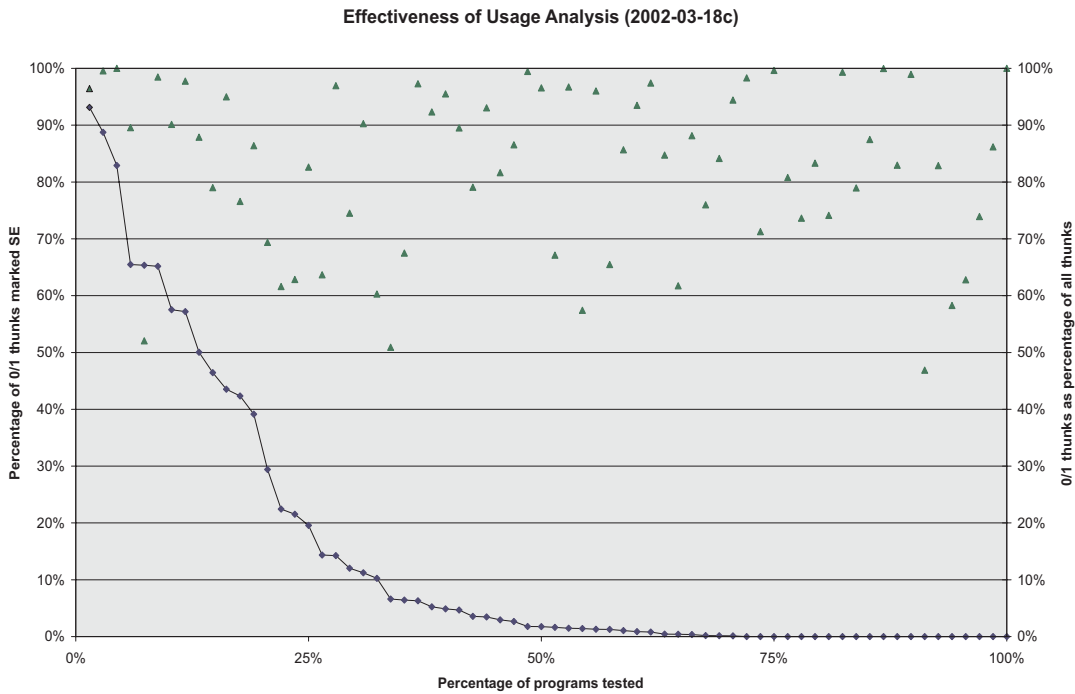
This graph shows that the analysis tends to do either well or poorly: most points lie in a cluster of points within 3% of the x -axis, but a number of points are much higher. The higher points appear to show a positive correlation between opportunity

⁹That is, time spent executing the user program, rather than initialising or garbage collecting.

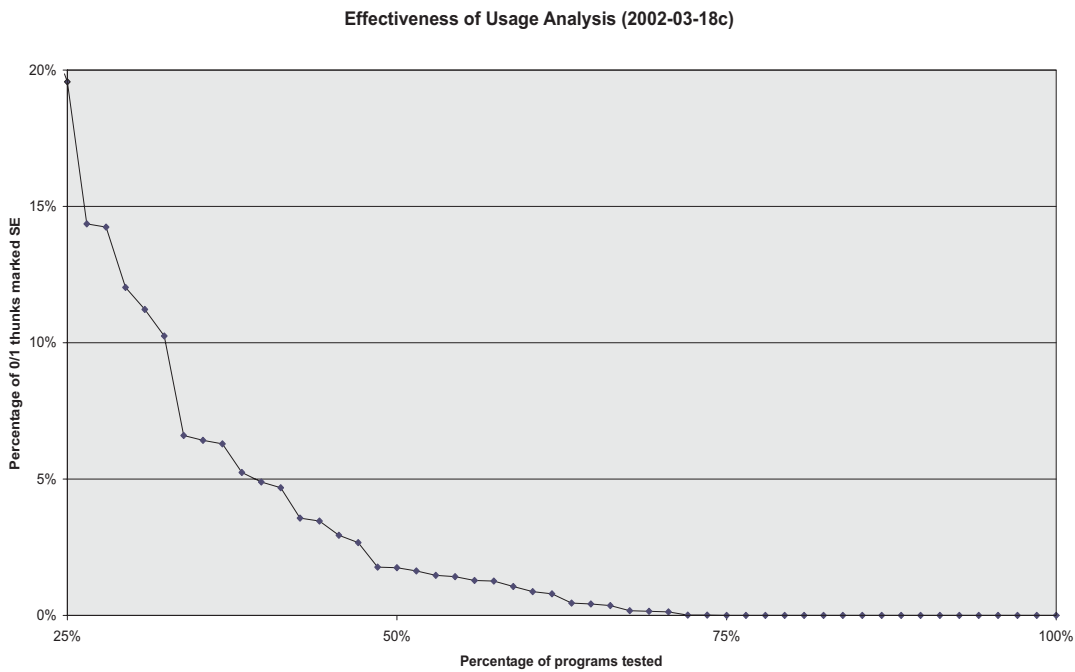
¹⁰Figure 6.3: results measured for all successfully-compiled programs in NoFib. Parameters: generalisation mode 1, usage specialisation on, annotation scheme 4, max 10, one-shot mode 3, normal (not heavy), for both libraries and programs.

¹¹Figure 6.4: parameters identical to Figure 6.3.

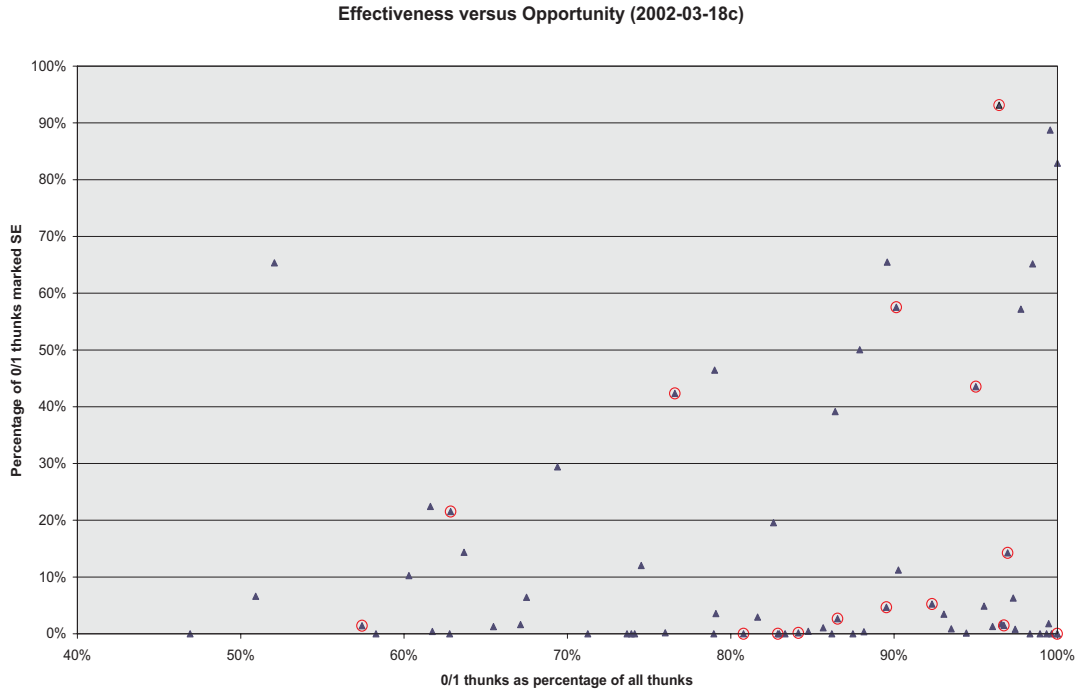
Figure 6.3 Effectiveness of usage analysis.



The bottom 75%, enlarged:



Line is percentage of thunks in fact entered at most once that were marked by the analysis. Triangles are the percentage of all thunks that were actually entered at most once.

Figure 6.4 Effectiveness versus opportunity.

Each marker shows the percentage of the thunks in fact used at most once in the program that were marked by the analysis, against the percentage of all thunks in fact used at most once. Circled markers correspond to the fifteen programs in the chosen subset.

and effectiveness: for programs with an opportunity of 50% the analysis detects no thunks, but as the opportunity rises the effectiveness rises more rapidly, to around 65% effectiveness at 100% opportunity.

There is one clear outlier, the program *cichelli*, for which the analysis detects 58% of the single-entry thunks, even though these are only 52% of the whole. This program invokes *map* many times in its inner loop, and the results are discovered by the inference to be used at most once. Thus the calls are specialised to *\$U1map*, and many updates are saved.

6.8.2 Comparing the analyses

In order to compare the effectiveness of the different analyses developed in this thesis, we ran the same test six times for different configurations of the analysis parameter settings:

All. The simple polymorphic usage analysis used above, with (►-DATA-ALL) used to annotate data types inside as well as on top (restricted to 10 usage arguments per data constructor, Section 6.4.11).

Full. The simple polymorphic usage analysis used above, with (►-DATA-FULL) used to annotate data types on top only.

Equal. The simple polymorphic usage analysis used above, with (\blacktriangleright -DATA-EQUAL) used to annotate data types with one usage annotation on all constructor arguments (as in [WPJ99], but with simple polymorphism).

Mono. The monomorphic analysis of Chapter 3, with (\blacktriangleright -DATA-EQUAL) (exactly the analysis of [WPJ99]).

None. No usage analysis at all.

Heavy. As for **All**, but with additional inference passes performed, to enhance the accuracy of usage information available to the simplifier.

The results are depicted in Figure 6.5.¹² Only the top 50% of programs are shown (ranked by effectiveness of **All**). **Heavy** and **None** are indicated by lines (the latter is everywhere zero, and so lies on the x -axis). The remaining four are indicated by shaded areas stacked in front of each other: **All** (dark grey) at the rear, then **Full** (mid grey), then **Equal** (light grey), and finally **Mono** (white) at the front. Triangular markers indicate the opportunity for each program (taken from the **All** run).

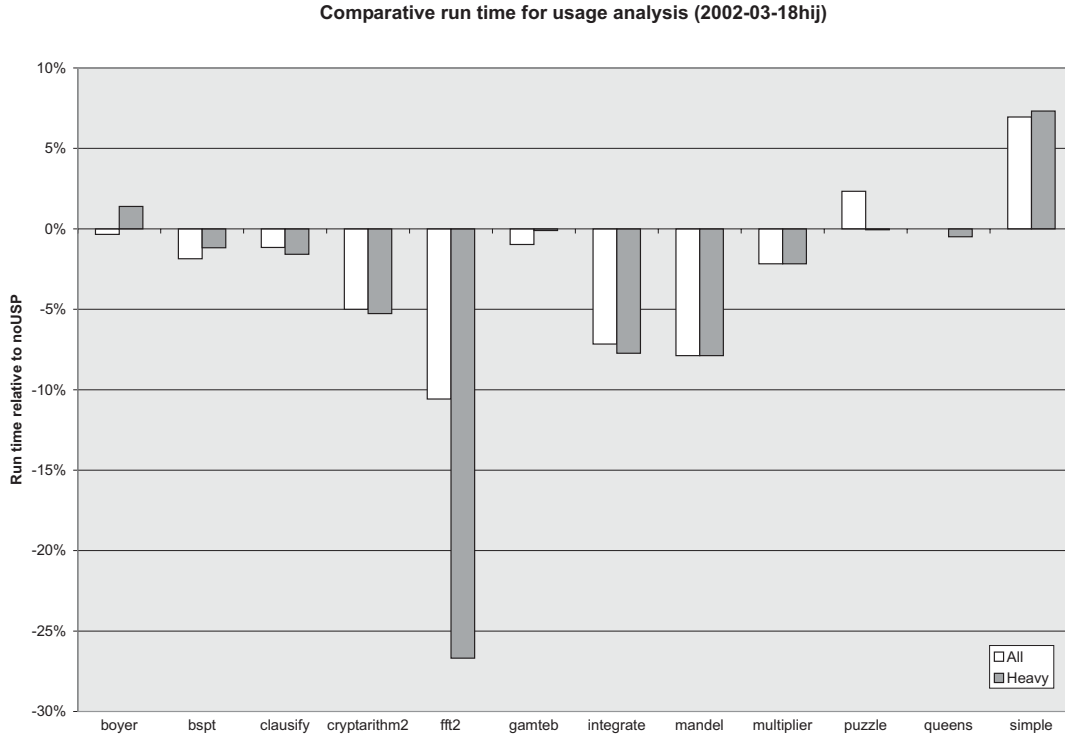
It is clear that the monomorphic analysis does very poorly, obtaining significant results for only a handful of programs. This is as we expected (Section 3.7.1). Even the polymorphic analysis does poorly if (\blacktriangleright -DATA-EQUAL) is used for data types, *i.e.*, if all constructor arguments are given the same usage. Finer treatment of data types is clearly required, and (\blacktriangleright -DATA-FULL) (one annotation per constructor argument) is sufficient to yield most of the benefit. For a few programs, however, this is not sufficient; annotating inside constructor arguments with (\blacktriangleright -DATA-ALL) (here restricted to a maximum of ten usage annotations per constructor argument) improves programs like `fft2`, `integrate`, and `constraints`. The common thread of these programs seems to be their use of dictionaries, which are represented by data types containing methods; a method selected from a dictionary will be assumed to use its arguments many times unless the annotation scheme descends inside its type. Finally, running the analysis multiple times during optimisation slightly reduces the measured effectiveness of the analysis; this is probably due to usage information being used aggressively by the simplifier to remove thunks entirely, and does not indicate inferiority of this analysis.

6.8.3 Run time and allocation

Does the analysis make programs run faster? Any answer must be only provisional at this stage, as much work remains to be done in exploiting the usage information appropriately (Section 6.10.3). But we are certainly omitting pushing and popping of update frames, and updating thunks, and the benefit of this should be significant even if (as seems possible) the abundance of usage information is fooling the simplifier into making bad choices.

The programs in NoFib all ran too fast to make realistic measurements of run times directly. To address this, we modified the programs to either to enlarge their

¹²Figure 6.5: parameters as stated, for both libraries and programs. One-shot mode 3, annotation scheme 2, or 4 max 0, or 4 max 10. Generalisation mode 1. Usage specialisation on.

Figure 6.6 Effect of analysis on run time.

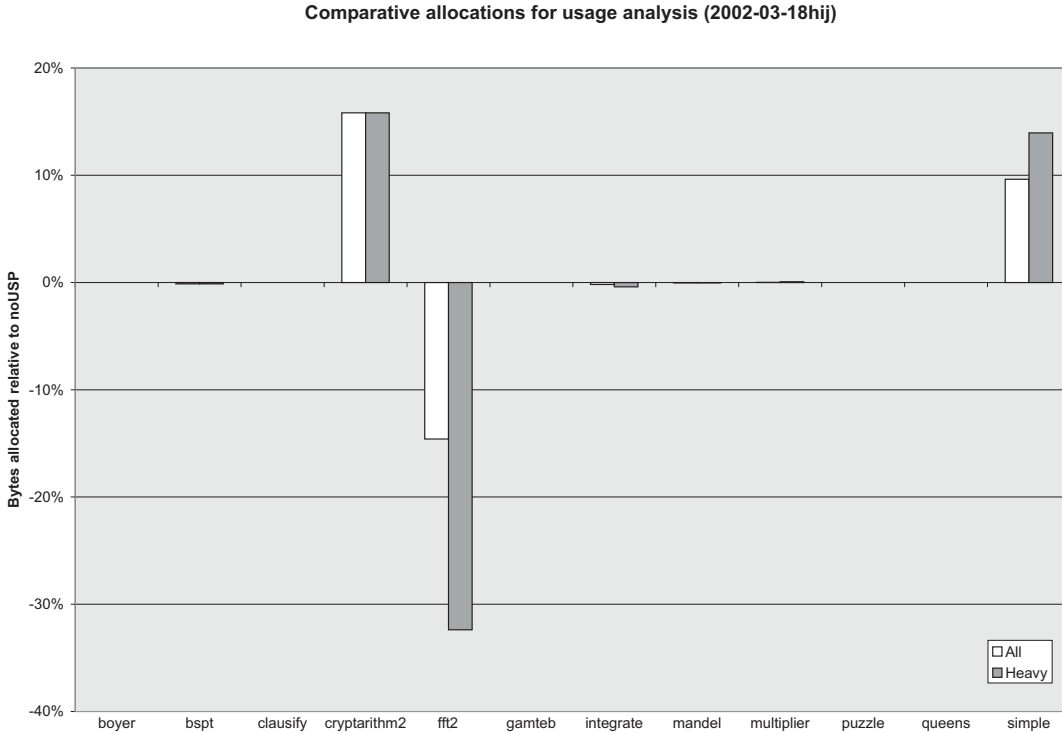
Each cluster corresponds to a program in the chosen subset. Bars represent the change in run time relative to a run without usage analysis, for usage analysis and heavy usage analysis.

problem size or to perform multiple repetitions of the problem (taking care to prevent sharing of solutions between repetitions!). We then performed run time measurements for the modified programs. For three programs (*cacheprof*, *reptile*, and *lift*) we were unable straightforwardly to increase the problem size, and so results from these program are omitted.

The run time results are shown in Figure 6.6 (and in Table 6.2). Times are plotted for **All** (the default setting) and **Heavy** (multiple inference passes) relative to **None**, the compiler without usage analysis turned on.¹³

For most programs, execution times improve, often by as much as 5% or more, although the average is around 2%. The programs that get faster are not always those with many updates being avoided (e.g., *mandel*), suggesting that other usage-enabled optimisations are an important component. For some programs execution times improve dramatically (*fft2*), but a few get worse. This is likely to be because the compiler is making poor inlining decisions; tradeoffs applicable when little usage information was available (such as “always inline through a used-once lambda”) may not be appropriate in the new setting. This appears to be the problem with *boyer* and *simple*. On the other hand, *puzzle* is merely affected by the inaccuracy of usage

¹³Figure 6.6: parameters apply to both libraries and programs. Other parameters: generalisation mode 1, usage specialisation on, annotation scheme 4, max 10, one-shot mode 3.

Figure 6.7 Allocations with and without the analysis.

Each cluster corresponds to a program in the chosen subset. Bars are the number of megabytes allocated during an extended execution of the program, for the conditions: without usage analysis, with usage analysis, with heavy usage analysis. No result was recorded for cacheprof.

information when only a single early inference pass is made; when multiple passes are made it improves.

Allocations are a frequent and relatively expensive operation during evaluation, and are also easily measured. Allocation is one statistic we might hope an accurate usage analysis to reduce, since one-shot lambdas are transparent to the floating transformations that attempt to minimise it. We measured allocations for the same extended runs, and show the results in Figure 6.7 (no measurement was collected for cacheprof in the **Heavy** case).¹⁴

Allocations seem not to be much affected by usage information. However, they are decreased in `fft2`, corresponding with its dramatic decrease in run time; but they are increased in `cryptarithm2` and `simple`. Since the former's run time improved slightly but a substantial number of updates were avoided, this suggests that extra allocations performed by misguided optimisation are masking run time improvements due to update avoidance here.

A selection of the effectiveness, run time, and allocation results for the chosen subset are also presented in Tables 6.1, 6.2, and 6.3 respectively (using programs

¹⁴Figure 6.7: parameters apply to both libraries and programs. Other parameters: generalisation mode 1, usage specialisation on, annotation scheme 4, max 10, one-shot mode 3.

Table 6.1 Opportunity and effectiveness of usage analysis (2002-03-18i).

With All usage analysis					
Program	NumThks	Num0/1	Opp	NumSE	Eff
spectral/boyer	138540109	114834097	(82.89%)	0	—
real/bspt	89006457	82063914	(92.20%)	4594989	(5.60%)
real/cacheprof	9188081	8226503	(89.53%)	385267	(4.68%)
spectral/clausify	92958053	90126008	(96.95%)	12831600	(14.24%)
spectral/cryptarithm2	92398443	89107603	(96.44%)	82983000	(93.13%)
spectral/fft2	72106558	70504863	(97.78%)	50397180	(71.48%)
real/gamteb	25270648	21381089	(84.61%)	952	(0.00%)
imaginary/integrate	33000051	31350037	(95.00%)	13650000	(43.54%)
real/lift	15627	13523	(86.54%)	361	(2.67%)
spectral/mandel	68665469	39396809	(57.37%)	493497	(1.25%)
spectral/multiplier	160187046	96737807	(60.39%)	21920860	(22.66%)
spectral/puzzle	88742980	71691273	(80.79%)	420	(0.00%)
imaginary/queens	9276092	9276075	(100.00%)	0	—
real/reptile	262944	254322	(96.72%)	3728	(1.47%)
spectral/simple	46398434	35407618	(76.31%)	15000598	(42.37%)

with extended run times). The opportunity and effectiveness percentages of Table 6.1 are stay remarkably stable between the **All** and **Heavy** analyses, and (for the opportunity) for no usage analysis as well. This is demonstrated in Tables 6.4 and 6.5.

Table 6.4, opportunity, does show some significant variations in the number of thunks allocated under different analyses, but it appears that “NumThks” and “Num0/1” vary together. This variation in allocation may be observed in graphical form in Figure 6.7, albeit in total bytes, rather than number of thunks. This matters because allocations include constructors as well as thunks, and because not all thunks are of the same size. However, the variation in number of thunks allocated with the three different analyses (not shown) shows very similar percentage variations in all cases to those of Table 6.3/Figure 6.7 except for `cryptarithm2`, where the number of thunks allocated for both **All** and **Heavy** is up 30.83%, against the number of bytes allocated which is up 15.82%.

The remaining tables are deferred to the end of the chapter: binary size (Table 6.6), mutator time, *i.e.*, time spent executing code rather than garbage collecting or in the operating system (Table 6.7), total garbage collected (Table 6.8), total size of modules in program (Table 6.9), total compilation time for all modules in program (Table 6.10), and opportunity and effectiveness for no analysis and the heavy usage analysis (Tables 6.11 and 6.12 respectively).

6.8.4 Examining the effect of one-shot lambda information

In order to better understand the consequences of usage-enabled optimisations (as opposed to simply update avoidance), we built a version of the libraries using only the *build* and ST hacks for one-shot lambda information. We then built the fifteen

Table 6.2 Run time (2002-03-18h,i,j).

Run time (sec)					
Program	None	All		Heavy	
spectral/boyer	11.49	11.45	(−0.35%)	11.65	(+1.39%)
real/bspt	10.22	10.03	(−1.86%)	10.1	(−1.17%)
real/cacheprof	1.43	1.44	(+0.70%)		—
spectral/clausify	16.41	16.22	(−1.16%)	16.15	(−1.58%)
spectral/cryptarithm2	11.01	10.46	(−5.00%)	10.43	(−5.27%)
spectral/fft2	16.45	14.71	(−10.58%)	12.06	(−26.69%)
real/gamteb	9.22	9.13	(−0.98%)	9.21	(−0.11%)
imaginary/integrate	6.98	6.48	(−7.16%)	6.44	(−7.74%)
real/lift	0	0	—	0	—
spectral/mandel	15.36	14.15	(−7.88%)	14.15	(−7.88%)
spectral/multiplier	15.18	14.85	(−2.17%)	14.85	(−2.17%)
spectral/puzzle	16.69	17.08	(+2.34%)	16.68	(−0.06%)
imaginary/queens	14.32	14.32	(+0.00%)	14.25	(−0.49%)
real/reptile	0.02	0.01	—	0.01	—
spectral/simple	13.8	14.76	(+6.96%)	14.81	(+7.32%)
<i>Geometric mean:</i>			(−2.19%)		(−4.07%)

Table 6.3 Bytes allocated (2002-03-18h,i,j).

Allocations (KB)					
Program	None	All		Heavy	
spectral/boyer	2658317	2658317	(+0.00%)	2658317	(+0.00%)
real/bspt	2079925	2077276	(−0.13%)	2077351	(−0.12%)
real/cacheprof	218012	216772	(−0.57%)		—
spectral/clausify	2505190	2505190	(+0.00%)	2505190	(+0.00%)
spectral/cryptarithm2	2540560	2942422	(+15.82%)	2942422	(+15.82%)
spectral/fft2	2229055	1903687	(−14.60%)	1507358	(−32.38%)
real/gamteb	1379477	1379489	(+0.00%)	1379490	(+0.00%)
imaginary/integrate	1309752	1307377	(−0.18%)	1304438	(−0.41%)
real/lift	390	389	(−0.26%)	390	(+0.00%)
spectral/mandel	2211441	2210654	(−0.04%)	2210654	(−0.04%)
spectral/multiplier	3599062	3599621	(+0.02%)	3601478	(+0.07%)
spectral/puzzle	2716233	2716233	(+0.00%)	2716233	(+0.00%)
imaginary/queens	1714215	1714215	(+0.00%)	1714215	(+0.00%)
real/reptile	7028	7027	(−0.01%)	7039	(+0.16%)
spectral/simple	1513394	1659026	(+9.62%)	1724393	(+13.94%)
<i>Geometric mean:</i>			(+0.46%)		(−0.83%)

Table 6.4 Stability of opportunity percentage over analyses.

Opportunity (thunks) Program	None		All		Heavy	
	NumThks	Num0/1	NumThks	Num0/1	NumThks	Num0/1
spectral/boyer	138540109	114834097	138540109	114834097	138540134	114834285
real/bspt	89006457	82059522	89006457	82063914	89009295	82056539
real/cacheprof	9139983	8178409	9188081	8226503	0	0
spectral/clausify	92958053	90126008	92958053	90126008	92958053	90126008
spectral/cryptarithm2	70625043	67334203	92398443	89107603	92398443	89107603
spectral/fft2	88838722	87241123	72106558	70504863	55345733	53727652
real/gamteb	25270212	21380554	25270648	21381089	25270616	21381040
imaginary/integrate	33150051	31500037	33000051	31350037	32850051	31200037
real/lift	15627	13523	15627	13523	15556	13452
spectral/mandel	68660344	39391684	68665469	39396809	68665469	39396809
spectral/multiplier	160137603	96688364	160187046	96737807	160267112	96897933
spectral/puzzle	88743820	71692113	88742980	71691273	88742620	71690913
imaginary/queens	9276092	9276075	9276092	9276075	9276092	9276075
real/reptile	262913	254291	262944	254322	263382	254684
spectral/simple	39832540	28847121	46398434	35407618	47769166	36780388
Arithmetic mean:		(85.93%)		(86.23%)		(85.99%)

Table 6.5 Stability of effectiveness percentage over usage analyses.

Effectiveness (thunks) Program	None			All			Heavy		
	Num0/1	NumSE	Eff	Num0/1	NumSE	Eff	Num0/1	NumSE	Eff
spectral/boyer	114834097	0 (0.00%)	114834097	114834097	0 (0.00%)	114834285	114834285	0 (0.00%)	0 (0.00%)
real/bspt	82059522	0 (0.00%)	82063914	82063914	4594989 (5.60%)	82056539	82056539	4595063 (5.60%)	4595063 (5.60%)
real/cacheprof	8178409	0 (0.00%)	8226503	8226503	385267 (4.68%)	0	0	0	—
spectral/clausify	90126008	0 (0.00%)	90126008	90126008	12831600 (14.24%)	90126008	90126008	12831600 (14.24%)	12831600 (14.24%)
spectral/cryptarithm2	67334203	0 (0.00%)	89107603	89107603	82983000 (93.13%)	89107603	89107603	82983000 (93.13%)	82983000 (93.13%)
spectral/fft2	87241123	0 (0.00%)	70504863	70504863	50397180 (71.48%)	53727652	53727652	33603580 (62.54%)	33603580 (62.54%)
real/gamteb	21380554	0 (0.00%)	21381089	21381089	952 (0.00%)	21381040	21381040	813 (0.00%)	813 (0.00%)
imaginary/integrate	31500037	0 (0.00%)	31350037	31350037	13650000 (43.54%)	31200037	31200037	13500000 (43.27%)	13500000 (43.27%)
real/lift	13523	0 (0.00%)	13523	13523	361 (2.67%)	13452	13452	291 (2.16%)	291 (2.16%)
spectral/mandel	39391684	0 (0.00%)	39396809	39396809	493497 (1.25%)	39396809	39396809	493510 (1.25%)	493510 (1.25%)
spectral/multiplier	96688364	0 (0.00%)	96737807	96737807	21920860 (22.66%)	96897933	96897933	21920860 (22.62%)	21920860 (22.62%)
spectral/puzzle	71692113	0 (0.00%)	71691273	71691273	420 (0.00%)	71690913	71690913	60 (0.00%)	60 (0.00%)
imaginary/queens	9276075	0 (0.00%)	9276075	9276075	0 (0.00%)	9276075	9276075	0 (0.00%)	0 (0.00%)
real/reptile	254291	0 (0.00%)	254322	254322	3728 (1.47%)	254684	254684	3898 (1.53%)	3898 (1.53%)
spectral/simple	28847121	0 (0.00%)	35407618	35407618	15000598 (42.37%)	36780388	36780388	15000599 (40.78%)	15000599 (40.78%)
Arithmetic mean:		(0.00%)			(20.21%)			(20.51%)	

chosen programs with all four possible settings: no one-shot lambda information (**None**), hack one-shot lambda information (**Hack**), analysis one-shot lambda information (**Anal**), and the logical OR of hack and analysis one-shot lambda information (**Hack+Anal**).

The consequences for the allocation performed by the program is graphed in Figure 6.8. Allocations were measured for the programs (without extending their run time) for the four settings, and also for the compiler without usage analysis at all. Values plotted are the change in allocations relative to this version. Points for the same program are joined by lines as indicated in the legend.¹⁵

Observe that with only the hack turned on, allocations are almost identical to those with the usage analysis turned off entirely; it is not clear why there is any difference at all, as one-shot lambda information should be the only way in which usage information affects the compiler. However, it is clear that the analysis has very little effect on allocations, either with the hack off or with it on, and certainly cannot be substituted for the hack. This is unfortunate. `fft2` displays an interesting interaction, where combining the two together yields significant benefit.

We expected the one-shot lambda information to have less direct effect on the effectiveness measure (Figure 6.9). Each cluster corresponds to a program, and contains bars for the four settings. Measurements were performed using **All**.¹⁶ Once again, this graph clearly shows that the hack has a far greater effect than the analysis on the transformation of the program.

6.8.5 Examining the effect of generalisation strategy

We also wanted to study the effects of altering how much generalisation was allowed, and whether specialisation was performed. The libraries were built with generalisation of exported binders only (as for **All**) and usage specialisation on, and then the chosen programs were measured for the four generalisation settings – no generalisation (as for **Mono**), exported binders only (as for **All**), top-level binders only, and all binders – and two specialisation settings – on in the left half, and off (both use of existing and generation of new) in the right.

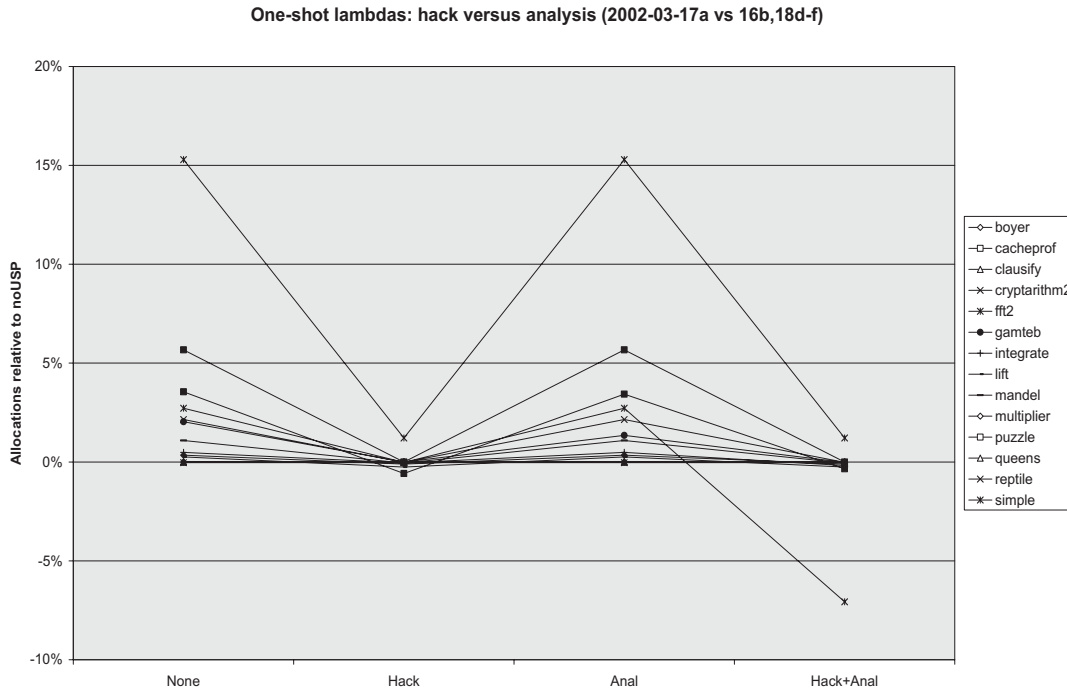
The consequences of this on effectiveness for the chosen programs is shown in Figure 6.10. Each line corresponds to a program in the chosen subset (note that results were not collected for `bspt` with usage specialisation on). An enlargement of the bottom 10% region is shown below the main graph.¹⁷

Although varying the generalisation parameters for the programs alone had little effect on most programs (library functions were already generalised, and so programs making heavy use of these still reaped the benefits without themselves being

¹⁵Figure 6.8: only the programs were compiled with varying settings; the libraries were compiled with the hack only. Other parameters: annotation scheme 4, max 10, normal (not heavy), no usage specialisation, for both libraries and programs.

¹⁶Figure 6.9: only the programs were compiled with varying settings; the libraries were compiled with the hack only. Other parameters: annotation scheme 4, max 10, normal (not heavy), no usage specialisation, for both libraries and programs.

¹⁷Figure 6.10: parameters: annotation scheme 4, max 10, one-shot mode 3, normal (not heavy), for both libraries and programs. Libraries compiled with generalisation mode 1, usage specialisation on.

Figure 6.8 Varying the analysis parameters: one-shot lambdas: allocations.

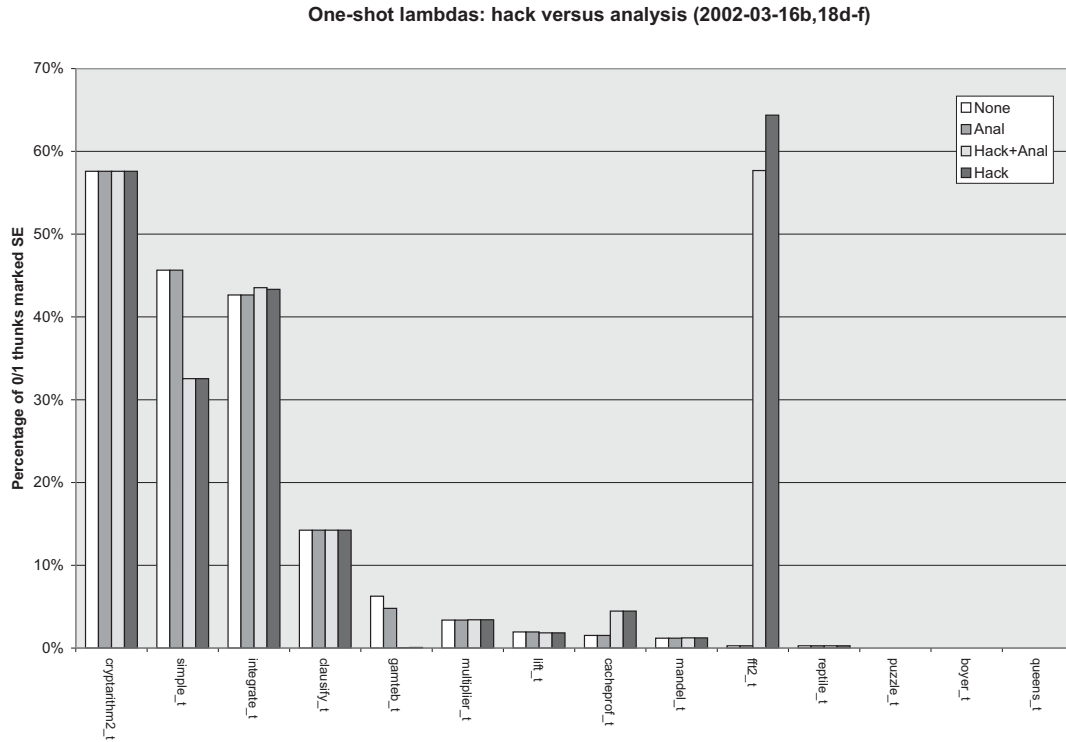
Each line corresponds to a program in the chosen subset. Points are number of bytes allocated by the program expressed as a percentage deviation from the no-usage-analysis case (with hack), against the conditions: no one-shot lambda detection, hacks (*build* and *ST*) only, usage analysis only, both hacks and analysis.

generalised), some trends can still be seen. Clearly, generalising more binders *reduces* the effectiveness of the analysis, just as predicted in Section 4.7.3, although generalising just exported binders is better than generalising none at all (see *bspt*). Turning usage specialisation off, as in the right half of the graph, also reduces the effectiveness, sometimes dramatically (see *multiplier*), because specialisation is a way of ameliorating the problems of generalisation (thunks in a specialised version can be annotated 1 rather than *u*). For *clausify* effectiveness is only reduced when non-exported toplevel binders are generalised; clearly this program has a toplevel binder that is generalisable, but in fact used at only a single, used-once, type.

Again, allocations are also of interest, and these graphs appear in Figure 6.11. Allocations are measured relative to the programs and libraries built with usage analysis switched off. The central region is enlarged below the main graph.¹⁸

Little effect of generalisation on allocations can be discerned, other than for *cryptarithm2* where clearly many specialised versions are being used of functions that are top-level but not exported.

¹⁸Figure 6.11: parameters: annotation scheme 4, max 10, one-shot mode 3, normal (not heavy), for both libraries and programs. Libraries compiled with generalisation mode 1, usage specialisation on.

Figure 6.9 Varying the analysis parameters: one-shot lambdas: effectiveness.

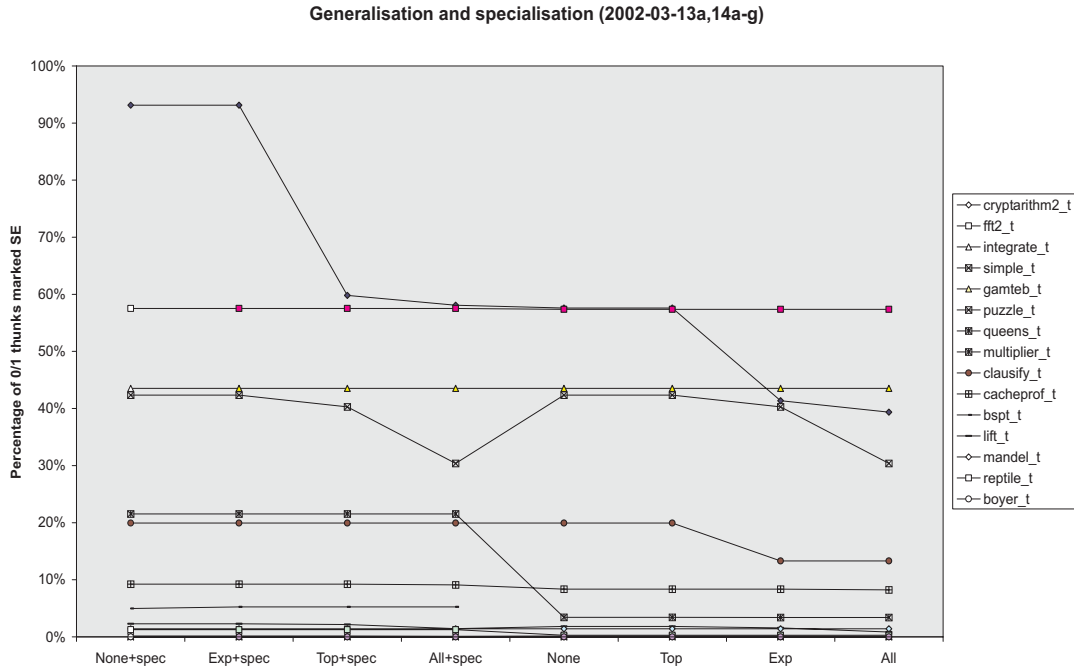
Each cluster corresponds to a program in the chosen subset. Bars are the effectiveness of the analysis, for the conditions: no one-shot lambda detection, hacks (build and ST) only, usage analysis only, both hacks and analysis.

6.8.6 Costs of analysis

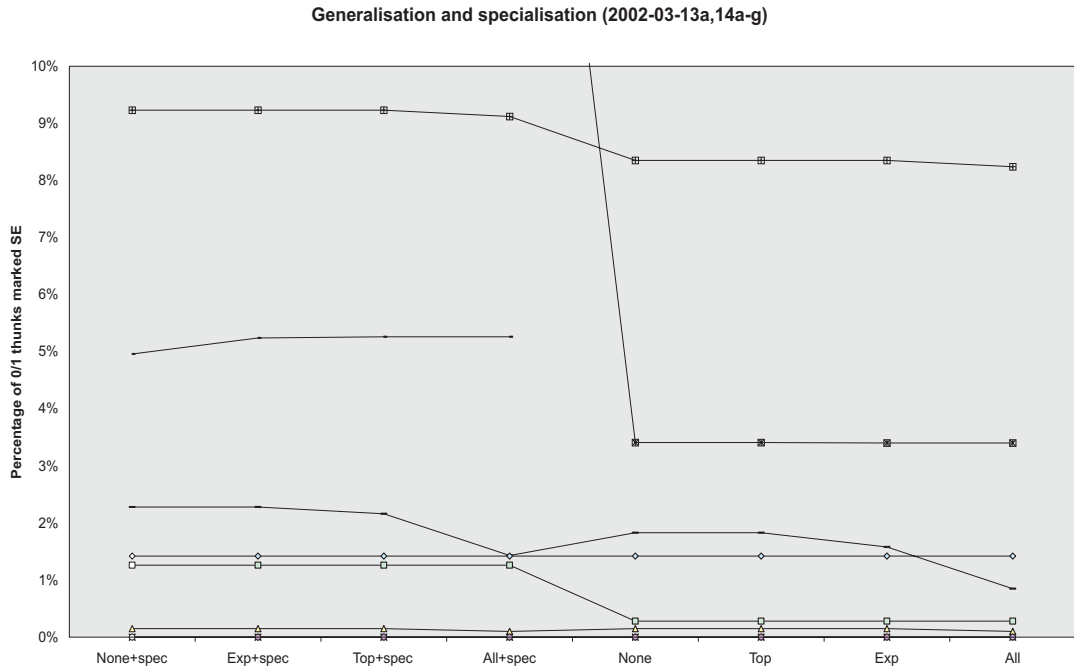
The usage analysis comes with a cost: compile times are increased, and specialisation and usage-enabled inlining means code sizes may increase also. In fact the code size increase is not significant with the present approach: on average the total size of object files for programs in our chosen set increased by 1.9%, rising to 4.6% with heavy usage analysis. Binaries (*i.e.*, including the linked libraries) increased by 0.4% and 0.8% respectively (see Tables 6.9 and 6.6).

Compile times, however, do increase significantly. The average total compile time for a program (possibly of multiple modules) in our chosen set increased by 70%, or 135% with heavy usage analysis. This is a significant cost, and is especially concerning because of its variation: *simple* took 271% longer to compile, while many programs took between just 20% and 30% longer (see Table 6.10). We attribute this cost to the inefficient implementation of the constraint solver in the present implementation (see Section 6.10.3), and believe it could be substantially improved without great difficulty.

Figure 6.10 Varying the analysis parameters:
generalisation and specialisation: effectiveness.

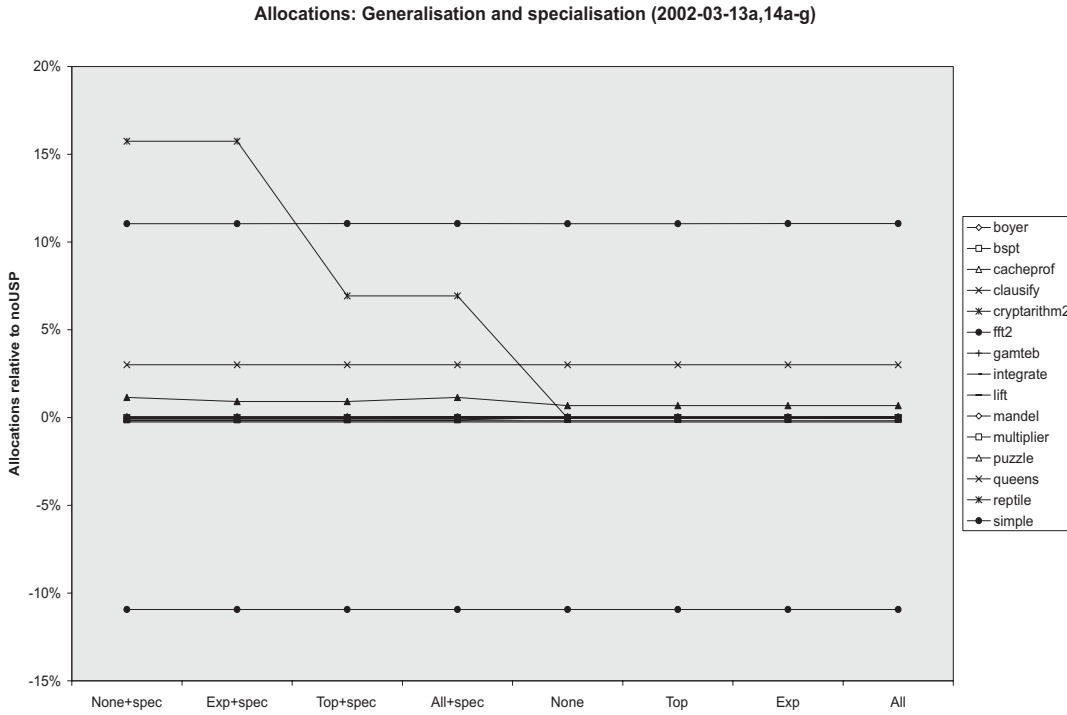


Enlarged view of 0%–10% effectiveness section:

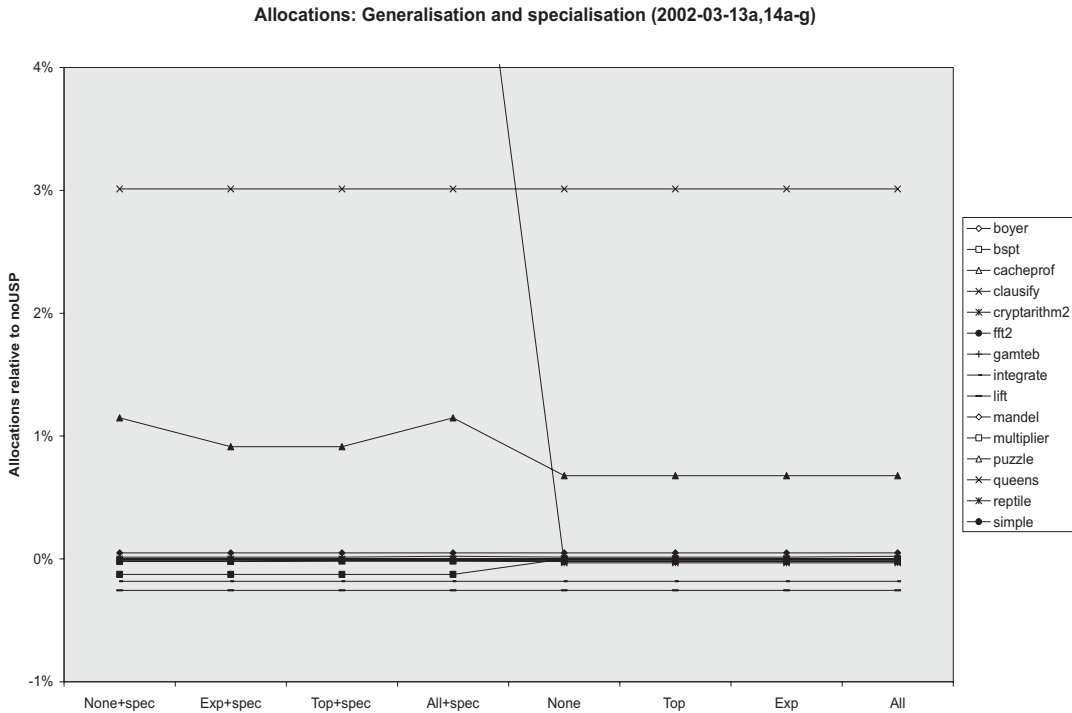


Each line corresponds to a program in the chosen subset. Points are the effectiveness of the analysis, against the conditions: generalisation mode 0 (no generalisation), 1 (exported binders only), 2 (all top-level binders), 3 (all binders), and usage specialisation (on or off). Order in legend corresponds to order in column “Exp+spec”. bspt failed to compile without *[sic]* usage specialisation.

Figure 6.11 Varying the analysis parameters:
generalisation and specialisation: allocations.



Enlarged view of -1% – $+4\%$ allocations section:



Each line corresponds to a program in the chosen subset. Points are the effectiveness of the analysis, against the conditions: generalisation mode 0 (no generalisation), 1 (exported binders only), 2 (all top-level binders), 3 (all binders), and usage specialisation (on or off). Order in legend corresponds to order in column “Exp+spec”. bspt failed to compile without [sic] usage specialisation.

Figure 6.12 Main loop of queens after optimisation.

```

oneToTen : List Int = enumFromToInt 1 10
one : Int = 1

go : List (List Int) → List (List Int)
= λqss0 : List (List Int) .
  case qss0 of
    Cons qs qss → let rest : List (List Int) = go qss
                    in
                    letrec go1 : List Int → List (List Int)
                        = λys0 : List Int .
                          case ys0 of
                            Cons y ys → case safe y one qs of
                                True → let qs1 : List Int
                                    = Cons y qs
                                    qss1 : List (List Int)
                                    = go1 ys
                                in
                                Cons qs1 qss1
                            False → go ys
                            Nil      → rest
                    in
                    go1 oneToTen
    Nil → Nil

```

6.9 Case studies

We now examine two programs for which the analysis was unable to discover a single used-at-most-once thunk. We find some clues toward designing a better analysis, and reasons why in some cases the analysis cannot be improved.

6.9.1 Queens: a case study

The code for the main loop of queens, after optimisation, is shown in Figure 6.12. The outer function, *go*, takes a list of board states and returns a list of board states with an additional queen added in the first file. The inner function, *go1*, is called for each of the input board states. It takes a list of queen positions to try in the new file, and computes a list of the legal board states containing the additional queen; it prepends this to the result of recursively calling *go* for the remaining input board states. The main loop contains two dynamically-allocated thunks, *rest* and *qss1* (the remaining bindings are to values).

The main function simply counts the number of board states; thus it uses each outermost cons-cell just once. One might therefore expect that the two thunks are entered at most once. Indeed, this is the case, as can be verified by manually marking these thunks single-entry, *but the usage analysis is unable to discover it*. Why is this?

The function *go1* is applied eleven times during a run, once for each node in the list of queens. Since this is more than once, any free variables of the function must be annotated ω , by rule (\blacktriangleright_2 -ABS). This rule is clearly justified in general, as we saw in Section 3.3.4. But *rest* is free in *go1*, yet is used just once. How?

The reason lies in the fact that *rest* occurs in only the case where *ys0* is Nil. Because *go1* is a simple primitive recursion over a list, and all lists have at most one Nil constructor, *rest* is in fact used just once. (The ω annotation of *qss1* is an indirect consequence of the annotation of *rest*.)

A way of recognising this case and dealing with it appropriately is not immediately obvious. Consider these two points. Firstly, generalising this to other data structures is not straightforward: a list has just one nil node, but a binary tree may have many leaves. Secondly, the primitive recursion is beneath a case and a constructor; in general, how can one recognise primitive recursion?

Certainly the code examined here is derived from an unfolding of *foldr*, and the occurrence of this function surely indicates the use of primitive recursion. However, the function no longer occurs at this point. We might consider preventing its unfolding (and thus preventing a large number of important optimisations, not least deforestation), or somehow leaving an indicator of its former presence (and ensuring the simplifier preserves the primitive-recursive semantics of the expression so indicated), but neither approach seems realistic.

6.9.2 Boyer: another case study

Another program in the NoFib test suite is *boyer*, a theorem prover written by Bob Boyer. It attempts to prove a theorem by repeatedly applying a set of rewrite rules. If the term can be unified with the left hand side of a rule, it is rewritten by applying the unifying substitution to the right hand side. If no rule matches, rewriting has completed and we may test if the result is the constant True or the constant False.

Of a total of 1 385 509 thunks allocated during the test run of *boyer*, 440 000 are never entered, 707 735 are entered exactly once, and only 237 152 are entered more than once. Despite this, our analysis marks *none* of them single-entry. We show below that in fact no analysis could conceivably mark more than 234 580 of these, leaving 913 155 thunks that must be marked updatable even though they may not in fact be used more than once. Given our belief that lazy languages are important and useful in practice, it should not really be such a surprise that some thunks are *necessarily* lazy.

Terms in *boyer* are represented by the following data structure.

```
data Term = Var Id | Fun Id (List Term) (List Lemma)
```

A Lemma is simply a pair of terms, representing a rewrite rule; the rewrite rules applying to each function are attached to occurrences of that function for convenience. Unification of a term with the left-hand side of a rule proceeds in the usual way. A variable in the rule matches anything in the term, and a function in the rule is matched by a function in the term only if the functions are the same and also all the arguments are unifiable.

A crucial point is that the analysis in fact annotates *bindings*, not the runtime entities we call thunks. Top-level bindings indeed correspond one-to-one with thunks at runtime, since they are allocated statically, but local bindings in fact allocate thunks *dynamically* whenever they are executed.

The opportunity we have measured so far is the proportion of *runtime* thunks that are entered at most once, but an inference may only annotate a binding • if *all* thunks allocated by it are entered at most once. There is no contention for top-level thunks, and indeed the analysis discovers rather many single-entry top-level thunks. But these are not very interesting, since each corresponds to at worst a single update, and there are never more than a hundred or so. For the much more significant local, dynamically-allocated thunks, however, the distinction is very important.

In the core code of *boyer* there are eight thunk bindings. Four of these contain intermediate results returned by the unifier when unifying two list of terms: for the head and for the tail of the list, they store the (success flag, unifying substitution) pair and the unifying substitution.

Considering the action of the unifier, we can see that these four thunks will very often be used just once or not at all. If the heads of the lists do not match, then we will enter the first thunk once to determine this, and the remaining three thunks not at all. If the heads match but the tails do not, then we will enter the first twice, the second possibly many times as the substitution is looked up repeatedly, but the third only once. It is only when the lists match completely that the first and third thunks are entered more than once. Since there are many rules but few matches, most instances of these thunks will be entered zero or once.

But these thunks must be marked with • or ! *statically*. Thus for, say, the first thunk considered above, any usage analysis must be able to assert that *all* instances of this thunk will be entered at most once during execution in order to annotate it single-entry. If even one instance is entered more than once, then a usage analysis *must* mark it updatable. Sadly, in cases like the present one, sometimes the instances are entered at most once, and other times they are entered many times. Without altering the code in some way to separate the allocations of the instances that will be entered at most once from those of instances that will be entered many times, *no analysis can possibly do better*.

Worse, not even altering the code can solve the problem in general. Here the usage of the thunks of interest depends on whether the input to the function matches the current rule or not. That is, the usage of the thunk depends upon its value! This certainly cannot be discovered in advance by any analysis. By manually altering the behaviour of the compiler for specific thunks, we were able to confirm that in the test run, all four of these thunks had in fact to be marked updatable.

Consequently, this example has shown that the counting of dynamic thunks in, e.g., Figure 6.4 can be misleading when considering the effectiveness of the analysis. Statistics on the number of code thunks that are always entered at most once should be collected, to allow the detection of such cases.

The remaining four thunks were also investigated. Two contain the computed boolean success flag returned by the unifier as part of the (success flag, unifying substitution) pair, one contains the argument list of a function after applying a substitution, and one contains the argument list of a function after applying a rewrite.

Of these, the last is actually used more than once during execution (as multiple rules are matched against the function), but the other three are invariably used at most once, despite the analysis being unable to discover this. Respectively, they are entered 9880, 217 460, and 7240 times, thus getting these annotations correct would save 17% of the updates performed during a run.

The first two thunks are each used in the same context by the unifier, as success flags. The flag is placed inside an intermediate-result thunk along with the substitution, and we have seen that this thunk may be used more than once. By the sharing condition of Section 5.3.2, this means that its contents are considered to be used more than once, and thus the success flag thunk is annotated `!`. The analysis is unable to realise that since the first access to the intermediate-result thunk is to obtain the success flag and the second is to obtain the substitution, there is no multiple use of the arguments. We address this issue speculatively in the appendix, Section C.4.4.

The remaining thunk is allocated to hold the result of recursively invoking the substitution function on the arguments of a `Fun`. It is marked `!` because the result of substitution is sometimes used twice: specifically, when substitution returns the input term unchanged, the term has been scrutinised once and is then returned to be used by the caller, and so must be annotated `!`. But the input term can only be returned unchanged when it is a `Var` (and not in scope of the substitution), and in this case there is no recursion and the thunk is not allocated! When the thunk is allocated, the input term is a `Fun`, and it is never returned unchanged to the caller; the scrutinisation is its only use. To correctly annotate this thunk, the analysis would have to take account of the fact that multiple use occurs only for `Vars`, and that this thunk occurs in a `Fun` branch. Presumably this would require usage types to include partial information on the possible values, and a mapping from these to the annotations to use in each case. Such a system would be complicated, and is beyond the scope of this thesis.

6.10 Conclusion

In this chapter we have demonstrated that the simple-polymorphic usage analysis we have designed can indeed be implemented in a production compiler for a full-scale language. After several false starts we have found a practical way of doing this, using usage projection types rather than embedding the analysis types directly into the type language of the compiler. We have demonstrated that the analysis has moderate benefit in practice. We also list a number of items of easy future work, areas in which the implementation could be improved.

6.10.1 The implementation

Throughout the progress of this thesis we have found implementation experience to be critical in guiding the development of the theory. Implementation has revealed the areas in which previous analyses are deficient; it has inspired possible solutions; it has identified errors in algorithms; it has forced the treatment of a full range of language constructs; and it has provided solid evidence of the success of the final

analysis.

A number of features of the compiler intermediate language and issues of implementation had not been addressed earlier in the design. Of necessity we addressed these in the implementation, and our solutions have been described. Some of these are of general import, such as issues relating to module structure, the precise treatment of data types, constructors, and case, and unboxed types and primitive operations, while some were more specific to the compiler we used, such as rules, existential constructors, and where usage projections were recorded.

Two extensions were required to obtain acceptable results. It is necessary to restrict generalisation to exported binders only, because over-generalisation causes too many thunks to be annotated *u*, requiring the flag *!*, rather than *1* (allowing *•*). And usage specialisation allows generalised functions such as *map* and *++* to yield good results when used in a *1*-annotated context. The details of the specialisation implementation in particular have been described.

6.10.2 Results obtained

In order to measure the performance of our analysis we used a substantial test suite, NoFib, containing around seventy moderately-sized Haskell programs. These provided a balanced and substantial body of code, most of it written to solve real problems, enabling a realistic assessment of the analysis. We measured both the success of our analysis at detecting used-once thunks (the immediate aim), and the impact on run time and allocations (an indirect but desirable consequence). We also examined the cost of the analysis, in terms of code size and compilation time.

The Glasgow Haskell Compiler was a hard target: it aggressively performs a wide range of transformations, in particular strictness analysis, unboxing, and inlining, which compete with usage analysis in reducing the costs of lazy evaluation. Nonetheless, we obtained significant benefit from the usage analysis despite this competition.¹⁹

The results suggest that the analysis is moderately effective, but surprisingly variable. For a few programs it does extremely well, identifying over half of all single-entry thunks in 15% of programs. But for many programs it does extremely poorly, with a median of just 3% of all single-entry thunks identified. This wide variability has not been successfully explained; it does not seem to be for lack of opportunity, since in all but one program tested more than half of all thunks were entered at most once. Some possible explanations are suggested by the case studies of *queens* and *boyer*, programs on which the analysis does poorly. In particular, a special property of primitive recursion on lists is not exploited by our (general) analysis, and zero-usage is not tracked. However, the study of *boyer* indicated that some used-once thunks are *in principle* unidentifiable by any analysis; these are inherently lazy, with the number of uses being determined at runtime by the data they hold. This suggests that more accurate measurements of the opportunity might be performed that exclude such thunks.

¹⁹Furthermore, the programs in the NoFib test suite are also a hard target: the compiler has been repeatedly tuned to perform well on these programs.

Measurements of run time show improvements of between two and four percent average, with over 26% recorded for `fft2`. Allocations are less affected, and sometimes worsen, although we conjecture that this is due to weaknesses in the way the simplifier makes use of the additional usage information it is provided with by the analysis.

Direct measurements of the effect of usage information on the simplifier showed that in most cases it had little impact, although perhaps using different metrics (such as run time) and more accurate usage information (the heavy analysis rather than the standard one) might give better results. The strategy of restricting generalisation to exported binders only (and certainly not allowing it for local bindings) is very important; this is evident but not especially clear from the measurements, which investigated varying this parameter for the compiled program only while leaving it constant for the standard libraries. Specialisation, similarly, is important for some programs, and occasionally yields a huge win (as with `cichelli`).

Varying the implementation parameters to simulate the several analyses of this thesis reassuringly confirms our earlier results, that the monomorphic analysis is quite inadequate, and that finer treatment of data structures is required than given by (`►-DATA-EQUAL`) of [WPJ99].

6.10.3 Easy future work

There are a number of improvements and additions to be made to the implementation that would be relatively straightforward, but that we simply did not have time to implement. For the benefit of future workers, we list them here.

Improving the constraint solver. It was originally my intention to use the almost-linear-time constraint solver described in Section 3.5.4. At the time of initially coding the prototype, however, no Haskell implementation of union-find was readily to hand, and the implementation of such a stateful algorithm did not appeal; instead we omitted the optimisation for equality, treating it as two inequality constraints, and used an $O(\log n)$ tree-based finite map implementation instead of $O(1)$ array lookup. These two factors mean that the constraint solver is much slower than it needs to be; optimising this would lead to very significant improvements in compilation time.

Accurately measuring the static opportunity. In Section 6.9.2 we discovered that the opportunity measurement can be misleading. The proportion of (static) thunks *in the program* for which *all instances* are entered at most once should be measured, rather than the proportion of (dynamic) thunk instances, so as not to penalise the analysis for programs which involve *necessary laziness*.

Specialisation generation. The naïve generation of specialisations (always a single specialisation, with all arguments 1) should be replaced by a better algorithm.

Update-frame check elision. Gustavsson [Gus98] is able to avoid certain update marker checks by means of an interval analysis (Section 2.6); we believe it should be possible for us to avoid some update marker checks by examining

the usage annotations on values: a one-shot lambda, for example, will always find its argument on the stack, never an update frame.

Improving the simplifier. We have not modified the simplifier at all; we have merely informed it of more one-shot lambdas. Now that more usage information is available, we should reconsider the heuristics used by the simplifier to decide, e.g., when to performing floating transformations. Choices made when few lambdas were one-shot may no longer be valid when many are. Additionally, there are other optimisations, such as inlining of used-once thunks, that are usage-specific and should be added (the simplifier does perform some inlining here, but directed by a syntactic analysis combined with one-shot lambda information, rather than by the inferred usage information).

Deferring the final pass. The final pass of the inference should be run after conversion to A-normal form, at the point where the choice of *actually*-exported identifiers has been made. At present we pessimise only *user*-exported identifiers, which is over-optimistic in the presence of cross-module inlining and subsequent optimisation. This is the cause of *bspt* and *cacheprof* sometimes failing to compile during tests.

Unnecessary constructor arguments. Section 6.4.10 describes a false assumption currently made about the variance of constructor arguments, which leads to unnecessarily-pessimistic types for some functions. Variance of constructor usage arguments should be taken into account by this optimisation.

We expect that a number of these modifications will yield significant improvements in the performance of the analysis.

Figure 6.13 Core terms.

```

data Expr b      -- "b" for the type of binders,
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b [Alt b]      -- Binder gets bound to value of scrutinee
                                   -- DEFAULT case must be *first*,
                                   -- if it occurs at all
  | Note   Note (Expr b)
  | Type   Type                    -- This should only show up at the top
                                   -- level of an Arg

type Arg b = Expr b                -- Can be a Type

type Alt b = (AltCon, [b], Expr b) -- (DEFAULT, [], rhs) is the
                                   -- default alternative

data AltCon = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT
  deriving (Eq, Ord)

data Bind b = NonRec b (Expr b)
  | Rec [(b, (Expr b))]

data Note
  = SCC CostCentre

  | Coerce
    Type      -- The to-type: type of whole coerce expression
    Type      -- The from-type: type of enclosed expression

  | InlineCall      -- Instructs simplifier to inline
                    -- the enclosed call

  | InlineMe        -- Instructs simplifier to treat the enclosed expr
                    -- as very small, and inline it at its call sites

```

And added for the usage analysis:

```

  | TopUsage      -- Records topmost usage annotation of this sub-
    Usage         -- expr for communicating to the A-nf converter

```

(from GHC version 5.03.20020220)

Figure 6.14 Core types.

```

type SuperKind = Type
type Kind      = Type

data Type
  = TyVarTy TyVar

  | AppTy
    Type      -- Function is *not* a TyConApp
    Type

  | TyConApp      -- Application of a TyCon
    TyCon        -- *Invariant* saturated applications of FunTyCon
                  -- and synonyms have their own constructors,
                  -- below.
    [Type]       -- Might not be saturated.

  | FunTy        -- Special case of TyConApp:
    Type        -- TyConApp FunTyCon [t1,t2]
    Type

  | ForAllTy      -- A polymorphic type
    TyVar
    Type

  | SourceTy      -- A high level source type
    SourceType   -- ...can be expanded to a representation type...

  | NoteTy        -- A type with a note attached
    TyNote
    Type         -- The expanded version

data TyNote
  = FTVNote TyVarSet -- The free type variables of the noted expr

  | SynNote Type      -- Used for type synonyms
                      -- The Type is always a TyConApp, and
                      -- is the un-expanded form.
                      -- The type to which the note is attached
                      -- is the expanded form.

data SourceType
  = ClassP Class [Type] -- Class predicate
  | IParam (IPName Name) Type -- Implicit parameter
  | NType TyCon [Type] -- A *saturated*, *non-recursive*
                      -- newtype application
                      -- [See notes at top about newtypes]

```

(from GHC version 5.03.20020220)

Table 6.6 Binary size (2002-03-18h,i,j).

Binary size (KB)					
Program	None	All		Heavy	
spectral/boyer	243	243	(+0.00%)	251	(+3.29%)
real/bspt	399	400	(+0.25%)	401	(+0.50%)
real/cacheprof	558	559	(+0.18%)		—
spectral/clausify	227	227	(+0.00%)	227	(+0.00%)
spectral/cryptarithm2	242	245	(+1.24%)	245	(+1.24%)
spectral/fft2	358	359	(+0.28%)	360	(+0.56%)
real/gamteb	428	430	(+0.47%)	430	(+0.47%)
imaginary/integrate	327	329	(+0.61%)	329	(+0.61%)
real/lift	276	278	(+0.72%)	278	(+0.72%)
spectral/mandel	396	398	(+0.51%)	399	(+0.76%)
spectral/multiplier	232	233	(+0.43%)	233	(+0.43%)
spectral/puzzle	237	236	(−0.42%)	236	(−0.42%)
imaginary/queens	210	210	(+0.00%)	210	(+0.00%)
real/reptile	425	426	(+0.24%)	430	(+1.18%)
spectral/simple	515	521	(+1.17%)	526	(+2.14%)
<i>Geometric mean:</i>			(+0.38%)		(+0.82%)

Table 6.7 Mutator time (2002-03-18h,i,j).

Mutator time (sec)					
Program	None	All		Heavy	
spectral/boyer	11.09	11.09	(+0.00%)	11.31	(+1.98%)
real/bspt	9.16	8.96	(−2.18%)	9	(−1.75%)
real/cacheprof	1.11	1.13	(+1.80%)		—
spectral/clausify	16.34	16.12	(−1.35%)	16.09	(−1.53%)
spectral/cryptarithm2	10.86	10.38	(−4.42%)	10.35	(−4.70%)
spectral/fft2	14.44	12.69	(−12.12%)	10.02	(−30.61%)
real/gamteb	7.27	7.22	(−0.69%)	7.2	(−0.96%)
imaginary/integrate	4.74	4.44	(−6.33%)	4.55	(−4.01%)
real/lift	0	0	—	0	—
spectral/mandel	15.3	14.08	(−7.97%)	14.06	(−8.10%)
spectral/multiplier	14.92	14.65	(−1.81%)	14.68	(−1.61%)
spectral/puzzle	15.87	16.34	(+2.96%)	15.88	(+0.06%)
imaginary/queens	14.28	14.29	(+0.07%)	14.17	(−0.77%)
real/reptile	0.02	0	—	0.01	—
spectral/simple	10.52	10.96	(+4.18%)	10.82	(+2.85%)
<i>Geometric mean:</i>			(−2.24%)		(−4.54%)

Table 6.8 Garbage collected (2002-03-18h,i,j).

Garbage collected (KB)					
Program	None	All		Heavy	
spectral/boyer	15533	15533	(+0.00%)	15905	(+2.39%)
real/bspt	30589	30929	(+1.11%)	30639	(+0.16%)
real/cacheprof	18289	17346	(−5.16%)		—
spectral/clausify	1744	1742	(−0.11%)	1742	(−0.11%)
spectral/cryptarithm2	1610	1270	(−21.12%)	1270	(−21.12%)
spectral/fft2	97871	96941	(−0.95%)	96411	(−1.49%)
real/gamteb	106632	106552	(−0.08%)	106553	(−0.07%)
imaginary/integrate	61058	54789	(−10.27%)	54848	(−10.17%)
real/lift	5	5	(+0.00%)	5	(+0.00%)
spectral/mandel	1093	1110	(+1.56%)	1110	(+1.56%)
spectral/multiplier	11602	11674	(+0.62%)	11923	(+2.77%)
spectral/puzzle	35878	35878	(+0.00%)	35878	(+0.00%)
imaginary/queens	97	97	(+0.00%)	97	(+0.00%)
real/reptile	24	24	(+0.00%)	24	(+0.00%)
spectral/simple	158766	166165	(+4.66%)	172795	(+8.84%)
<i>Geometric mean:</i>			(−2.19%)		(−1.48%)

Table 6.9 Total size of modules (2002-03-18h,i,j).

Total size of modules (KB)					
Program	None	All		Heavy	
spectral/boyer	34	34	(+0.00%)	41	(+20.59%)
real/bspt	148	150	(+1.35%)	150	(+1.35%)
real/cacheprof	255	256	(+0.39%)		—
spectral/clausify	14	14	(+0.00%)	15	(+7.14%)
spectral/cryptarithm2	18	20	(+11.11%)	20	(+11.11%)
spectral/fft2	19	19	(+0.00%)	19	(+0.00%)
real/gamteb	49	50	(+2.04%)	51	(+4.08%)
imaginary/integrate	6	6	(+0.00%)	6	(+0.00%)
real/lift	56	58	(+3.57%)	58	(+3.57%)
spectral/mandel	12	13	(+8.33%)	13	(+8.33%)
spectral/multiplier	19	19	(+0.00%)	19	(+0.00%)
spectral/puzzle	23	23	(+0.00%)	23	(+0.00%)
imaginary/queens	1	1	(+0.00%)	1	(+0.00%)
real/reptile	140	140	(+0.00%)	145	(+3.57%)
spectral/simple	144	148	(+2.78%)	153	(+6.25%)
<i>Geometric mean:</i>			(+1.92%)		(+4.57%)

Table 6.10 Total compilation time (2002-03-18h,i,j).

Program compile time (KB)					
Program	None	All		Heavy	
spectral/boyer	3.9	9.4	(+141.03%)	19.5	(+400.00%)
real/bspt	30.48	68.01	(+123.13%)	127.06	(+316.86%)
real/cacheprof	33.18	88.25	(+165.97%)		—
spectral/clausify	2.7	3.43	(+27.04%)	4.39	(+62.59%)
spectral/cryptarithm2	4.12	5.8	(+40.78%)	7.94	(+92.72%)
spectral/fft2	5.46	7.82	(+43.22%)	9.03	(+65.38%)
real/gamteb	17.04	25.39	(+49.00%)	27.8	(+63.15%)
imaginary/integrate	1.61	2.13	(+32.30%)	2.43	(+50.93%)
real/lift	10.52	29.47	(+180.13%)	53.05	(+404.28%)
spectral/mandel	4.05	5.58	(+37.78%)	6.55	(+61.73%)
spectral/multiplier	2.98	3.64	(+22.15%)	5.03	(+68.79%)
spectral/puzzle	3.83	4.99	(+30.29%)	6.77	(+76.76%)
imaginary/queens	1.07	1.31	(+22.43%)	1.46	(+36.45%)
real/reptile	24.66	34.58	(+40.23%)	48.59	(+97.04%)
spectral/simple	18.05	66.96	(+270.97%)	160.39	(+788.59%)
<i>Geometric mean:</i>			(+69.85%)		(+134.78%)

Table 6.11 Opportunity and effectiveness without usage analysis (2002-03-18h).

Without usage analysis					
Program	NumThks	Num0/1	Opp	NumSE	Eff
spectral/boyer	138540109	114834097	(82.89%)	0	—
real/bspt	89006457	82059522	(92.20%)	0	—
real/cacheprof	9139983	8178409	(89.48%)	0	—
spectral/clausify	92958053	90126008	(96.95%)	0	—
spectral/cryptarithm2	70625043	67334203	(95.34%)	0	—
spectral/fft2	88838722	87241123	(98.20%)	0	—
real/gamteb	25270212	21380554	(84.61%)	0	—
imaginary/integrate	33150051	31500037	(95.02%)	0	—
real/lift	15627	13523	(86.54%)	0	—
spectral/mandel	68660344	39391684	(57.37%)	0	—
spectral/multiplier	160137603	96688364	(60.38%)	0	—
spectral/puzzle	88743820	71692113	(80.79%)	0	—
imaginary/queens	9276092	9276075	(100.00%)	0	—
real/reptile	262913	254291	(96.72%)	0	—
spectral/simple	39832540	28847121	(72.42%)	0	—

Table 6.12 Opportunity and effectiveness of heavy usage analysis (2002-03-18j).

With heavy usage analysis					
Program	NumThks	Num0/1	Opp	NumSE	Eff
spectral/boyer	138540134	114834285	(82.89%)	0	—
real/bspt	89009295	82056539	(92.19%)	4595063	(5.60%)
real/cacheprof	0	0	—	0	—
spectral/clausify	92958053	90126008	(96.95%)	12831600	(14.24%)
spectral/cryptarithm2	92398443	89107603	(96.44%)	82983000	(93.13%)
spectral/fft2	55345733	53727652	(97.08%)	33603580	(62.54%)
real/gamteb	25270616	21381040	(84.61%)	813	(0.00%)
imaginary/integrate	32850051	31200037	(94.98%)	13500000	(43.27%)
real/lift	15556	13452	(86.47%)	291	(2.16%)
spectral/mandel	68665469	39396809	(57.37%)	493510	(1.25%)
spectral/multiplier	160267112	96897933	(60.46%)	21920860	(22.62%)
spectral/puzzle	88742620	71690913	(80.79%)	60	(0.00%)
imaginary/queens	9276092	9276075	(100.00%)	0	—
real/reptile	263382	254684	(96.70%)	3898	(1.53%)
spectral/simple	47769166	36780388	(77.00%)	15000599	(40.78%)

Chapter 7.



Conclusion and Future Work

In the preceding chapters we have motivated and developed our simple polymorphic usage analysis, measured its effectiveness, and considered one possible future extension. We now summarise the argument of the thesis, make some observations on the development of the analysis and implementation, enumerate several future research directions, and conclude.

7.1 The problem

Implementations of lazy functional languages must spend much time allocating, evaluating, and updating thunks, the delayed and shared computations that give these languages their name. These activities are expensive, both in time and in heap allocation. Therefore, an analysis that identifies where such activities are unnecessary would enable lazy functional programs to run faster and use less heap space. This is the essential observation with which we began our research. Specifically, if one knows that a thunk will never be used more than once, it need not be updated once evaluated, and in certain circumstances the allocation can be avoided entirely by simply inlining the computation where it is used. Our measurements have confirmed that there is significant potential for such an analysis.

7.2 The solution

In this thesis, we have designed a sound, practical, type-based usage analysis that achieves acceptable results on a range of programs. The analysis handles all the features of real functional languages, including type polymorphism and general algebraic data types. We have designed a novel form of polymorphism, *simple polymorphism*, and sought to understand its power. To justify the analysis we have written an obviously-correct operational semantics and proven soundness with respect to it. Finally, we have implemented the analysis in a production compiler, and made comprehensive measurements of its performance in this environment, and we have examined cases where it performs poorly and explained the reasons why. One interesting direction has been explored speculatively.

The analysis was developed incrementally. First, subsumption was added to an existing monomorphic analysis in order to address the problem of poisoning. Implementation results suggested that polymorphism was required, but we believed the obvious constrained polymorphism to be too expensive. Therefore, secondly, we developed simple polymorphism, adding it to the well-typing rules and designing a sound inference algorithm for it. Thirdly, we added in two key features of real functional languages, type polymorphism and general algebraic data types. Type polymorphism was relatively easily added, but its incorporation into the semantics and proofs underlying our analysis involved some subtlety. Algebraic data types opened up a wide design space, which we were able to codify and explore by means of annotation schemes. These abstracted the selection of a point in this design space from the workings of the well-typing rules, inference (and implementation) and soundness proofs.

Simple polymorphism is a new point on the power/complexity curve for type systems, used in this thesis for usage analysis but certainly applicable more widely. We have given (trivial) type rules and (non-trivial) inference rules for simple polymorphism, including a novel *approximating* closure algorithm used when generalising a term. Practical experience shows that it works fairly well. Theoretically, we have shown soundness of the well-typing rules and of the inference, and proven a complexity result. We have given some intuition for the behaviour of the algorithm and its choice of generalised type, and suggested directions for understanding it more formally both directly and as a restriction of constrained polymorphism, but we have not given simple polymorphism a truly formal foundation.

The operational semantics encodes our notion of usage. It is presented in such a way as to clearly distinguish the *outputs* of a usage analysis (update flags) from the *intermediate values* used in computation (usage annotations): other presentations, such as [TWM95a], have confused the two, giving for example an operational semantics that depends on types despite intending a type-erasure implementation. For the purposes of proof, we also give a version of the semantics which *preserves types* while retaining exactly the behaviour of a type-erasure semantics; this is significant because a naïve typed semantics would lose sharing in the presence of polymorphism.

Implementing the analysis has been central to our entire programme. The development has been a process of refinement, with theory enabling implementation but

implementation guiding, extending and verifying theory. While the implementation has been a *huge* effort, without it we would not have seen the need for simple polymorphism, for the finer treatment of algebraic data types, for usage specialisation, or for handling the constructs considered in Sections 6.4 and 6.5. Without implementation, the original *Once Upon a Type* analysis seemed sufficient, and indeed better than work that had gone before [TWM95a, §1.1]. With implementation, all these issues and more were forced upon us.

7.3 The development

We based our work on that of Turner, Mossin, and Wadler [TWM95a, TWM95b], but soon observed the limitations of their system: the problem of *poisoning*, where a function applied to an argument used many times would force its arguments at other call sites also to be treated as used many times even if the function in fact used them only once, and the restriction to a *toy language*. The problem of poisoning was easily addressed by use of *subsumption*. The extension to a *full language* was a more significant consideration, and became one of the major themes of this thesis.

The Glasgow Haskell Compiler – open-source and well-supported – was close at hand when we were considering this extension. A successful implementation in GHC would be solid proof that the analysis was applicable to a real, full-featured functional language and compiler, and it would give us the ability to measure the performance of the analysis on real code rather than the traditional `nfib` and `queens` benchmarks. Further, GHC already implements a large set of powerful optimisations, and we would be able to see the marginal benefit of usage analysis in the presence of these, rather than an unrealistic measurement of its performance on otherwise-unoptimised code. The output of usage analysis could also be used to guide the many optimisations already known to be able to make use of usage information.

7.3.1 Usage analysis

Obtaining good (or even just mediocre) results from a type-based usage analysis turned out to be much harder than we had expected. As already noted, we discovered that the lack of subsumption proved fatal to the analysis of Turner *et al.* [TWM95a, TWM95b]. But even with subsumption, we discovered that just two thunks in the entire standard libraries of GHC were marked as used at most once! This disappointing result set us on the quest for a more powerful analysis, during which we made a number of important contributions.

An obvious extension to our analysis would have been to add full *constrained polymorphism* to the types used to describe usage. This powerful technique would likely have yielded good results (although the data type considerations below would still apply), but at considerable cost. At the time we were designing the analysis, a large body of work demonstrated that type inference in the presence of subtyping and polymorphism was expensive and unmanageable in practice. In our quest, therefore, we sought a balance between power and practicality, and constrained polymorphism was considered impractical.

The rejection of constrained polymorphism led us to consider a more restricted form, since polymorphism was plainly required in order to give the terms we were examining reasonable types. Naturally for us, but apparently novel, we chose to consider *simple polymorphism*: polymorphism *without* constraints, yet in a type system with subsumption. Insights derived from implementation were critical in the development of simple polymorphism, testing the correctness of the algorithm and providing motivating examples.

Examination of analysed programs still showed disappointing results, however, and focussed our attention on better handling of data structures, and the use of specialisation. *Specialisation* proved relatively straightforward to implement. *Algebraic data types* of the Haskell/ML variety, however, turned out to have had rather little attention since around 1978, and we were unable to discover much published on the matter of subtyping for such types. We also realised that there was a wide design space in terms of the level of detail with which data types were treated by the analysis. We already had a preliminary approach to this in place for our monomorphic implementation [WPJ99], but it was not until much later that we invented the *annotation schemes* used in Chapter 5 to abstract out the entire design space from the remainder of the analysis; these are part of a development of algebraic data types that should be applicable in a range of settings.

7.3.2 Implementing a type-based analysis

Not only is usage analysis a harder problem than we expected; *implementing* the usage analysis in GHC was surprisingly difficult. Adding a type-based analysis to a typed compiler intentionally designed as a testbed for new analyses seemed like a straightforward enough task, but in practice a number of issues meant that it was not at all straightforward.

The obvious course when implementing a type-based analysis in a typed compiler is to embed the types of the analysis into the type language of the compiler. This has the great attractions of *simplicity* and *directness*: there is no need to find somewhere to affix the annotations, and the theoretical inference and the practical implementation are near-identical. It also has the allure of *correctness*: well-typedness, and thus the accuracy of the analysis and of transformations with respect to the analysis, will be verified and preserved by all the mechanisms already present in the compiler to ensure accuracy of ordinary type information. We discovered over the course of this thesis that these benefits are much harder to grasp than they appear.

Embedding the types of the analysis into the type language of the compiler means *changing the data type of types* in the compiler. Types are wired very deeply into a type-based compiler: a large proportion of its actions are directed toward generating, interpreting, and preserving them. A change to the data type that represents them will therefore require alterations to a large fraction of the code in the compiler. While many of these changes will be minor, even such minor changes require the relevant code to be read and understood. The crucial issue, therefore, is that embedding the types of the analysis into the type language of the compiler requires *understanding and modifying most of its code*. For a compiler of reasonable size, this is a large task.

Furthermore, embedding the types of the analysis into the type language of the compiler not only aids correctness, it *enforces* correctness. Incorrect usage information does not merely cause a potential space leak or poor transformation; it is a *type error*. Since compilation involves multiple transformations of the code, each must either be understood in enough detail to discern its usage behaviour and enable correct annotations to be inserted, or be followed immediately by a clean-up inference pass to recalculate correct usage information. Since the latter is costly in terms of compile time, we wish to take the former route as often as possible. So embedding the types of the analysis into the type language of the compiler means *understanding every transformation it performs*, including working out its usage behaviour.

Of course, these activities will lead to interesting discoveries: the analysis design will cover in great detail *all* of the features of the language compiler, and many of these may not have received theoretical treatment before. Some of these will lead to elegant theoretical developments that would have been unlikely to be discovered any other way. The rest will, at the very least, provide an acid test for the theory: is it useful in practice, or does it break down? Theories may seem elegant and powerful without practical experience to reveal hidden limitations; practical experience is certainly capable of inspiring new, motivated theory.

Implementing an analysis involves more than just implementing the inference for the intermediate language. In order to use the results of the inference in GHC, the translation into *low-level intermediate code* (STG) had to be modified to mark the appropriate thunks single-entry, the *code generator* had to be modified to obey these marks, and the *runtime system* had to be modified to handle the new situation. Empirical verification and performance measurement required further changes to the code generator and runtime system. At the other end, because GHC performs typechecking of the user program in the source language rather than the intermediate language, correctly handling separate compilation required modification to the *source language representation* and the *typechecker*.

Having engaged in all of the activities above, we can state from experience that, even for a type system as simple as simple polymorphism, embedding the types of the analysis into the type language of a production compiler is not a good idea. The impact is just too great, and the effort involved in making it all work is more than we were able to supply, even after in some cases months of work. Instead, a less ambitious plan succeeded, using a type system *alongside* the existing one, attaching these types as annotations to language terms, trading correctness for implementability. The analysis types are checked only by the analysis itself, and are preserved only by transformations that have been appropriately extended; other transformations, and unhandled type and term forms, are simply approximated or allowed to corrupt the analysis. Inference is repeated where necessary to ensure correctness where it counts. This is the analysis as finally implemented in Chapter 6.

An alternative would have been to implement a new compiler from scratch, with the types of the analysis built in. This approach can be very successful, as for example in the regions inference of the ML Kit [BTV96]. But apart from being rather too large a project to undertake in a single thesis, this approach would not have achieved the goals stated at the head of this section, of demonstrating applicability to a real, full-featured lazy functional language and compiler and in conjunction with most other known optimisations for such languages.

7.4 Future work

Much work remains to be done in the areas of simple polymorphism and usage analysis.

Simple polymorphism. Simple polymorphism has been motivated by practical considerations, and the intuition and algorithm are both very intensional. Theoretically, we have very little understanding of what it is and how it behaves. Following on from the suggestions in Sections 4.5 and 4.8.4, we would like to be able to give a concise and extensional description of the type chosen by the closure algorithm and its relationship to the most general constrained polymorphic type for the same term. We would like to investigate the possible benefits of applying the heuristics of Section 4.7.2 when generalising.

The inference should also be related to the approximating inference of Nordlander (Section 4.8.2), and the use of non-rank-1 polymorphism (Section 4.7.4) and polymorphic recursion (Section 4.4.2) considered.

Constrained polymorphism. The recent work of Rehof and Fähndrich [RF01] and of Gustavsson and Svenningsson [GS01b], described in Section 4.8.1.6, suggests that full constrained polymorphism may not necessarily be as expensive as was thought. It would be extremely interesting to modify the type system and inference of the present analysis to incorporate one or other of these approaches, and to investigate the power and cost of the analysis in the context of our full implementation. It seems likely that most of the content of this thesis, for example the extension to general algebraic data types and type polymorphism, and the proof techniques and much of the soundness proofs themselves, would carry over directly.

Algebraic data types. Annotation schemes have been developed as a notation to capture the wide design space for annotation of algebraic data types. But only a few points in this space have been investigated. More annotation schemes should be designed and their effectiveness measured in practice; our design framework and implementation is general enough to allow this to be done without difficulty. The most interesting issue is how best to limit the number of usage arguments given to a data type or constructor, and how a limited number of arguments should be distributed over the available annotation positions. The present implementation uses an extremely *ad hoc* (albeit rather effective) approach; a more principled and less wasteful approach would be better. For example, in

```
data Shape = Square Id (Int, Int) Int
           | Triangle Id (Int, Int) (Int, Int) (Int, Int)
```

it is likely that the *Id* fields will be used identically, whether the object is a *Square* or a *Triangle*, and so they could share the same annotation.

Implementation. It is said that no program is ever complete, and this is certainly true in this case.

- A major issue to investigate is how best to make use of *usage information* within GHC. The evidence of Chapter 6 suggests that, while the simplifier is making use of the information provided by the analysis, the choices it is making are not always optimal, meaning that allocations and run time sometimes *increase* when usage analysis is enabled. This is not surprising, since until the analysis was implemented very little usage information was available to the simplifier, and tradeoffs made when one-shot lambdas were rare are likely to look different now they are much more common. These tradeoffs need to be revisited, and specifically the programs that run slower or use more heap as a result of usage analysis must be investigated and the causes traced.
- Similarly, more *investigation* should be performed with regard to the effectiveness of the analysis in detecting thunks used at most once. Why is it that for some programs the analysis detects as many as 93% of all used-once thunks, while for over a third of programs tested less than 1% are found? Whatever the answer, it is likely that improvements to the analysis will be discovered in the course of the investigation.
- *Usage specialisation* is important for obtaining good results from the analysis, but the way in which it is implemented at present is quite minimal. The choice of which specialisations to generate at the moment is simply to generate one specialisation with all arguments specialised to 1. Instead, the choice should be informed by the known uses of the function, and if appropriate, more than one specialisation should be generated. The right specialisations to generate should be investigated, in view of the tradeoff between code size and accuracy. Again, the implementation is sufficiently general to allow this without difficulty.
- A number of smaller points that were left unimplemented merely for lack of time have been listed in Section 6.10.3.

Analysis. The analysis currently addresses only usage, *i.e.*, used-at-most-once. In Appendix C we consider extending the analysis to include annotations taken from $\mathcal{P}(\mathbb{N})$, subsuming not only usage but strictness and absence as well. This system should be proven sound, but in addition an inference for it should be designed and shown strictly to generalise our existing usage analysis. It should be implemented, and its power compared with more traditional strictness and absence analyses. A unified usage, strictness, and absence analyser along the lines of the usage analysis presented could be placed within the Glasgow Haskell Compiler and subsume the existing strictness and absence analyser.

7.5 Concluding remarks

In this thesis we have demonstrated that simple usage polymorphism is a practical and reasonably effective analysis for inferring usage information for lazy functional languages, enabling optimisations that reduce the cost of lazy evaluation. While the results obtained are not dramatic, they are significant, and it seems likely that further investigation will lead to greater gains.

*Of making many books there is no end,
and much study is a weariness of the flesh.*

*The end of the matter;
all has been heard.¹*

¹Ecclesiastes 12:12b–13a (New Revised Standard Version).

Appendix A.

Index of Notation

A.1 Basic notation

Vectors. $\overline{x_i}$ denotes the vector (ordered sequence) $x_1 x_2 \dots x_n$. This notation is also used for more complex structures, such as bindings, case branches, *etc.*.

Substitution. $M[A/x]$ denotes the result of substituting A for every occurrence of the variable x in M . $M[\overline{A_i}/\overline{x_i}]$ denotes the simultaneous substitution of $\overline{A_i}$ for $\overline{x_i}$. $\phi = [\overline{A_i}/\overline{x_i}]$ names the given substitution, and $M[\phi]$ applies the named substitution to M .

Definition. $A \triangleq B$ defines A to be equal to B .

A.2 Languages

L_0	toy source language. <i>p.</i> 26.
LX	toy executable language. <i>p.</i> 31.
LXC	set of configurations for LX . <i>p.</i> 31.
LIX_0	toy source-instrumented executable language. <i>p.</i> 29.
$LIXC_0$	set of configurations for LIX_0 . <i>p.</i> 30.
LIX_1	toy monomorphic usage-annotated instrumented executable language. <i>p.</i> 42.

LIX_2	toy polymorphic usage-annotated instrumented executable language. <i>p.</i> 78.
FL_0	full source language. <i>p.</i> 129.
$FLIX_0$	full source-instrumented executable language. <i>p.</i> 132.
$FLIXC_0$	set of configurations for $FLIX_0$. <i>p.</i> 132.
$FLIX_2$	full polymorphic usage-annotated instrumented executable language. <i>p.</i> 136.

A.3 Alphabetic notation

A	source- or executable-language atom. <i>p.</i> 26.
a	usage-typed atom. <i>p.</i> 43.
$\alpha, \beta, \gamma, \delta$	type variable. <i>p.</i> 129.
$ann^\varepsilon(\sigma)$	set of ε -ve annotations of σ . <i>p.</i> 291.
$BadBinding$	set of binding update-flag error configurations. <i>p.</i> 34.
$BadValue$	set of value update-flag error configurations. <i>p.</i> 34.
$BlackHole$	set of black hole configurations. <i>p.</i> 34.
C	abstract machine configuration. <i>p.</i> 33.
C	constraint. <i>p.</i> 55.
χ	update flag variable. <i>p.</i> 29.
$Clos$	closure operation. <i>p.</i> 87.
CS	constraint solver. <i>p.</i> 57.
$DEFAULT$	default branch of a case statement. <i>p.</i> 178.
$demanded$	demands during evaluation. <i>p.</i> 50.
$\text{dom}(\Gamma)$	domain (bound variables) of type environment. <i>p.</i> 28.
$\text{dom}(H)$	domain (bound variables) of heap. <i>p.</i> 33.
$\text{dom}(S)$	domain (bound variables) of stack. <i>p.</i> 33.
E	non-shallow evaluation context (composition of R s). <i>p.</i> 277.
e	usage-typed expression. <i>p.</i> 43.
ε	empty stack. <i>p.</i> 30.
ε	polarity (variance) variable. <i>p.</i> 44, <i>p.</i> 78.
F	forbidden usage variables. <i>p.</i> 102.
f, g, h	term variable (function). <i>p.</i> 26.
$FreshLUB$	least upper bound. <i>p.</i> 55.
$ftv^\varepsilon(\psi)$	free ε -ve type variables of ψ . <i>p.</i> 146.
$fuw^\varepsilon(\psi)$	free ε -ve usage variables of ψ . <i>p.</i> 78, <i>p.</i> 146.

G	(polarised) usage variables to generalise. <i>p. 102.</i>
Γ	type environment. <i>p. 28.</i>
gfp	greatest fixed point operator. <i>p. 102.</i>
H	heap. <i>p. 33.</i>
i, j, k, l, m	index variable.
\mathcal{IT}_0	trivial instrumented translation (from L_0 or FL_0 to LIX_0 or $FLIX_0$). <i>p. 31.</i>
\mathcal{IT}_1	monomorphic usage analysis (translation from L_0 or FL_0 to LIX_1 or $FLIX_1$). <i>p. 53.</i>
\mathcal{IT}_2	polymorphic usage analysis (translation from L_0 or FL_0 to LIX_2 or $FLIX_2$). <i>p. 86.</i>
K_i	data constructor. <i>p. 129.</i>
κ	usage annotation. <i>p. 43.</i>
M	source- or executable-language term. <i>p. 26.</i>
m, n	index bound.
$\mu\alpha . t$	recursive type definition. <i>p. 154.</i>
n	integer. <i>p. 26.</i>
$occur$	syntactic occurrences of a variable in an expression. <i>p. 48.</i>
ω	usage annotation “possibly used many times”. <i>p. 43.</i>
$PClos$	closure operation proper. <i>p. 102.</i>
$Pess$	pessimisation. <i>p. 56, p. 92.</i>
ψ	either τ - or σ -type. <i>p. 43.</i>
R	shallow evaluation context. <i>p. 30.</i>
$R[M]$	filled shallow evaluation context. <i>p. 32.</i>
$\text{rng}(\Gamma)$	range (binder types) of type environment. <i>p. 56, p. 92.</i>
S	stack. <i>p. 33.</i>
S	substitution. <i>p. 55.</i>
s	usage projection type schemes. <i>p. 170.</i>
σ	usage type with topmost annotation. <i>p. 43.</i>
T	type constructor. <i>p. 129.</i>
T	unannotated usage projection type. <i>p. 170.</i>
T	placeholder for type constructor (usage projection types). <i>p. 170.</i>
t	source type. <i>p. 26.</i>
\mathcal{T}_0	trivial uninstrumented translation (from L_0 or FL_0 to LX or FLX). <i>p. 37, p. 161.</i>
τ	usage type without topmost annotation. <i>p. 43, p. 80.</i>
trans	translation of configuration to term. <i>p. 37.</i>
TransitiveClosure	constraint solver. <i>p. 102, p. 105.</i>

$TrivClos$	trivial (non-)closure algorithm. <i>p.</i> 101.
\mathcal{U}	set of equivalence classes of usage variables. <i>p.</i> 102.
U	annotated usage projection type. <i>p.</i> 170.
u, v, w, x	usage variable. <i>p.</i> 80.
V	source- or executable-language value. <i>p.</i> 30.
V	multiset of free term variables in inference. <i>p.</i> 55.
v	usage-typed value. <i>p.</i> 43.
$Value$	set of value configurations. <i>p.</i> 34.
$Wrong$	set of type-error configurations. <i>p.</i> 34.
x, y, z	term variable. <i>p.</i> 26.

A.4 Nonalphabetic notation

$g \circ f$	function composition, <i>i.e.</i> , $g(f(\cdot))$.
$S \not\sqsubset T$	S and T disjoint, <i>i.e.</i> , $S \cap T = \emptyset$. <i>p.</i> 33
$(c ? e_1 : e_2)$	conditional, <i>i.e.</i> , if c then e_1 else e_2 .
$\bullet, !$	update flags: “not updatable/copyable” and “updatable/copyable”. <i>p.</i> 29.
$1, \omega$	usage annotations (“used at most once” and “possibly used many times”). <i>p.</i> 43.
\natural	stripping (mapping from instrumented executable language to source language). <i>p.</i> 29.
\flat	erasure (mapping from instrumented to uninstrumented executable language). <i>p.</i> 29.
κ^\dagger	update flag corresponding to usage annotation. <i>p.</i> 47, <i>p.</i> 83.
$ V $	update flag of a value. <i>p.</i> 32, 5.4.
$\langle H; M; S \rangle$	abstract machine configuration. <i>p.</i> 33.
$[\cdot]$	hole. <i>p.</i> 30.
$x = e$	binding. <i>p.</i> 26.
$x : t$	x has type t . <i>p.</i> 26.
$\#x$	update frame. <i>p.</i> 33.
\emptyset	empty set.
\emptyset	empty heap. <i>p.</i> 30.
\emptyset	trivial constraint. <i>p.</i> 55.
$\{\}$	empty multiset. <i>p.</i> 55.
$\{\cdot\}$	multiset brackets. <i>p.</i> 55.
$V_1 \uplus V_2$	multiset union. <i>p.</i> 55.
$V_1 \sqcup V_2$	multiset lub. <i>p.</i> 55.

$V \setminus A$	deletion of set from multiset. <i>p.</i> 55.
$\Lambda u . e$	usage abstraction. <i>p.</i> 81.
$\Lambda \alpha . e$	type abstraction. <i>p.</i> 129.
$e \kappa$	usage application. <i>p.</i> 81.
$e \tau$	type application. <i>p.</i> 129.
$\forall u . \tau$	usage-generalised type. <i>p.</i> 78.
$\forall \alpha . \tau$	type-generalised type. <i>p.</i> 129.
\vdash	typing judgement. <i>p.</i> 28, <i>p.</i> 47, <i>p.</i> 83, <i>p.</i> 130, <i>p.</i> 137.
\vdash_b	constrained-polymorphic well-typing rules. <i>p.</i> 123.
\blacktriangleright_1	usage inference phase 1, in LIX_1 or $FLIX_1$. <i>p.</i> 53.
\blacktriangleright_2	usage inference phase 1, in LIX_2 or $FLIX_2$. <i>p.</i> 87, <i>p.</i> 159.
\blacktriangleright	data type annotation scheme. <i>p.</i> 148.
$y \in \Gamma$	y is a variable bound by Γ , <i>i.e.</i> , $y \in \text{dom}(\Gamma)$. <i>p.</i> 46.
Γ, Γ'	union of type environments. <i>p.</i> 28.
\sqsubseteq	goodness ordering on types and solution sets. <i>p.</i> 52.
\triangleright	guard. <i>p.</i> 51.
$\langle ; ; \rangle$	initial configuration operator. <i>p.</i> 34.
\mapsto	small-step transition relation. <i>p.</i> 32.
\mapsto^*	reflexive, transitive closure of \mapsto . <i>p.</i> 34.
\mapsto_δ	primitive reduction relation. <i>p.</i> 32.
\downarrow	terminates (with configuration). <i>p.</i> 34.
\leq	primitive usage annotation ordering. <i>p.</i> 43, <i>p.</i> 86.
\preccurlyeq	(usage) subtyping relation. <i>p.</i> 49, <i>p.</i> 86, <i>p.</i> 135.
\preceq	semantic subtyping relation. <i>p.</i> 95.
$\llbracket \sigma \rrbracket$	denotation of σ . <i>p.</i> 95.
$[t]_\sigma^\omega$	annotate t everywhere with ω , yielding a σ -type. <i>p.</i> 43.
$[t]_\tau^\omega$	annotate t everywhere with ω , yielding a τ -type. <i>p.</i> 43.
$[t]_\sigma^{\text{fresh}}$	annotate t everywhere with fresh usage variables, yielding a σ -type. <i>p.</i> 55.
$[t]_\tau^{\text{fresh}}$	annotate t everywhere with fresh usage variables, yielding a τ -type. <i>p.</i> 55.
$+, -$	polarities: positive/covariant, negative/contravariant. <i>p.</i> 44, <i>p.</i> 78.
$\bar{\varepsilon}$	sign negation. <i>p.</i> 78, <i>p.</i> 145.
$\varepsilon \cdot \varepsilon'$	sign multiplication. <i>p.</i> 145.
$\langle \kappa \leq \kappa' \rangle$	primitive inequality constraint. <i>p.</i> 52.
$\langle \kappa = \kappa' \rangle$	primitive equality constraint. <i>p.</i> 52.
$\{P \Rightarrow C\}$	C if P holds, \emptyset otherwise. <i>p.</i> 55.
$C \vdash^e D$	constraint C entails constraint D . <i>p.</i> 52.

$\vdash^e SC$	constraint C is solved by substitution S . <i>p.</i> 52.
\wedge	constraint combination. <i>p.</i> 55.
$\leq_C, \leq_C^\varepsilon$	partial order induced by constraint C (flipped if $\varepsilon = -$). <i>p.</i> 101.
$R^{\pm*}$	reflexive, symmetric, transitive closure, <i>i.e.</i> , $(R \cup R^{-1})^*$. <i>p.</i> 102.
A/R	quotient of set by relation. <i>p.</i> 102.
$[a]_R$	equivalence class of a under equivalence relation R . <i>p.</i> 102.
$A \setminus B$	set subtraction. <i>p.</i> 102.
$S _A$	restriction: substitution S restricted to domain A . <i>p.</i> 108.
$A \ni a$	is contained in, <i>i.e.</i> , $a \in A$. <i>p.</i> 154.
\star	placeholder for trivial or unknown type (usage projection types). <i>p.</i> 170.
$\{t\}$	the default projection: computing a safe usage projection type. <i>p.</i> 170.
\perp	bottom: no information or non-termination. <i>p.</i> 179.

Appendix B.

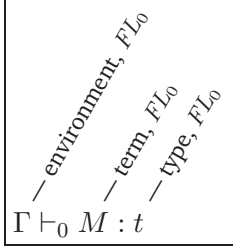
The Full Type System

In this appendix we collect all the definitions, well-typing rules, and inference rules for the full languages FL_0 , FLX , and $FLIX_2$.

Figure B.1 The full source language FL_0 .

Terms	$M ::=$	A $ $ n $ $ $K_i \overline{t_k} \overline{A_j}$ $ $ $\lambda x : t . M$ $ $ $M A$ $ $ $\Lambda \alpha . M$ $ $ $M t$ $ $ $\text{case } M : T \overline{t_k} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow M_i}$ $ $ $M_1 + M_2$ $ $ $\text{if0 } M \text{ then } M_1 \text{ else } M_2$ $ $ $\text{letrec } \overline{x_i : t_i = M_i} \text{ in } M$	atom literal (integer) constructor term abstraction term application type abstraction type application case expression primop (addition) zero-test conditional recursive let binding
Atoms	$A ::=$	x $ $ $A t$	term variable atom type application
t -types	$t ::=$	$t_1 \rightarrow t_2$ $ $ Int $ $ $T \overline{t_k}$ $ $ $\forall \alpha . t$ $ $ α	function type primitive type (integers) algebraic data type type-generalised type type variable
Decls	$T :$	$\text{data } T \overline{\alpha_k} = \overline{K_i \overline{t_{ij}}}$	algebraic data type declaration

Figure B.2 Well-typing rules for the source language FL_0 .



$$\frac{}{\Gamma, x : t \vdash_0 x : t} (\vdash_0\text{-VAR}) \quad \frac{}{\Gamma \vdash_0 n : \text{Int}} (\vdash_0\text{-LIT})$$

$$\frac{\Gamma \vdash_0 M : \text{Int} \quad \Gamma \vdash_0 M_i : t \quad i = 1, 2}{\Gamma \vdash_0 \text{if0 } M \text{ then } M_1 \text{ else } M_2 : t} (\vdash_0\text{-IF0})$$

$$\frac{\Gamma \vdash_0 M_i : \text{Int} \quad i = 1, 2}{\Gamma \vdash_0 M_1 + M_2 : \text{Int}} (\vdash_0\text{-PRIMOP})$$

$$\frac{\Gamma, x : t_1 \vdash_0 M : t_2}{\Gamma \vdash_0 \lambda x : t_1 . M : t_1 \rightarrow t_2} (\vdash_0\text{-ABS}) \quad \frac{\Gamma \vdash_0 M : t_1 \rightarrow t_2 \quad \Gamma \vdash_0 A : t_1}{\Gamma \vdash_0 M A : t_2} (\vdash_0\text{-APP})$$

$$\frac{\begin{array}{l} \Gamma, \overline{x_j : t_j} \vdash_0 M_i : t_i \quad \text{for all } i \\ \Gamma, \overline{x_j : t_j} \vdash_0 M : t \end{array}}{\Gamma \vdash_0 \text{letrec } \overline{x_i : t_i} = \overline{M_i} \text{ in } M : t} (\vdash_0\text{-LETREC})$$

$$\frac{\Gamma, \alpha \vdash_0 M : t \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash_0 \Lambda \alpha . M : \forall \alpha . t} (\vdash_0\text{-TYABS}) \quad \frac{\Gamma \vdash_0 M : \forall \alpha . t_1}{\Gamma \vdash_0 M t_2 : t_1[t_2/\alpha]} (\vdash_0\text{-TYAPP})$$

$$\frac{\begin{array}{l} t_{ij}^\circ = t_{ij}[\overline{t_k}/\overline{\alpha_k}] \quad \text{all } j \\ \Gamma \vdash_0 A_j : t_{ij}^\circ \quad \text{all } j \\ \text{where data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \end{array}}{\Gamma \vdash_0 K_i \overline{t_k} \overline{A_j} : T \overline{t_k}} (\vdash_0\text{-CON})$$

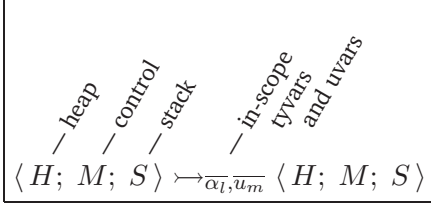
$$\frac{\begin{array}{l} \Gamma \vdash_0 M : T \overline{t_k} \\ t_{ij}^\circ = t_{ij}[\overline{t_k}/\overline{\alpha_k}] \quad \text{all } i, j \\ \Gamma, \overline{x_{ij} : t_{ij}^\circ} \vdash_0 M_i : t \quad \text{all } i \\ \text{where data } T \overline{\alpha_k} = \overline{K_i t_{ij}} \end{array}}{\Gamma \vdash_0 \text{case } M : T \overline{t_k} \text{ of } \overline{K_i x_{ij}} \rightarrow \overline{M_i} : t} (\vdash_0\text{-CASE})$$

Figure B.3 The polymorphically usage-typed language $FLIX_2$.

Terms	$e ::=$ a $ $ n $ $ $K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j}$ $ $ $\lambda^{\kappa, \chi} x : \sigma . e$ $ $ $e a$ $ $ $\Lambda \alpha . e$ $ $ $e \tau$ $ $ $\Lambda u . e$ $ $ $e \kappa$ $ $ $\text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i x_{ij}} \rightarrow e_i$ $ $ $e_1 + e_2$ $ $ $\text{add}_n e$ $ $ $\text{if0 } e \text{ then } e_1 \text{ else } e_2$ $ $ $\text{letrec } \overline{x_i : \sigma_i =^{\chi_i} e_i} \text{ in } e$	atom literal (integer) constructor term abstraction term application type abstraction type application usage abstraction usage application case expression primop (addition) partially-saturated primop zero-test conditional recursive let binding
Atoms	$a ::=$ x $ $ $a \tau$ $ $ $a \kappa$	term variable atom type application atom usage application
τ -types	$\tau ::=$ $\sigma_1 \rightarrow \sigma_2$ $ $ Int $ $ $T \overline{\kappa_l} \overline{\tau_k}$ $ $ $\forall \alpha . \tau$ $ $ α $ $ $\forall u . \tau$	function type primitive type (integer) algebraic data type type-generalised type type variable usage-generalised type
σ -types	$\sigma ::= \tau^\kappa$	usage-annotated type
Decls	$T : \text{data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \sigma_{ij}}$	data type declaration
Usage annotations	$\kappa ::=$ 1 $ $ ω $ $ u, v	used at most once possibly used many times usage variable
Update flags	$\chi ::=$ \bullet $ $ $!$	not updatable/copyable updatable/copyable

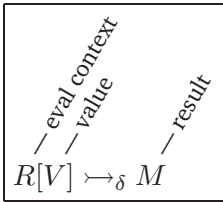
Figure B.4 The executable language $FLIX_2$ and configurations $FLIXC_2$.

Terms	e	$::=$	$R[e]$ $\text{letrec } \overline{x_i : \sigma_i =^{\chi_i} e_i} \text{ in } e$ $\Lambda \alpha . e$ $\Lambda u . e$ a V	filled evaluation context recursive let binding type abstraction usage abstraction atom value
Shallow evaluation contexts	R	$::=$	$[\cdot] a$ $[\cdot] \tau$ $[\cdot] \kappa$ $\text{case } [\cdot] : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of}$ $\quad \overline{K_i} \overline{x_{ij}} \rightarrow e_i$ $[\cdot] + e$ $\text{add}_n [\cdot]$ $\text{if0 } [\cdot] \text{ then } e_1 \text{ else } e_2$	term application type application usage application case expression primop (addition) partially-saturated primop zero-test conditional
Atoms	a	$::=$	x $a \tau$ $a \kappa$	term variable atom type application atom usage application
Values	V	$::=$	n $K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_k}$ $\lambda^{\kappa, \chi} x : \sigma . e$ $\Lambda \alpha . V$ $\Lambda u . V$	literal (integer) constructor term abstraction type abstraction of value usage abstraction of value
Configurations	C	$::=$	$\langle H; e; S \rangle$	where $\text{dom}(H) \not\subseteq \text{dom}(S)$
Heaps	H	$::=$	\emptyset $H, x : \sigma =^{\kappa, \chi} e$	where $x \notin \text{dom}(H)$
Stacks	S	$::=$	ε R, S $\#x : \sigma, S$	where $x \notin \text{dom}(S)$

Figure B.5 The full operational semantics of $FLIXC_2$.

$\langle H; R[e]; S \rangle$	\mapsto_{α_l, u_m}	$\langle H; e; R, S \rangle$	$(\mapsto\text{-UNWIND})$
$\langle H; V; R, S \rangle$	\mapsto_{α_l, u_m}	$\langle H; e; S \rangle$	$(\mapsto\text{-REDUCE})$
		if $R[V] \mapsto_\delta e$	
$\langle H; \text{letrec } \overline{x_i : \sigma_i =^{\chi_i} e_i} \text{ in } e; S \rangle$	\mapsto_{α_l, u_m}	$\langle H, y_i : \forall \overline{u_m} . \forall \overline{\alpha_l} . \sigma_i =^{\chi_i} \Lambda \overline{u_m} . \Lambda \overline{\alpha_l} . e_i[\phi]; e[\phi]; S \rangle$	$(\mapsto\text{-LETREC})$
		where $\overline{y_i} \not\in \text{dom}(H) \cup \text{dom}(S)$	
		$\phi = [\overline{y_i} \ \overline{u_m} \ \overline{\alpha_l} / \overline{x_i}]$	
$\langle H; \Lambda \alpha . e; S \rangle$	\mapsto_{α_l, u_m}	$\langle H'; \Lambda \alpha' . e'; S' \rangle$	$(\mapsto\text{-TYLAM})$
		if $\langle H; e[\alpha'/\alpha]; S \rangle \mapsto_{\alpha_l, \alpha', \overline{u_m}} \langle H'; e'; S' \rangle$	
		α' fresh	
$\langle H; \Lambda u . e; S \rangle$	\mapsto_{α_l, u_m}	$\langle H'; \Lambda u' . e'; S' \rangle$	$(\mapsto\text{-ULAM})$
		if $\langle H; e[u'/u]; S \rangle \mapsto_{\alpha_l, \overline{u_m}, u'} \langle H'; e'; S' \rangle$	
		u' fresh	
$\langle H, x : \sigma =^\bullet e; x; S \rangle$	\mapsto_{α_l, u_m}	$\langle H; e; S \rangle$	$(\mapsto\text{-VAR-ONCE})$
$\langle H, x : \sigma =^! e; x; S \rangle$	\mapsto_{α_l, u_m}	$\langle H; e; \#x : \sigma, S \rangle$	$(\mapsto\text{-VAR-MANY})$
$\langle H; V; \#x : \sigma, S \rangle$	\mapsto_{α_l, u_m}	$\langle H, x : \sigma =^! V; V; S \rangle$	$(\mapsto\text{-UPDATE})$
		where $ V = \bullet \Rightarrow x \notin \text{fv}(H, V, S)$	

where $ n $	$= \omega$
$ K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j} $	$= \kappa$
$ \lambda^{\kappa, \chi} x : \sigma . e $	$= \kappa$
$ \Lambda \alpha . V $	$= V $
$ \Lambda u . V $	$= V $



$(\lambda^{\kappa, \chi} x : \sigma . e) a$	\mapsto_δ	$e[a/x]$	$(\mapsto_\delta\text{-APP})$
$(\Lambda \alpha . e) \tau$	\mapsto_δ	$e[\tau/\alpha]$	$(\mapsto_\delta\text{-TYAPP})$
$(\Lambda u . e) \kappa$	\mapsto_δ	$e[\kappa/u]$	$(\mapsto_\delta\text{-UAPP})$
$n + e$	\mapsto_δ	$\text{add}_n e$	$(\mapsto_\delta\text{-PRIMOP-L})$
$\text{add}_{n_1} n_2$	\mapsto_δ	n_3	if $n_3 = n_1 + n_2$ $(\mapsto_\delta\text{-PRIMOP-R})$
$\text{case } K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j} : (T \overline{\kappa'_l} \overline{\tau'_k})^{\kappa'} \text{ of } K_i \overline{x_{ij}} \rightarrow e_i$	\mapsto_δ	$e_i[\overline{a_j}/\overline{x_{ij}}]$	$(\mapsto_\delta\text{-CASE})$
$\text{if0 } n \text{ then } e_1 \text{ else } e_2$	\mapsto_δ	e_i	if $i = (n = 0 ? 1 : 2)$ $(\mapsto_\delta\text{-IF0})$

Figure B.6(a) Well-typing rules for $FLIX_2$.

$\Gamma \vdash_2 e : \sigma$
 \swarrow environment, LIX_2
 \swarrow term, LIX_2
 \swarrow type, LIX_2

$$\frac{}{\Gamma, x : \sigma \vdash_2 x : \sigma} (\vdash_2\text{-VAR}) \qquad \frac{}{\Gamma \vdash_2 n : \text{Int}^\omega} (\vdash_2\text{-LIT})$$

$$\frac{\Gamma \vdash_2 e : \text{Int}^1 \quad \Gamma \vdash_2 e_i : \sigma \quad i = 1, 2}{\Gamma \vdash_2 \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma} (\vdash_2\text{-IF0})$$

$$\frac{\Gamma \vdash_2 e_i : \text{Int}^1 \quad i = 1, 2}{\Gamma \vdash_2 e_1 + e_2 : \text{Int}^\omega} (\vdash_2\text{-PRIMOP}) \qquad \frac{\Gamma \vdash_2 e : \text{Int}^1}{\Gamma \vdash_2 \text{add}_n e : \text{Int}^\omega} (\vdash_2\text{-PRIMOP-R})$$

$$\frac{\begin{array}{l} \Gamma, x : \sigma_1 \vdash_2 e : \sigma_2 \\ \text{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \text{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \quad \text{for all } y \in \Gamma \end{array}}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-ABS})$$

$$\frac{\Gamma \vdash_2 e : (\sigma_1 \rightarrow \sigma_2)^1 \quad \Gamma \vdash_2 a : \sigma_1}{\Gamma \vdash_2 e a : \sigma_2} (\vdash_2\text{-APP})$$

$$\frac{\begin{array}{l} \Gamma, \overline{x_j : \sigma_j} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j} \vdash_2 e : \sigma \\ \left(\text{occur}(x_i, e) + \sum_{j=1}^n \text{occur}(x_i, e_j) \right) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \vdash_2 \text{letrec } x_i : \sigma_i =_{|\sigma_i|^\dagger} e_i \text{ in } e : \sigma} (\vdash_2\text{-LETREC})$$

Figure B.6(b) Well-typing rules for $FLIX_2$.

$$\begin{array}{c}
\frac{\Gamma \vdash_2 e : \sigma' \quad \sigma' \preceq \sigma}{\Gamma \vdash_2 e : \sigma} \text{ (}\vdash_2\text{-SUB)} \\[10pt]
\frac{\Gamma, u \vdash_2 e : \tau^\kappa \quad u \notin (fuv(\Gamma) \cup fuv(\kappa))}{\Gamma \vdash_2 \Lambda u . e : (\forall u . \tau)^\kappa} \text{ (}\vdash_2\text{-UABS)} \quad \frac{\Gamma \vdash_2 e : (\forall u . \tau)^\kappa}{\Gamma \vdash_2 e \kappa' : (\tau[\kappa'/u])^\kappa} \text{ (}\vdash_2\text{-UAPP)} \\[10pt]
\frac{\Gamma, \alpha \vdash_2 e : \tau^\kappa \quad \alpha \notin ftv(\Gamma)}{\Gamma \vdash_2 \Lambda \alpha . e : (\forall \alpha . \tau)^\kappa} \text{ (}\vdash_2\text{-TYABS)} \quad \frac{\Gamma \vdash_2 e : (\forall \alpha . \tau_1)^\kappa}{\Gamma \vdash_2 e \tau_2 : (\tau_1[\tau_2/\alpha])^\kappa} \text{ (}\vdash_2\text{-TYAPP)} \\[10pt]
\frac{\begin{array}{l} \sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } j \\ \Gamma \vdash_2 a_j : \sigma_{ij}^\circ \quad \text{all } j \\ |\sigma_{ij}^\circ| \leq \kappa \quad \text{all } j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}} \end{array}}{\Gamma \vdash_2 K_i^{\kappa, \kappa^\dagger} \overline{\kappa_l} \overline{\tau_k} a_j : (T \overline{\kappa_l} \overline{\tau_k})^\kappa} \text{ (}\vdash_2\text{-CON)} \\[10pt]
\frac{\begin{array}{l} \Gamma \vdash_2 e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \\ \sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } i, j \\ \Gamma, x_{ij} : \sigma_{ij}^\circ \vdash_2 e_i : \sigma \quad \text{all } i \\ occur(x_{ij}, e_i) > 1 \Rightarrow |\sigma_{ij}^\circ| = \omega \quad \text{all } i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}} \end{array}}{\Gamma \vdash_2 \text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i} \overline{x_{ij}} \rightarrow e_i : \sigma} \text{ (}\vdash_2\text{-CASE)} \\[10pt]
\begin{array}{l} \text{where } 1^\dagger \triangleq \bullet \\ \omega^\dagger \triangleq ! \\ u^\dagger \triangleq ! \end{array}
\end{array}$$

Figure B.7 Translation function and well-typing rule for $FLIXC_2$ contexts.

$$\frac{\emptyset \vdash_2 \text{trans}\langle H; e; S \rangle : \sigma}{\vdash_2 \langle H; e; S \rangle : \sigma} \text{ (}\vdash_2\text{-CONFIG)}$$

$$\begin{array}{ll}
\text{trans}\langle H; e; R, S \rangle & = \text{trans}\langle H; R[e]; S \rangle \\
\text{trans}\langle H; e; \#x : \sigma, S \rangle & = \text{trans}\langle H, x : \sigma =^! e; x; S \rangle \\
\text{trans}\langle H; e; \varepsilon \rangle & = \text{letrec } H \text{ in } e
\end{array}$$

Figure B.8 The subtype (\preccurlyeq) and primitive (\leq) orderings over $FLIX_2$.

$$\boxed{\psi \preccurlyeq \psi}$$

$$\frac{\kappa_1 \leq \kappa_2 \quad \tau_1 \preccurlyeq \tau_2}{\tau_1^{\kappa_1} \preccurlyeq \tau_2^{\kappa_2}} (\preccurlyeq\text{-ANNOT}) \quad \frac{\tau_1 \preccurlyeq \tau_2}{\forall u . \tau_1 \preccurlyeq \forall u . \tau_2} (\preccurlyeq\text{-ALL-U})$$

$$\frac{}{\text{Int} \preccurlyeq \text{Int}} (\preccurlyeq\text{-LIT}) \quad \frac{\sigma_3 \preccurlyeq \sigma_1 \quad \sigma_2 \preccurlyeq \sigma_4}{\sigma_1 \rightarrow \sigma_2 \preccurlyeq \sigma_3 \rightarrow \sigma_4} (\preccurlyeq\text{-ARROW})$$

$$\frac{\tau_1 \preccurlyeq \tau_2}{\forall \alpha . \tau_1 \preccurlyeq \forall \alpha . \tau_2} (\preccurlyeq\text{-ALL}) \quad \frac{}{\alpha \preccurlyeq \alpha} (\preccurlyeq\text{-TYVAR})$$

$$\frac{\begin{array}{l} \alpha_k \in \text{ftv}^\varepsilon(\sigma_{ij}) \Rightarrow \tau_k \preccurlyeq^\varepsilon \tau'_k \text{ for all } k, \varepsilon, i, j \\ u_l \in \text{fuv}^\varepsilon(\sigma_{ij}) \Rightarrow \kappa_l \leq^\varepsilon \kappa'_l \text{ for all } l, \varepsilon, i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}} \end{array}}{T \overline{\kappa_l} \overline{\tau_k} \preccurlyeq T \overline{\kappa'_l} \overline{\tau'_k}} (\preccurlyeq\text{-TYCON})$$

$$\boxed{\kappa \leq \kappa}$$

$$\frac{}{\omega \leq \kappa} \quad \frac{}{\kappa \leq 1} \quad \frac{}{u \leq u}$$

$$\begin{array}{c} 1 \\ \uparrow \\ \omega \end{array}$$

Figure B.9 Positive and negative free occurrences.

$ftv^\varepsilon(\psi)$	$fu v^\varepsilon(\psi)$
$ftv^\varepsilon(\tau^u) = ftv(\tau)$	$fu v^\varepsilon(\tau^u) = fu v(u) \cup fu v(\tau)$
$ftv^\varepsilon(\text{Int}) = \emptyset$	$fu v^\varepsilon(\text{Int}) = \emptyset$
$ftv^\varepsilon(\sigma_1 \rightarrow \sigma_2) = ftv^\varepsilon(\sigma_1) \cup ftv^\varepsilon(\sigma_2)$	$fu v^\varepsilon(\sigma_1 \rightarrow \sigma_2) = fu v^\varepsilon(\sigma_1) \cup fu v^\varepsilon(\sigma_2)$
$ftv^\varepsilon(\forall \alpha . \tau) = ftv^\varepsilon(\tau) \setminus \{\alpha\}$	$fu v^\varepsilon(\forall \alpha . \tau) = fu v^\varepsilon(\tau)$
$ftv^\varepsilon(\forall u . \tau) = ftv^\varepsilon(\tau)$	$fu v^\varepsilon(\forall u . \tau) = fu v^\varepsilon(\tau) \setminus \{u\}$
$ftv^+(\alpha) = \{\alpha\}$	$fu v^\varepsilon(\alpha) = \emptyset$
$ftv^-(\alpha) = \emptyset$	$fu v^+(u) = \{u\}$
$ftv^\varepsilon(u) = \emptyset$	$fu v^-(u) = \emptyset$
$ftv^\varepsilon(T \overline{\kappa_l} \overline{\tau_k}) = \left\{ \alpha \mid \bigvee_{k,\varepsilon'} \left(\alpha \in ftv^{\varepsilon \cdot \varepsilon'}(\tau_k) \wedge \alpha_k \in \bigcup_{ij} ftv^{\varepsilon'}(\sigma_{ij}) \right) \right\}$	
$fu v^\varepsilon(T \overline{\kappa_l} \overline{\tau_k}) = \left\{ u \mid \begin{array}{l} \bigvee_{l,\varepsilon'} \left(u \in fu v^{\varepsilon \cdot \varepsilon'}(\kappa_l) \wedge u_l \in \bigcup_{ij} fu v^{\varepsilon'}(\sigma_{ij}) \right) \\ \vee \bigvee_{k,\varepsilon'} \left(u \in fu v^{\varepsilon \cdot \varepsilon'}(\tau_k) \wedge \alpha_k \in \bigcup_{ij} ftv^{\varepsilon'}(\sigma_{ij}) \right) \end{array} \right\}$	
where data $(T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}}$	

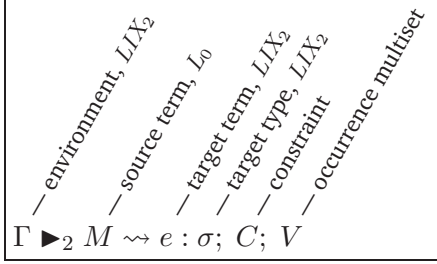
Figure B.10 Annotations.

$ann^\varepsilon(\psi)$

$ann^+(\tau^\kappa)$	$=$	$ann^+(\tau) \cup \{\kappa\}$
$ann^-(\tau^\kappa)$	$=$	$ann^-(\tau)$
$ann^\varepsilon(\text{Int})$	$=$	\emptyset
$ann^\varepsilon(\sigma_1 \rightarrow \sigma_2)$	$=$	$ann^\varepsilon(\sigma_1) \cup ann^\varepsilon(\sigma_2)$
$ann^\varepsilon(\forall \alpha . \tau)$	$=$	$ann^\varepsilon(\tau)$
$ann^\varepsilon(\forall u . \tau)$	$=$	$ann^\varepsilon(\tau) \setminus \{u\}$
$ann^\varepsilon(\alpha)$	$=$	\emptyset
$ann^\varepsilon(T \overline{\kappa_l} \overline{\tau_k})$	$=$	$\left\{ \kappa \mid \bigvee_{l,\varepsilon'} \left(\kappa \in ann^{\varepsilon \cdot \varepsilon'}(\kappa_l) \wedge u_l \in \bigcup_{ij} ann^{\varepsilon'}(\sigma_{ij}) \right) \right\}$

where data $(T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}}$

Figure B.11(a) Type inference rules from FL_0 to $FLIX_2$.



$$\frac{\text{fresh } \overline{v_i} \quad \tau \text{ a usage-monotype}}{\Gamma, x : (\forall \overline{u_i}. \tau)^\kappa \triangleright_2 x \rightsquigarrow x \overline{v_i} : (\tau[\overline{v_i}/\overline{u_i}])^\kappa; \emptyset; \{x\}} (\triangleright_2\text{-VAR})$$

$$\frac{}{\Gamma \triangleright_2 n \rightsquigarrow n : \text{Int}^\omega; \emptyset; \{ \}} (\triangleright_2\text{-LIT})$$

$$\frac{\begin{array}{l} \Gamma \triangleright_2 M \rightsquigarrow e : \text{Int}^\kappa; C; V \\ \Gamma \triangleright_2 M_i \rightsquigarrow e_i : \sigma_i; C_i; V_i \quad i = 1, 2 \\ (C_3, \sigma) = \text{FreshLUB}(\sigma_1, \sigma_2) \end{array}}{\Gamma \triangleright_2 \text{if0 } M \text{ then } M_1 \text{ else } M_2 \rightsquigarrow \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma; C \wedge C_1 \wedge C_2 \wedge C_3; V \uplus (V_1 \sqcup V_2)} (\triangleright_2\text{-IF0})$$

$$\frac{\Gamma \triangleright_2 M_i \rightsquigarrow e_i : \text{Int}^{\kappa_i}; C_i; V_i \quad i = 1, 2}{\Gamma \triangleright_2 M_1 + M_2 \rightsquigarrow e_1 + e_2 : \text{Int}^\omega; C_1 \wedge C_2; V_1 \uplus V_2} (\triangleright_2\text{-PRIMOP})$$

$$\frac{\begin{array}{l} \sigma_1 = [t_1]_\sigma^{\text{fresh}} \quad \text{fresh } v \\ \Gamma, x : \sigma_1 \triangleright_1 M \rightsquigarrow e : \sigma_2; C_1; V \\ C_2 = \{V(x) > 1 \Rightarrow \langle |\sigma_1| = \omega \rangle\} \\ C_3 = \bigwedge_{y \in \Gamma} \{V(y) > 0 \Rightarrow \langle |\Gamma(y)| \leq v \rangle\} \end{array}}{\Gamma \triangleright_1 \lambda x : t_1. M \rightsquigarrow \lambda^{v, v^\dagger} x : \sigma_1. e : (\sigma_1 \rightarrow \sigma_2)^v; C_1 \wedge C_2 \wedge C_3; V \setminus \{x\}} (\triangleright_2\text{-ABS})$$

$$\frac{\begin{array}{l} \Gamma \triangleright_2 M \rightsquigarrow e : (\sigma_1 \rightarrow \sigma_2)^\kappa; C_1; V_1 \\ \Gamma \triangleright_2 A \rightsquigarrow a : \sigma'_1; C_2; V_2 \\ C_3 = \{\sigma'_1 \preceq \sigma_1\} \end{array}}{\Gamma \triangleright_2 M A \rightsquigarrow e a : \sigma_2; C_1 \wedge C_2 \wedge C_3; V_1 \uplus V_2} (\triangleright_2\text{-APP})$$

$$\frac{\Gamma, \alpha \triangleright_2 M \rightsquigarrow e : \tau^\kappa; C; V \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \triangleright_2 \Lambda \alpha. M \rightsquigarrow \Lambda \alpha. e : (\forall \alpha. \tau)^\kappa; C; V} (\triangleright_2\text{-TYABS})$$

$$\frac{\Gamma \triangleright_2 M \rightsquigarrow e : (\forall \alpha. \tau_1)^\kappa; C; V \quad \tau_2 = [t]_\tau^{\text{fresh}}}{\Gamma \triangleright_2 M t \rightsquigarrow e \tau_2 : (\tau_1[\tau_2/\alpha])^\kappa; C; V} (\triangleright_2\text{-TYAPP})$$

Figure B.11(b) Type inference rules from FL_0 to $FLIX_2$.

$$\begin{array}{c}
\tau_i^{v_i} = \lceil t_i \rceil_\sigma^{fresh} \quad \text{for all } i \\
\\
\Gamma, \overline{x_j : \tau_j^{v_j}} \blacktriangleright_2 M_i \rightsquigarrow e_i : \sigma'_i; C_1^i; V_i \quad \text{for all } i \\
\\
C_1 = \bigwedge_i (C_1^i \wedge \{\sigma'_i \preceq \tau_i^{v_i}\}) \\
\\
(C_1', \overline{u_k}, S) = Clos(C_1, \Gamma, \overline{\tau_i^{v_i}}) \\
\\
\Gamma, \overline{x_j : (\forall \overline{u_k} . S \tau_j)^{v_j}} \blacktriangleright_2 M \rightsquigarrow e : \sigma; C_2; V \\
\\
C_3 = \bigwedge_i \{ (V(x_i) + \sum_j V_j(x_i)) > 1 \Rightarrow v_i = \omega \} \\
\hline
\Gamma \blacktriangleright_2 \text{ letrec } \overline{x_i : t_i = \overline{M_i}} \text{ in } M \quad (\blacktriangleright_2\text{-LETREC}) \\
\\
\rightsquigarrow \text{ letrec } \overline{x_i : (\forall \overline{u_k} . S \tau_i)^{v_i} =_{v_i}^\dagger \Lambda \overline{u_k} . S e_i[(x_j \overline{u_k})/\overline{x_j}]} \text{ in } e : \sigma; \\
\\
C_1' \wedge C_2 \wedge C_3; (\biguplus_i V_i \uplus V) \setminus \{\overline{x_i}\} \\
\\
\\
\begin{array}{c}
\tau_k = \lceil t_k \rceil_\tau^{fresh} \quad \text{for all } k \quad \text{fresh } v, \overline{v_l} \\
\sigma_{ij}^\circ = \sigma_{ij}[\overline{v}/u, \overline{v_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{for all } i \\
\Gamma \blacktriangleright_2 A_j \rightsquigarrow a_j : \sigma'_j; C_1^j; V_j \quad \text{for all } j \\
C_1 = \bigwedge_j (C_1^j \wedge \{\sigma'_j \preceq \sigma_{ij}^\circ\}) \\
C_2 = \bigwedge_j \{ |\sigma_{ij}^\circ| \leq v \} \\
\text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} \\
\hline
\Gamma \blacktriangleright_2 K_i \overline{t_k} \overline{A_j} \rightsquigarrow K_i^{v, v^\dagger} \overline{v_l} \overline{\tau_k} \overline{a_j} : (T \overline{v_l} \overline{\tau_k})^v; C_1 \wedge C_2; \biguplus_j V_j \quad (\blacktriangleright_2\text{-CON})
\end{array} \\
\\
\begin{array}{c}
\Gamma \blacktriangleright_2 M \rightsquigarrow e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa; C_1; V \\
\sigma_{ij}^\circ = \sigma_{ij}[\overline{\kappa}/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{for all } i, j \\
\Gamma, \overline{x_{ij} : \sigma_{ij}^\circ} \blacktriangleright_2 M_i \rightsquigarrow e_i : \sigma_i; C_2^i; V_i \quad \text{for all } i \\
C_2 = \bigwedge_i C_2^i \quad (C_3, \sigma) = FreshLUB(\overline{\sigma_i}) \\
C_4 = \bigwedge_{ij} \{ V_i(x_{ij}) > 1 \Rightarrow |\sigma_{ij}^\circ| = \omega \} \\
\text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} \\
\hline
\Gamma \blacktriangleright_2 \text{ case } M : T \overline{t_k} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow M_i} \quad (\blacktriangleright_2\text{-CASE}) \\
\rightsquigarrow \text{ case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i : \sigma; } \\
C_1 \wedge C_2 \wedge C_3 \wedge C_4; V \uplus (\biguplus_i V_i \setminus \{\overline{x_{ij}}\})
\end{array}
\end{array}$$

Figure B.12 The trivial non-closure operation.

$$\text{TrivClos}(C, \Gamma, \overline{\tau_i \kappa_i}) \triangleq (C, [], [])$$

Figure B.13 The closure operation.

$$\begin{array}{c} \text{initial constraint} \\ \text{initial environment} \\ \text{types to generalise} \end{array} \quad \begin{array}{c} \text{candidate variables} \\ \text{forbidden variables} \end{array} \quad \begin{array}{c} \text{final constraint} \\ \text{variables to generalise} \\ \text{unifying substitution} \end{array}$$

$$\text{Clos}(C_0, \Gamma, \overline{\tau_i \kappa_i}) \triangleq \text{PClos}(C_0, G_0, F_0) \triangleq (C', \overline{u_i}, S) \quad \text{where}$$

$$\begin{aligned} G_0 &= \{u^\varepsilon \mid u \in \text{fv}^\varepsilon(\overline{\tau_i})\} \\ F_0 &= \text{fv}(\overline{\kappa_i}) \cup \text{fv}(\Gamma) \end{aligned}$$

$$\begin{aligned} (C, S_0) &= \text{TransitiveClosure}(C_0) \\ G &= G_0 \setminus \text{dom}(S_0) \\ F &= F_0 \setminus \text{dom}(S_0) \end{aligned}$$

$$\begin{aligned} \Phi(A) &= G \cap \{u^-, v^+ \mid u \leq_C v \wedge u^- \in A \wedge v^+ \in A\} \\ &\quad \cap \{u^\varepsilon \mid u^\varepsilon \in A \wedge \neg \exists x \in (F \cup \{v \mid v^\varepsilon \in (G \setminus A)\}) . x \leq_C^\varepsilon u\} \\ G' &= \text{gfp}(\Phi) \end{aligned}$$

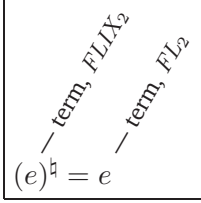
$$\begin{aligned} (\sim) &= \{(u, v) \mid u \leq_C v \wedge u^- \in G' \wedge v^+ \in G'\}^{\pm*} \\ &\quad \text{where } R^{\pm*} \triangleq (R \cup R^{-1})^* \\ \mathcal{U} &= \{u \mid u^\varepsilon \in G'\} / (\sim) \end{aligned}$$

$\overline{u_i}$ = a vector containing one representative
from each equivalence class in \mathcal{U}

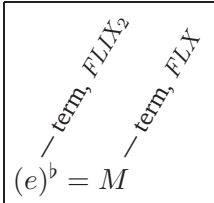
$$\begin{aligned} S = & \{(x \mapsto u_i) \mid \exists u^- \in G' . u \leq_C x \wedge \exists v^+ \in G' . x \leq_C v \wedge u_i \in [u]_{(\sim)}\} \\ & \cup \{(x \mapsto \omega) \mid \neg \exists u^- \in G' . u \leq_C x \wedge \exists v^+ \in G' . x \leq_C v\} \\ & \cup \{(x \mapsto 1) \mid \exists u^- \in G' . u \leq_C x \wedge \neg \exists v^+ \in G' . x \leq_C v\} \end{aligned}$$

where $x \in (\text{fv}(C) \setminus \text{dom}(S_0))$

$$C' = SC$$

Figure B.14 Stripping \natural and erasure \flat for $FLIX_2$.

$$\begin{aligned}
(x)^{\natural} &= x \\
(n)^{\natural} &= n \\
(K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j})^{\natural} &= K_i^{\kappa} \overline{\kappa_l} \overline{\tau_k} \overline{(a_j)^{\natural}} \\
(\lambda^{\kappa, \chi} x : \sigma . e)^{\natural} &= \lambda^{\chi} x : \sigma . (e)^{\natural} \\
(e a)^{\natural} &= (e)^{\natural} (a)^{\natural} \\
(\Lambda \alpha . e)^{\natural} &= \Lambda \alpha . (e)^{\natural} \\
(e \tau)^{\natural} &= (e)^{\natural} \tau \\
(\Lambda u . e)^{\natural} &= \Lambda u . (e)^{\natural} \\
(e \kappa)^{\natural} &= (e)^{\natural} \kappa \\
(\text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^{\kappa} \text{ of } \overline{K_i x_{ij}} \rightarrow e_i)^{\natural} &= \text{case } (e)^{\natural} : (T \overline{\kappa_l} \overline{\tau_k})^{\kappa} \text{ of } \overline{K_i x_{ij}} \rightarrow (e_i)^{\natural} \\
(e_1 + e_2)^{\natural} &= (e_1)^{\natural} + (e_2)^{\natural} \\
(\text{add}_n e)^{\natural} &= n + (e)^{\natural} \\
(\text{if0 } e \text{ then } e_1 \text{ else } e_2)^{\natural} &= \text{if0 } (e)^{\natural} \text{ then } (e_1)^{\natural} \text{ else } (e_2)^{\natural} \\
(\text{letrec } \overline{x_i : \sigma_i =_{\chi_i} e_i} \text{ in } e)^{\natural} &= \text{letrec } \overline{x_i : \sigma_i = (e_i)^{\natural}} \text{ in } (e)^{\natural}
\end{aligned}$$



$$\begin{aligned}
(x)^{\flat} &= x \\
(n)^{\flat} &= n \\
(K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j})^{\flat} &= K_i^{\chi} \overline{(a_j)^{\flat}} \\
(\lambda^{\kappa, \chi} x : \sigma . e)^{\flat} &= \lambda^{\chi} x . (e)^{\flat} \\
(e a)^{\flat} &= (e)^{\flat} (a)^{\flat} \\
(\Lambda \alpha . e)^{\flat} &= (e)^{\flat} \\
(e \tau)^{\flat} &= (e)^{\flat} \\
(\Lambda u . e)^{\flat} &= (e)^{\flat} \\
(e \kappa)^{\flat} &= (e)^{\flat} \\
(\text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^{\kappa} \text{ of } \overline{K_i x_{ij}} \rightarrow e_i)^{\flat} &= \text{case } (e)^{\flat} \text{ of } \overline{K_i x_{ij}} \rightarrow (e_i)^{\flat} \\
(e_1 + e_2)^{\flat} &= (e_1)^{\flat} + (e_2)^{\flat} \\
(\text{add}_n e)^{\flat} &= \text{add}_n (e)^{\flat} \\
(\text{if0 } e \text{ then } e_1 \text{ else } e_2)^{\flat} &= \text{if0 } (e)^{\flat} \text{ then } (e_1)^{\flat} \text{ else } (e_2)^{\flat} \\
(\text{letrec } \overline{x_i : \sigma_i =_{\chi_i} e_i} \text{ in } e)^{\flat} &= \text{letrec } \overline{x_i =_{\chi_i} (e_i)^{\flat}} \text{ in } (e)^{\flat}
\end{aligned}$$

Appendix C.

Extending the Lattice

Compilers for lazy functional languages usually perform *strictness analysis* to identify thunks and function arguments that are certainly used, enabling optimisations such as call-by-value evaluation, unboxing, and so on. Instead of treating strictness independently, however, it may be seen as a *usage* property: to say that an argument is strict is simply to say that it is used at least once.

Motivated by this observation, the present appendix considers extending our type-based usage analysis to a more detailed annotation domain that is capable of expressing strictness (and absence) in addition to used-at-most-once-ness. Such an analysis could subsume existing separate, non-type-based analyses, reducing the number of analysis passes and enabling optimisations to be directed simply by the computed usage types. The details of the analysis have not been fully worked out and so this appendix is somewhat speculative, documenting the current progress toward the goal.

We begin in Section C.1 by motivating the inclusion of a zero annotation to indicate “not used”. Section C.2 shows that incorporating the strict primitive `seq` into the language necessitates a distinction between the notions of demand and use, which have so far been considered identical. Having completed these explorations, in Section C.3 we present a language and an operational semantics to define the meaning of demand and use in this new context. Finally, Section C.4 sketches a demand/use type system for this language. We discuss the results of the appendix in Section C.5, and conclude with related work in Section C.6.

C.1 Absence, usage, and zero usage

Absence analysis (discovering whether an argument or a thunk is not used) clearly requires a 0-annotation, denoting “not used”. Consider the following expression:

$$\begin{array}{l} \text{letrec } a = e_1 \\ \quad b = e_2 \\ \text{in} \\ \text{letrec } p_1 = \lambda x . \lambda y . x \\ \text{in} \\ (p_1 \ a \ b) \end{array}$$

The usage type system so far described (\vdash_2 , Chapter 5) observes that a and b each occur once in their scope and assumes they will therefore be demanded at most once, annotating them with 1. In fact, however, p_1 ignores its second argument, and so b ’s thunk is not used at all. It would therefore be safe to avoid passing b to p_1 and to omit binding b entirely, thus:

$$\begin{array}{l} \text{letrec } a = e_1 \\ \text{in} \\ \text{letrec } p_1 = \lambda x . x \\ \text{in} \\ (p_1 \ a) \end{array} \rightsquigarrow$$

This optimisation is called the *absent argument transformation*, and to enable us to discover opportunities for it we must have an annotation 0 that may be inferred for absent arguments like y and dead bindings like b . While such code is not often written directly by programmers, it occurs frequently in the output of the *worker/wrapper transformation* which is used in GHC to exploit the results of strictness analysis, as discussed in [PJS98a, §6.4].

But it is not only the absent argument transformation that can benefit from a 0-annotation. The following extension of the example above shows that a 0-annotation can also improve the accuracy of simple update avoidance:

$$\begin{array}{l} \text{letrec } a = e_1 \\ \quad b = e_2 \\ \quad t = \text{Pair } a \ b \\ \text{in} \\ \text{letrec } p_1 = \lambda t . \text{case } t \text{ of Pair } x \ y \rightarrow x \\ \quad p_2 = \lambda t . \text{case } t \text{ of Pair } x \ y \rightarrow y \\ \text{in} \\ (p_1 \ t) + (p_2 \ t) \end{array}$$

Now t occurs twice, but its components a and b are each used only once. To infer this (and avoid updating their thunks) we must sum the uses arising from the two function applications, where p_1 uses the first component of its argument once and the second zero times, and p_2 uses the first component of its argument zero times and the second once. With a distinct annotation 0 the sum is $(1, 0) \oplus (0, 1) = (1, 1)$,

but with only 1 and ω annotations, the sum is $(1, 1) \oplus (1, 1) = (\omega, \omega)$. Therefore to infer accurate annotations for such examples we must allow a zero annotation.

As an aside, we observe from these examples that in the presence of zero-use arguments we can no longer rely on a purely syntactic occurrence function (*occur*, Section 3.3.3): the number of syntactic occurrences is no longer even a lower bound on the number of uses. Instead, we incorporate explicit linear-style environment manipulation in our well-typing rules (Section C.4; see also Section 3.9.7).

C.2 Strictness and the demand/use distinction

Strictness analysis determines whether a function's arguments will certainly be used, and may therefore be evaluated eagerly before the call. To perform this so-called call-by-value optimisation as a source-to-source transformation, the language must provide a means of forcing evaluation. This may be either a strict-let construct, or the seq primitive. We here follow Haskell [PH⁺96] and use seq.

The binary primitive seq strictly evaluates its first argument to weak head normal form, ignores the result, and returns its second argument. This introduces the possibility that a function-valued variable may be *demanded* but not *used*.¹ Consider this fragment:

$$\begin{array}{l} \text{letrec } f = (\text{letrec } z = e_1 \text{ in } \lambda x . x + z) \\ \text{in } \quad \text{seq } f (f \ 10) \end{array}$$

On execution, this first allocates a thunk for f . It then evaluates the seq, which begins by evaluating f . This allocates a thunk for z and returns the function $(\lambda x . x + z)$, updating the thunk for f with the function. The returned function is ignored, and $(f \ 10)$ is evaluated. Once again f is required; this time no allocation is done and the function is immediately applied to 10. This evaluation requires the value of z ; after the sum is computed the result is returned.

Notice that the value of f is demanded twice, but even though z is free in this function, its value is demanded only once; the thunk for f must be annotated $!$, but that for z may safely be annotated \bullet . This is because even though f is *demanded* twice, it is *used* only once. To obtain accurate results in the presence of seq, it is necessary to distinguish between demand and use in the semantics. We see this clearly when considering the strictness of the functions f_1 and f_2 in the following:

$$\begin{array}{l} f_1 = \lambda x . \text{letrec } g = \lambda y . x + y \text{ in } (g \ 3) \\ f_2 = \lambda x . \text{letrec } g = \lambda y . x + y \text{ in } (\text{seq } g \ 3) \end{array}$$

In both cases the value of g is *demanded* once, but whereas in f_1 the function is also *used* (applied) once, demanding the free variable x once and making f_1 strict, in f_2 the function is not used and x is not demanded, and so f_2 is non-strict. Crucially, seq demands its first argument but does not use it; without the distinction between the two f_2 would be annotated incorrectly, either marking f_2 strict instead of non-strict

¹This is certainly possible for non-functions also, but whereas only with seq are $\lambda x . \perp$ and \perp distinguishable, $K \perp$ and \perp are already distinguishable in *FLX*, with case. Thus for us the interesting behaviour of seq relates chiefly to functions.

or marking g dead when it is live. Both would lead to incorrect transformations and runtime errors.²

In summary,

- The concept *demand* applies to *heap bindings* (and function arguments); a heap binding for x is demanded whenever x appears in evaluation position during evaluation; its value is looked up in the heap at this point.
- The concept *use* applies to *values* (and generalises to expressions, where it applies to the ultimate value of the expression). A value may be either a literal, an abstraction or a constructed value. A literal is used when it is given as argument to a primop or conditional. An abstraction is used when it is applied; *i.e.*, by β -reduction. A constructed value is used when it is deconstructed; *i.e.*, by case-reduction.

We are primarily interested in demand: if we know a certain heap binding is used in a certain way we may optimise its generation and behaviour. But as we have seen, in order to calculate this we must track use also. And as a bonus, knowing the use facilitates certain program transformations too: for example, we may safely float a binding through a used-once lambda (Section 1.3.4). Thus calculating both is both useful and necessary. Below, in place of simple usage annotations κ we use pairs of demand δ and use ν annotations.

C.3 An operational semantics

We have now developed some intuition for what is required to handle zero usage:

- an *expanded annotation domain* including at least zero as well as 1, ω ;
- *explicit environment management* rather than syntactic occurrence counting;
- a *distinction between demand and use*.

We may therefore proceed to give an consistent and intuitive operational semantics.

C.3.1 The language

We begin by modifying the full language *FLX*, which we defined in Section 5.1.4, to incorporate the new usage annotations. Figure C.1 gives the new language, which

²Although it yields a convenient example, *seq* is not in fact necessary to demonstrate the distinction: the pathological case of short-circuit *letrec* bindings exposes related behaviour. Consider the program fragment:

$$\begin{array}{l} \text{letrec } f = g \\ \quad g = (\text{letrec } z = e_1 \text{ in } \lambda x . x + z) \\ \text{in } (f\ 3) + (f\ 4) \end{array}$$

Here the function g will be demanded only once (after the first application, f will be updated with its value), but its value will be used twice. Thus f and z must be annotated $!$, but g may be annotated \bullet . Such short-circuit bindings are pathological, however, because they will be quickly removed by the simplifier in most cases.

Figure C.1 The extended-annotated executable language *ELX*.

Terms	$M ::=$	A $ n^\nu$ $ K_i^\nu \overline{A_k}$ $ \lambda^\nu x . M$ $ M A$ $ \text{case } M \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow M_i}$ $ M_1 + M_2$ $ \text{add}_n M$ $ \text{if0 } M \text{ then } M_1 \text{ else } M_2$ $ \text{seq } M_1 M_2$ $ \text{letrec } x_i =_{\delta_i, \nu_i} M_i \text{ in } M$	atom literal (integer) constructor term abstraction term application case expression primop (addition) partially-saturated primop conditional strict sequencing recursive let binding
Atoms	$A ::=$	x	term variable
Demands	$\delta \in$	\mathcal{D}	permitted demand
Uses	$\nu \in$	\mathcal{D}	permitted use

we call *ELX* (Extended eXecutable Language). We consider the uninstrumented language directly, without types, to avoid clutter and because these are irrelevant to our purposes here.

The primitive strict sequencing operation $\text{seq } M_1 M_2$ evaluates M_1 to a value, ignores it, and returns the result of evaluating M_2 .

Demand annotations are denoted δ and usage annotations ν . Both are taken from an annotation domain \mathcal{D} to be defined below (Section C.3.2), and together they replace the update flags $\chi \in \{\bullet, !\}$.

Demands appear only on bindings, because they only apply to thunks. Usages appear on *all* values, unlike *FLX* where they appear on abstractions only, so that all usages may be tracked. Usages also appear on bindings; these apply to the values of the bindings. The annotations describe the demand and/or use that the annotated object *permits*; the operational semantics will be arranged so that an attempt to demand/use the object other than as permitted by the annotation is an error.

Otherwise the language is identical to *FLX*.

C.3.2 The annotation domain and operations

The most immediately obvious domain for our annotations is $\mathcal{P}(\mathbb{N})$. Usage and demand operationally occur a natural number of times, and we take the powerset to handle the (static) nondeterminism introduced by conditional and case expressions. An annotation $x \in \mathcal{P}(\mathbb{N})$ permits use m times iff $m \in x$. Infinite subsets are required to annotate recursive functions.

Figure C.2 The annotation domain and operations.

Symbol	Name	Definition	Intended meaning: permits...
\mathcal{D}		$\mathcal{P}(\mathbb{N})$	(annotation domain)
\perp	<i>bottom</i>	\mathbb{N}	any use
\emptyset	<i>zero</i>	$\{0\}$	exactly zero use
$\mathbb{1}$	<i>one</i>	$\{1\}$	exactly one use
\top	<i>top</i>	\emptyset	(overconstrained, impossible)
$x \oplus y$	<i>plus</i>	$\{m + n \mid m \in x \wedge n \in y\}$	x then y
$x \ominus y$	<i>minus</i>	$\bigcap_{n \in y} \{m - n \mid m \in x \wedge m \geq n\}$	remaining of x after y
$x \leq y$	<i>permits</i>	$x \supseteq y$	(true if x permits exact use y)
<hr/>			
$x \cdot y$	<i>times</i>	$\{\sum_{i=1}^m n_i \mid m \in x \wedge \forall i. n_i \in y\}$	x copies of y
$x \wedge y$	<i>and</i>	$x \cup y$	x and y
$x \triangleright y$	<i>guard</i>	$\bigcup_{m \in x} (m = 0 ? \{0\} : y)$	\emptyset if $x \leq \emptyset$ and y if x permits non-zero

This is not the only possible annotation domain, however, and so we present the annotations and the operations over them as an algebra, with $\mathcal{P}(\mathbb{N})$ as one model. We do not give the equations, and so the algebra itself is only sketched; the model is however fully defined. The operational semantics and type system are defined over this algebra, allowing the model to be changed without affecting correctness.

The constants and operations are given in Figure C.2, along with the definitions for the $\mathcal{P}(\mathbb{N})$ model. Operations below the dashed line are required only for the well-typing rules, and are not used in the operational semantics. We now consider the operations in detail.

- $\mathcal{D}, \perp, \emptyset, \mathbb{1}, \top$ are straightforward.
- Addition $x \oplus y$ permits any possible use in x followed by any possible use in y . It is modelled by a summed cross-product of possible uses from x and y .
- Subtraction $x \ominus y$ denotes the number of uses that are certainly permitted of x after y uses have occurred. It is modelled by subtracting each use permitted by y from x , and then taking the intersection of the resulting sets.
- The relation $x \leq y$ holds if x permits use y , that is, if every use in y is also in x . It is modelled by the superset relation.

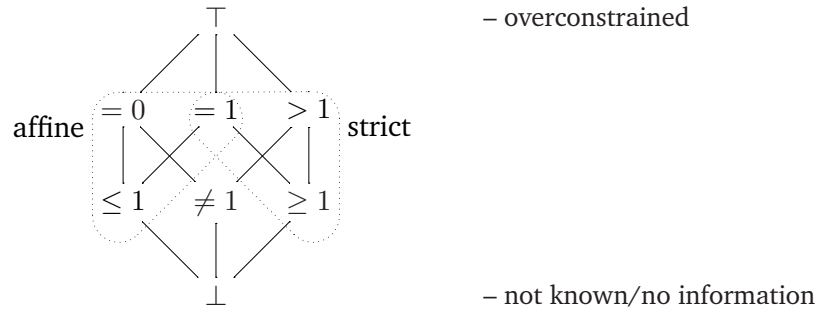
Note that “ $x \leq y$ ” is not the same as “ $x \ominus y$ is not equal to \top ”; the former means that (*exactly y and no more*) is permitted, whereas the latter means that (*y and possibly some more use*) is permitted.

- Product $x \cdot y$ denotes a sum of x copies of use y . Notice that this is in general asymmetric.

- Conjunction $x \wedge y$ permits use x and permits use y .³ This is the union of the possibilities offered by x and y .
- Guard $x \triangleright y$ permits zero use if x permits zero use, and permits use y if x permits non-zero use.

The algebra admits models other than $\mathcal{P}(\mathbb{N})$. For example, the system of the rest of the thesis may be obtained by setting $\mathcal{D} = \{1, \omega\}$ and making the obvious remaining identifications: $\perp = \omega$, $0 = 1 = 1$, etc.⁴ \top has no model in this domain, but since it never occurs in well-typed or well-annotated terms, this causes no difficulty (it may be added without affecting the system of the rest of the thesis). The definition of the operations for $\{1, \omega\}$ is left as an easy exercise. Notice that \leq is just the familiar primitive ordering $\omega \leq 1$ from Section 3.3.5.

Another possible model is the Bierman lattice [Bie91, Bie92], which represents affine (used at most once), strict (used at least once), and absent (not used) usage in a single annotation domain.



The Bierman lattice is attractive from an implementation point of view because it is small and finite (only seven non- \top annotations), and yet it distinguishes all the useful usage-like behaviour of a binding or expression. One hope for the future of our work is that a Bierman-lattice-based usage analysis will replace GHC's present backwards strictness/absence analyser.

C.3.3 The operational semantics

The operational semantics is based on the *FLX* semantics of earlier chapters, but we replace the coarse $\{\bullet, !\}$ distinction with annotations from \mathcal{D} .

For the purposes of the operational semantics, we set up syntactic categories as before, as shown in Figure C.3 (cf. Figures 2.3 and 5.3). The new primitive *seq* introduces a new shallow evaluation context *seq* $[\cdot] M$, which defers the second argument while evaluating the first. Heap bindings and update frames take demand and usage annotations; the annotations on update frames are those that *will* apply

³Dually, this could have been called “or” because it permits use x or y ; we call it “and” because the decision is made by the user, not by the annotation. Perhaps $x \& y$ (linear “with”) would be a better notation.

⁴The operational semantics (Figure C.4) must be changed slightly to omit pushing an update frame if the allowable demand or use would be 0.

Figure C.3 Syntactic categories for the *ELX* operational semantics.

Terms	$M ::= R[M]$ $\quad \text{ letrec } x_i =_{\delta_i, \nu_i} \overline{M_i} \text{ in } M$ $\quad A$ $\quad V^\nu$	
Shallow evaluation contexts	$R ::= [\cdot] A$ $\quad \text{ case } [\cdot] \text{ of } \overline{K_i x_{ij}} \rightarrow \overline{M_i}$ $\quad [\cdot] + M$ $\quad \text{ add}_n [\cdot]$ $\quad \text{ if0 } [\cdot] \text{ then } M_1 \text{ else } M_2$ $\quad \text{ seq } [\cdot] M$	
Atoms	$A ::= x$	
Values	$V^\nu ::= n^\nu$ $\quad K_i^\nu \overline{A_k}$ $\quad \lambda^\nu x . M$	
Demands	$\delta \in \mathcal{D}$	
Uses	$\nu \in \mathcal{D}$	
Configurations	$C ::= \langle H; M; S \rangle$	where $\text{dom}(H) \not\subseteq \text{dom}(S)$
Heaps	$H ::= \emptyset$ $\quad H, x =_{\delta, \nu} M$	where $x \notin \text{dom}(H)$
Stacks	$S ::= \varepsilon$ $\quad R, S$ $\quad \#^{\delta, \nu} x, S$	where $x \notin \text{dom}(S)$

to the updated binding (as distinct from those that applied to the binding before lookup). The remainder should be entirely unsurprising.

The operational semantics itself appears in Figure C.4. It consists of a transition relation $C \mapsto C'$ from configurations to configurations, defined in terms of a primitive transition relation $R[V] \mapsto_\delta M$ which performs the basic computations. To capture ill-annotated terms, we replace the failure configuration sets *BadBinding* and *BadValue* with the new configuration sets *BadDemand* and *BadUsage*, which denote an incorrect demand annotation and an incorrect usage annotation, respectively.⁵ The primitive transition relation is unchanged from previously, apart from the addition of the obvious rule for seq. The new notation \perp is used instead of ω to annotate the result of a primop application (\mapsto -PRIMOP-R). Of the transition rules, (\mapsto -UNWIND) and (\mapsto -LETREC) are unchanged. The heart of the demand-and-use story is contained in the three rules (\mapsto -LOOKUP) (combining the former (\mapsto -VAR-ONCE) and (\mapsto -VAR-MANY)), (\mapsto -UPDATE), and (\mapsto -REDUCE). We consider these one at a time. Recall that in a configuration $\langle H; M; S \rangle$ the term M is called the *control* (Section 2.4.1).

- Rule (\mapsto -LOOKUP) describes what happens when a binding is demanded. The variable x is used to look up the binding in the heap, and its right-hand side is placed in the control. Simultaneously, an update frame is pushed on the stack which will reinstate the binding once it is reduced to a value. The demand and usage annotations on the binding are updated by computing the new values and placing these on the update frame.

It is the looking-up of a binding that constitutes a demand; thus we subtract exactly one demand from the permitted demand δ of the binding: $\delta \ominus \mathbb{1}$, where $\mathbb{1}$ is the “exactly one” element of \mathcal{D} .

We must also remove the usage of the value retrieved from the usage annotation ν . The value has been placed in the control of the configuration, and so we may determine its usage by examining the stack. This is the purpose of the auxiliary function $use(S)$: if a shallow evaluation context R is on top, the usage is given by $use(R)$ and the remainder of the stack is irrelevant; if an update frame is on top then the usage it will allow to the heap binding is added and the usage of the remainder of the stack determined recursively; and if the stack is empty, no usage is made at all.⁶ The usage that will be made of the value is removed from the initial permitted usage ν to give the remaining usage permitted to the updated binding.

⁵ *Value*, *BlackHole*, and *Wrong* are just as in Section 2.4.3, *mutatis mutandis*. The remaining stuck configurations are defined as follows. If a configuration C' is of the form $\langle H, x =^{\delta, \nu} M; x; S \rangle$ where $\delta \ominus \mathbb{1} = \top$, or of the form $\langle x_i =^{\delta_i, \nu_i} M_i; V^\nu; \varepsilon \rangle$ where it is not the case that for all i , $\delta_i \leq 0$, then $C' \in \text{BadDemand}$ (this indicates an incorrect demand annotation). If a configuration C' is of the form $\langle H, x =^{\delta, \nu} M; x; S \rangle$ where $\nu \ominus use(S) = \top$, or of the form $\langle H; V^{\nu_1}; \#^{\delta_2, \nu_2} x, S \rangle$ where $\nu_1 \ominus \nu_2 = \top$, or of the form $\langle H; V^\nu; R, S \rangle$ where $\neg(\nu \leq use(R))$, or of the form $\langle x_i =^{\delta_i, \nu_i} M_i; V^\nu; \varepsilon \rangle$ where either $\neg(\nu \leq 0)$ or it is not the case that for all i , $\nu_i \leq 0$, then $C' \in \text{BadUsage}$ (this indicates an incorrect usage annotation).

⁶ It is assumed that the toplevel makes no use of the final result of a program; this is reasonable because a realistic program will communicate its result to the outside world by means of side-effects beyond the scope of this semantics.

Figure C.4 The operational semantics of *ELX* with auxiliary functions $use(S)$, $use(R)$.

$$\boxed{\langle H; M; S \rangle \rightarrow \langle H; M; S \rangle}$$

$$\langle H; R[M]; S \rangle \rightarrow \langle H; M; R, S \rangle \quad (\rightarrow\text{-UNWIND})$$

$$\langle H; V^\nu; R, S \rangle \rightarrow \langle H; M; S \rangle \quad (\rightarrow\text{-REDUCE})$$

if $\nu \leq use(R)$ and $R[V^\nu] \rightarrow_\delta M$

$$\langle H; \text{letrec } \overline{x_i =^{\delta_i, \nu_i} M_i} \text{ in } M; S \rangle \rightarrow \langle H, y_i =^{\delta_i, \nu_i} M_i[\phi]; M[\phi]; S \rangle \quad (\rightarrow\text{-LETREC})$$

where $\overline{y_i} \not\subseteq (\text{dom}(H) \cup \text{dom}(S))$
 $\phi = [\overline{y_i}/\overline{x_i}]$

$$\langle H, x =^{\delta, \nu} M; x; S \rangle \rightarrow \langle H; M; \#^{\delta \ominus 1, \nu \ominus use(S)} x, S \rangle \quad (\rightarrow\text{-LOOKUP})$$

if $\delta \ominus 1 \neq \top$ and $\nu \ominus use(S) \neq \top$

$$\langle H; V^{\nu_1}; \#^{\delta_2, \nu_2} x, S \rangle \rightarrow \langle H, x =^{\delta_2, \nu_2} V^{\nu_2}; V^{\nu_1 \ominus \nu_2}; S \rangle \quad (\rightarrow\text{-UPDATE})$$

if $\nu_1 \ominus \nu_2 \neq \top$

$$\boxed{R[V^\nu] \rightarrow_\delta M}$$

$$(\lambda^\nu x . M) A \rightarrow_\delta M[A/x] \quad (\rightarrow_\delta\text{-APP})$$

$$\text{case } K_k^\nu \overline{A_j} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow M_i} \rightarrow_\delta M_k[\overline{A_j}/\overline{x_{kj}}] \quad (\rightarrow_\delta\text{-CASE})$$

$$n^\nu + M \rightarrow_\delta \text{add}_n M \quad (\rightarrow_\delta\text{-PRIMOP-L})$$

$$\text{add}_{n_1} n_2^\nu \rightarrow_\delta n_3^\perp \quad \text{if } n_3 = n_1 + n_2 \quad (\rightarrow_\delta\text{-PRIMOP-R})$$

$$\text{if0 } n^\nu \text{ then } M_1 \text{ else } M_2 \rightarrow_\delta M_i \quad \text{if } i = (n = 0) ? 1 : 2 \quad (\rightarrow_\delta\text{-IF0})$$

$$\text{seq } V^\nu M \rightarrow_\delta M \quad (\rightarrow_\delta\text{-SEQ})$$

$$\boxed{use(S) = \nu}$$

$$use(\#^{\delta, \nu} x, S) = \nu \oplus use(S)$$

$$use(R, S) = use(R)$$

$$use(\varepsilon) = 0$$

$$\boxed{use(R) = \nu}$$

$$use([\cdot] A) = 1$$

$$use(\text{case } [\cdot] \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow M_i}) = 1$$

$$use([\cdot] + M) = 1$$

$$use(\text{add}_n [\cdot]) = 1$$

$$use(\text{if0 } [\cdot] \text{ then } M_1 \text{ else } M_2) = 1$$

$$use(\text{seq } [\cdot] M) = 0$$

Either of these subtractions may fail; if the demand or use annotations δ, ν do not permit the demand/use attempted then the rule may not fire, and the configuration is left in a *BadDemand* or *BadUsage* state respectively.

This is why we do not need to remove used-once bindings after they have been used; instead they remain, but have $\delta = 0$ and are therefore unusable.

- Rule (\rightarrow -UPDATE) always fires with a value in the control, and all values have a usage annotation. The update frame records the usage that will be permitted by the new binding. This usage is subtracted from the permitted usage of the value in the control, leaving the remaining usage. The demand of the new binding is simply taken from the update frame.

Again, the subtraction may fail, leaving the configuration in a *BadUsage* state.

- Rule (\rightarrow -REDUCE) performs a basic computation by using the primitive transition relation \rightarrow_δ . The relevant usage checking is performed by (\rightarrow -REDUCE) rather than by the \rightarrow_δ rules.

The primitive transition will make usage $use(R)$ of the value, and consume it. Thus the value must permit this usage. This requires $\nu \leq use(R)$ rather than $\nu \ominus use(R) \neq \top$, because there will be no further uses (we could write instead $(\nu \ominus use(R)) \leq 0$). If this test fails, the configuration is left in a *BadUsage* state to indicate an incorrect usage annotation.

There is a further termination condition:

- If the configuration terminates in a state $\langle \overline{x_i = \delta_i, \nu_i} M_i; V^\nu; \varepsilon \rangle$, then we must have $\nu \leq 0$ and for all i , $\delta_i \leq 0$ and $\nu_i \leq 0$; if this holds we are in *Value*, otherwise we are in *BadDemand* or *BadUsage* as appropriate.

C.3.4 Example of behaviour

We now work through an example evaluation, demonstrating the way in which exact uses and demands of each thunk are tracked. The example is not short, but we work through it in some detail in order to demonstrate clearly how the model works, and to convince the reader of its intuitive correctness.

We take a program based on the example of Section C.1 but using a pair type defined by data $P \alpha = \text{MkP } \alpha \alpha$. We write 2 for the demand/use annotation $1 \oplus 1$.

```

letrec a =1,1 e1
      b =1,1 e2
      q =2,2 (MkP a b)2
in
letrec fst =1,1  $\lambda^\perp p . \text{case } p \text{ of MkP } x y \rightarrow x$ 
      snd =1,1  $\lambda^\perp p . \text{case } p \text{ of MkP } x y \rightarrow y$ 
in
fst q + snd q

```

The annotations require that a and b are each demanded and used exactly once, that q is demanded and used exactly twice, and that fst and snd may be demanded

and/or used any number of times. We abbreviate the right-hand side of fst , $\lambda^\perp p$. case p of MkP $x y \rightarrow x$, by F , and $\lambda^\perp p$. case p of MkP $x y \rightarrow y$ by S .

We begin after the five bindings are placed on the heap by (\rightarrow -LETREC).

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, fst = \perp, \perp \ F, snd = \perp, \perp \ S; \\ fst \ q + snd \ q; \\ \varepsilon \rangle$$

$$\rightarrow^2 \quad \{ (\rightarrow\text{-UNWIND}), (\rightarrow\text{-UNWIND}) \}$$

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, fst = \perp, \perp \ F, snd = \perp, \perp \ S; \\ fst; \\ [\cdot] \ q, [\cdot] + snd \ q \rangle$$

$$\rightarrow \quad \{ (\rightarrow\text{-LOOKUP}); use(S) = \mathbb{1}, \perp \ominus \mathbb{1} = \perp, \perp \ominus \mathbb{1} = \perp. \}$$

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, snd = \perp, \perp \ S; \\ \lambda^\perp p \text{ . case } p \text{ of MkP } x \ y \rightarrow x; \\ \#^{\perp, \perp} fst, [\cdot] \ q, [\cdot] + snd \ q \rangle$$

$$\rightarrow \quad \{ (\rightarrow\text{-UPDATE}); \perp \ominus \perp = \perp \}$$

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, fst = \perp, \perp \ F, snd = \perp, \perp \ S; \\ \lambda^\perp p \text{ . case } p \text{ of MkP } x \ y \rightarrow x; \\ [\cdot] \ q, [\cdot] + snd \ q \rangle$$

Since F is already a value, the lookup is immediately followed by a matching update. The demand made by this is of course exactly one; the use is also one in this case because on top of the stack is an application frame, and this makes one use of the function in the control. Subtracting one demand and one use from \perp, \perp leaves the annotations untouched, both on the binding and on the abstraction value.

$$\rightarrow \quad \{ (\rightarrow\text{-REDUCE}) \text{ by } (\rightarrow_\delta\text{-APP}); use([\cdot] \ q) = \mathbb{1} \text{ and } \perp \leq \mathbb{1} \}$$

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, fst = \perp, \perp \ F, snd = \perp, \perp \ S; \\ \text{case } q \text{ of MkP } x \ y \rightarrow x; \\ [\cdot] + snd \ q \rangle$$

The application can go ahead because the application frame $[\cdot] \ q$ on the stack makes exactly one use of the function in the control, and the function permits use \perp , which strictly permits one usage.

$$\rightarrow \quad \{ (\rightarrow\text{-UNWIND}) \}$$

$$\langle a = \mathbb{1}, \mathbb{1} \ e_1, b = \mathbb{1}, \mathbb{1} \ e_2, q = \mathbb{2}, \mathbb{2} \ (\text{MkP } a \ b)^2, fst = \perp, \perp \ F, snd = \perp, \perp \ S; \\ q; \\ \text{case } [\cdot] \text{ of MkP } x \ y \rightarrow x, [\cdot] + snd \ q \rangle$$

$$\begin{array}{l}
\longrightarrow \quad \{ (\multimap\text{-LOOKUP}); \text{use}(S) = 1, 2 \ominus 1 = 1, 2 \ominus 1 = 1. \} \\
\langle a =^{\mathbb{1}, \mathbb{1}} e_1, b =^{\mathbb{1}, \mathbb{1}} e_2, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
(\text{MkP } a \ b)^2; \\
\#^{\mathbb{1}, \mathbb{1}} q, \text{case } [\cdot] \text{ of MkP } x \ y \rightarrow x, [\cdot] + snd \ q \rangle
\end{array}$$

The lookup of q removes one demand and one use; this time the use will be made by a case frame. Notice *all* the permitted uses are left on the expression in the control until it is reduced to a value; once this is done ($\multimap\text{-UPDATE}$) will remove those uses that are to remain on the binding.

$$\begin{array}{l}
\longrightarrow \quad \{ (\multimap\text{-UPDATE}); 2 \ominus 1 = 1. \} \\
\langle a =^{\mathbb{1}, \mathbb{1}} e_1, b =^{\mathbb{1}, \mathbb{1}} e_2, q =^{\mathbb{1}, \mathbb{1}} (\text{MkP } a \ b)^1, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
(\text{MkP } a \ b)^1; \\
\text{case } [\cdot] \text{ of MkP } x \ y \rightarrow x, [\cdot] + snd \ q \rangle
\end{array}$$

In this case, the expression was already a value. We are now left with just one use on the value in the control, exactly that required by the stack. The remaining use is on the value in the binding. Notice that for this to work, we require that $\forall x, y. x \ominus (x \ominus y) \leq y$.

$$\begin{array}{l}
\longrightarrow \quad \{ (\multimap\text{-REDUCE}) \text{ by } (\multimap_\delta\text{-CASE}); \text{use}(\text{case } [\cdot] \text{ of MkP } x \ y \rightarrow x) = 1 \text{ and } 1 \leq 1 \} \\
\langle a =^{\mathbb{1}, \mathbb{1}} e_1, b =^{\mathbb{1}, \mathbb{1}} e_2, q =^{\mathbb{1}, \mathbb{1}} (\text{MkP } a \ b)^1, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
a; \\
[\cdot] + snd \ q \rangle
\end{array}$$

$$\begin{array}{l}
\longrightarrow \quad \{ (\multimap\text{-LOOKUP}); \text{use}(S) = 1, 1 \ominus 1 = 0, 1 \ominus 1 = 0. \} \\
\langle b =^{\mathbb{1}, \mathbb{1}} e_2, q =^{\mathbb{1}, \mathbb{1}} (\text{MkP } a \ b)^1, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
e_1; \\
\#^{0, 0} a, [\cdot] + snd \ q \rangle
\end{array}$$

$$\begin{array}{l}
\longrightarrow^* \quad \{ \text{evaluate } e_1, \text{yielding } n_1 \} \\
\langle b =^{\mathbb{1}, \mathbb{1}} e_2, q =^{\mathbb{1}, \mathbb{1}} (\text{MkP } a \ b)^1, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
n_1^1; \\
\#^{0, 0} a, [\cdot] + snd \ q \rangle
\end{array}$$

$$\begin{array}{l}
\longrightarrow \quad \{ (\multimap\text{-UPDATE}); 1 \ominus 0 = 1. \} \\
\langle a =^{0, 0} n_1^0, b =^{\mathbb{1}, \mathbb{1}} e_2, q =^{\mathbb{1}, \mathbb{1}} (\text{MkP } a \ b)^1, fst =^{\perp, \perp} F, snd =^{\perp, \perp} S; \\
n_1^1; \\
[\cdot] + snd \ q \rangle
\end{array}$$

This lookup and update uses a up completely, leaving no permitted demands or uses. Any further attempts will get stuck, indicating an incorrect annotation.

$$\begin{array}{l}
\multimap^* \quad \{ \text{and do the same for } b \} \\
\langle a =^{0,0} n_1^0, b =^{0,0} n_2^0, q =^{0,0} (\text{MkP } a \ b)^0, fst =^{\perp,\perp} F, snd =^{\perp,\perp} S; \\
n_2^{\perp}; \\
\text{add}_{n_1} [\cdot] \rangle \\
\multimap \quad \{ (\multimap\text{-REDUCE}) \text{ by } (\multimap_{\delta}\text{-PRIMOP-R}); use(\text{add}_{n_1}) = \mathbb{1} \text{ and } \mathbb{1} \leq \mathbb{1} \} \\
\langle a =^{0,0} n_1^0, b =^{0,0} n_2^0, q =^{0,0} (\text{MkP } a \ b)^0, fst =^{\perp,\perp} F, snd =^{\perp,\perp} S; \\
n_3^{\perp}; \\
\varepsilon \rangle \\
\Box \quad \{ \text{In Value, because } \perp \leq 0 \text{ and } \delta_i, \nu_i \leq 0. \}
\end{array}$$

During execution, we verify that none of the annotations are too small: if we attempt to demand a thunk that has already had all its demands used up, it is an error. Once the program has terminated, however, we must still verify that none of the annotations were too *large*: the value in the control must permit zero use, and all the thunk annotations must permit both zero demand and zero use. This is indeed the case here, and so the annotations on the original program were correct.

C.4 A type system

We now wish to design a type system that is sound with respect to this operational semantics, and captures as much of it as possible. We design a monomorphic type system for now, leaving the extension to simple polymorphism (or otherwise) to future work.

C.4.1 The type language

The new term and type languages are presented in Figure C.5.⁷ There are few changes to the term language. Lambda abstractions are decorated with the type of their argument. letrec bindings are decorated with their types; the demand and use of the binding are simply the topmost annotations from this type and so are omitted. For algebraic data types, constructors take vectors of usage arguments $\overline{\nu}_l$ and type arguments $\overline{\alpha}_k$ as well as their component value arguments, and case expressions are annotated with the type of their scrutinee; algebraic data types are discussed further in Section C.4.4.

The type language is more interesting. We have three kinds of type:

- τ -types are unannotated;
- σ -types have a usage annotation ν , and are used to type expressions and values, which are simply *used*; and

⁷As we do not intend to give an inference algorithm or polymorphic type system here, we omit the former distinction (Section 3.2.2) between operational-semantic- and type- annotations. For present purposes, the two are identical, and are both denoted by δ for demands and ν for uses.

Figure C.5 The typed language EL_1 .

Terms	$e ::=$ a $ $ n^ν $ $ $K_i^\nu \overline{v_l} \overline{\tau_k} \overline{a_k}$ $ $ $\lambda^\nu x : \rho . e$ $ $ $e a$ $ $ $\text{case } e : T \overline{v_l} \overline{\tau_k} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i}$ $ $ $e_1 + e_2$ $ $ $\text{add}_n e$ $ $ $\text{if0 } e \text{ then } e_1 \text{ else } e_2$ $ $ $\text{seq } e_1 e_2$ $ $ $\text{letrec } \overline{x_i : \rho_i = e_i} \text{ in } e$	atom literal (integer) constructor term abstraction term application case expression primop (addition) partially-saturated primop conditional strict sequencing recursive let binding
Atoms	$a ::=$ x	term variable
τ -types	$\tau ::=$ $\rho \rightarrow \sigma$ $ $ Int $ $ $T \overline{v_l} \overline{\tau_k}$	function type primitive type (integer) algebraic data type
σ -types	$\sigma ::=$ τ^ν	value type
ρ -types	$\rho ::=$ σ^δ (also written $\tau^{\delta, \nu}$)	thunk type
Demands	$\delta \in \mathcal{D}$	permitted demand
Uses	$\nu \in \mathcal{D}$	permitted use
Decls	$T : \text{data } T \overline{u_l} \overline{\alpha_k} = \overline{K_i \overline{\varrho_{ij}}}$	data type declaration

The restricted thunk type ϱ is defined in the text.

- ρ -types have both a usage annotation ν and a demand annotation δ , and are used to type thunks and variables, which are both *used* and *demanded*.

The kind of the arrow constructor, namely $\rho \rightarrow \sigma$, may be surprising. Since we work in A-normal form (Section 2.2), a function argument is always a variable. Thus it is meaningful to talk about the number of times the argument is demanded, and to record it in the type. The *result* of a function application however is a value, and demand is irrelevant for values.

Demands and uses are both taken from the same domain \mathcal{D} . Algebraic data type demand/use parameters are denoted ν by convention. To simplify their interpretation, the component types ϱ_{ij} in a data type declaration are restricted: usage variables u_l may occur only as topmost annotations or usage arguments to embedded data type constructors; they may not annotate arguments or results of arrow types, nor may they appear inside type arguments of embedded data type constructors. Section C.4.4 elaborates.

C.4.2 Lifting the annotation operations to types and environments

For use in the well-typing rules, we lift some of the annotation operations of Figure C.2 to types and then to environments. The liftings to ρ -types are as follows:

$$\begin{aligned}
 (\rho \rightarrow \sigma)^{\delta, \nu} \oplus (\rho \rightarrow \sigma)^{\delta', \nu'} &= (\rho \rightarrow \sigma)^{\delta \oplus \delta', \nu \oplus \nu'} & \text{Int}^{\delta, \nu} \oplus \text{Int}^{\delta', \nu'} &= \text{Int}^{\delta \oplus \delta', \nu \oplus \nu'} \\
 (\rho \rightarrow \sigma)^{\delta, \nu} \wedge (\rho \rightarrow \sigma)^{\delta', \nu'} &= (\rho \rightarrow \sigma)^{\delta \wedge \delta', \nu \wedge \nu'} & \text{Int}^{\delta, \nu} \wedge \text{Int}^{\delta', \nu'} &= \text{Int}^{\delta \wedge \delta', \nu \wedge \nu'} \\
 \nu \cdot (\rho \rightarrow \sigma)^{\delta', \nu'} &= (\rho \rightarrow \sigma)^{\nu \cdot \delta', \nu \cdot \nu'} & \nu \cdot \text{Int}^{\delta', \nu'} &= \text{Int}^{\nu \cdot \delta', \nu \cdot \nu'} \\
 \nu \triangleright (\rho \rightarrow \sigma)^{\delta', \nu'} &= (\rho \rightarrow \sigma)^{\nu \triangleright \delta', \nu \triangleright \nu'} & \nu \triangleright \text{Int}^{\delta', \nu'} &= \text{Int}^{\nu \triangleright \delta', \nu \triangleright \nu'}
 \end{aligned}$$

$$\begin{aligned}
 (T \overline{\nu_l} \overline{\tau_k})^{\delta, \nu} \oplus (T \overline{\nu'_l} \overline{\tau_k})^{\delta', \nu'} &= (T \overline{\nu_l \oplus \nu'_l} \overline{\tau_k})^{\delta \oplus \delta', \nu \oplus \nu'} \\
 (T \overline{\nu_l} \overline{\tau_k})^{\delta, \nu} \wedge (T \overline{\nu'_l} \overline{\tau_k})^{\delta', \nu'} &= (T \overline{\nu_l \wedge \nu'_l} \overline{\tau_k})^{\delta \wedge \delta', \nu \wedge \nu'} \\
 \nu \cdot (T \overline{\nu'_l} \overline{\tau_k})^{\delta', \nu'} &= (T \overline{\nu \cdot \nu'_l} \overline{\tau_k})^{\nu \cdot \delta', \nu \cdot \nu'} \\
 \nu \triangleright (T \overline{\nu'_l} \overline{\tau_k})^{\delta', \nu'} &= (T \overline{\nu \triangleright \nu'_l} \overline{\tau_k})^{\nu \triangleright \delta', \nu \triangleright \nu'}
 \end{aligned}$$

The operations are lifted by applying them pointwise to topmost annotations and usage arguments of type constructors, but *not* to functions or type arguments of type constructors.⁸ Usage arguments of type constructors must be included to count uses of individual components, as in the example from Section C.3.4. Section C.4.4 discusses the restrictions to data type declarations that allow us to define \oplus *etc.* so simply on type constructors. Liftings to σ -types simply omit the topmost δ and δ' annotations.

We further lift these operations to environments: to compute $\Gamma_1 \oplus \Gamma_2$, where $\Gamma_1 = (x_1 : \rho_1, x_2 : \rho_2, \dots)$ and $\Gamma_2 = (x'_1 : \rho'_1, x'_2 : \rho'_2, \dots)$, we add to Γ_1 pairs of the form $x'_i : \tau_i'^{0,0}$ for each x'_i not already appearing in Γ_1 , and similarly for Γ_2 . We then

⁸Possibly \wedge could be redefined to descend inside function types, using \wedge at positive positions and its dual \vee at negative positions.

perform \oplus pointwise on each variable. For example,

$$\begin{aligned}
 & (x : \text{Int}^{\delta_x, \nu_x}, y : \text{Int}^{\delta_y, \nu_y}) \oplus (x : \text{Int}^{\delta'_x, \nu'_x}, z : \text{Int}^{\delta_z, \nu_z}) \\
 &= (x : (\text{Int}^{\delta_x, \nu_x} \oplus \text{Int}^{\delta'_x, \nu'_x}), y : (\text{Int}^{\delta_y, \nu_y} \oplus \text{Int}^{0,0}), z : (\text{Int}^{0,0} \oplus \text{Int}^{\delta_z, \nu_z})) \\
 &= (x : \text{Int}^{(\delta_x \oplus \delta'_x), (\nu_x \oplus \nu'_x)}, y : \text{Int}^{(\delta_y \oplus 0), (\nu_y \oplus 0)}, z : \text{Int}^{(0 \oplus \delta_z), (0 \oplus \nu_z)}) \\
 &= (x : \text{Int}^{(\delta_x \oplus \delta'_x), (\nu_x \oplus \nu'_x)}, y : \text{Int}^{\delta_y, \nu_y}, z : \text{Int}^{\delta_z, \nu_z})
 \end{aligned}$$

We do the same for $\Gamma_1 \wedge \Gamma_2$; missing variables become conjunctions with zero use and demand. $\nu \cdot \Gamma$ and $\nu \triangleright \Gamma$ are simply computed pointwise on the variable types.

C.4.3 Well-typing rules

The well-typing rules for the language are given in Figure C.6. In the judgement form $\Gamma \vdash_E e : \sigma$, the variables in the environment have ρ -types, since they denote thunks, and the expression has a σ -type, since it is being evaluated to a value. As before, environments are unordered; in addition, variables that are neither demanded nor used may be arbitrarily added or removed. Thus $(\Gamma_1, \Gamma_2) = (\Gamma_2, \Gamma_1, x : \text{Int}^{0,0})$, if $x \notin \text{dom}(\Gamma_1, \Gamma_2)$.

Environments are treated linearly (as suggested in Section 3.9.7): the types in the environment describe the exact uses and demands made in evaluating the expression to weak head normal form; where two environments must be combined (as in $(\vdash_E\text{-PRIMOP})$), the demands and uses are added together.⁹ This replaces the previous method of counting demands by searching for syntactic occurrences in the terms with a principled, logical approach.

We now consider each rule in turn.

Using a variable demands it exactly once from the environment, and we encode this in rule $(\vdash_E\text{-VAR})$. Using a literal requires nothing from the environment, and the resulting value may be used as many times as specified in the annotation.

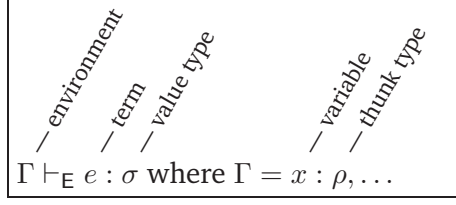
The conditional *if0* e then e_1 else e_2 , rule $(\vdash_E\text{-IF0})$, uses the value of the condition exactly once, and then returns the value either of e_1 or e_2 . The condition is always computed, and so the environment required for this is always required by the conditional; but only one branch is computed, and so we compute the and of the two environments, yielding an environment that permits both uses.

Evaluating a primitive operation demands each argument exactly once, and yields a result that may be used arbitrarily as indicated by \perp .

Rule $(\vdash_E\text{-ABS})$ records the demands and uses ρ made of the argument x while evaluating the body e , placing this information in the type and decorating the binder. Demands and uses made of other variables during evaluation will be repeated every time the function is used, and so the product operator \cdot is used to multiply the demands and uses in Γ by the use of the abstraction. Note specifically that if the function is never used and $\nu = 0$, the demands and uses in Γ will never occur.

Rule $(\vdash_E\text{-APP})$ depends on the A-normal form presentation; an argument to a function is always a variable, and its demand is taken from the argument type ρ of

⁹That is, in the *parlance* of linear logic, such operators are “multiplicative”.

Figure C.6 Well-typing rules for EL_1 (cf. Figure 3.2.

$$\begin{array}{c}
\frac{}{x : \tau^{\mathbb{1}, \nu} \vdash_E x : \tau^\nu} (\vdash_E\text{-VAR}) \quad \frac{}{\emptyset \vdash_E n^\nu : \text{Int}^\nu} (\vdash_E\text{-LIT}) \\
\\
\frac{\Gamma \vdash_E e : \text{Int}^{\mathbb{1}} \quad \Gamma_1 \vdash_E e_1 : \sigma \quad \Gamma_2 \vdash_E e_2 : \sigma}{\Gamma \oplus (\Gamma_1 \wedge \Gamma_2) \vdash_E \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma} (\vdash_E\text{-IF0}) \\
\\
\frac{\Gamma_1 \vdash_E e_1 : \text{Int}^{\mathbb{1}} \quad \Gamma_2 \vdash_E e_2 : \text{Int}^{\mathbb{1}}}{\Gamma_1 \oplus \Gamma_2 \vdash_E e_1 + e_2 : \text{Int}^\perp} (\vdash_E\text{-PRIMOP}) \quad \frac{\Gamma \vdash_E e : \text{Int}^{\mathbb{1}}}{\Gamma \vdash_E \text{add}_n e : \text{Int}^\perp} (\vdash_E\text{-PRIMOP-R}) \\
\\
\frac{\Gamma, x : \rho \vdash_E e : \sigma}{\nu \cdot \Gamma \vdash_E \lambda^\nu x : \rho . e : (\rho \rightarrow \sigma)^\nu} (\vdash_E\text{-ABS}) \quad \frac{\Gamma \vdash_E e : (\rho \rightarrow \sigma)^{\mathbb{1}}}{\Gamma, x : \rho \vdash_E e x : \sigma} (\vdash_E\text{-APP}) \\
\\
\frac{\begin{array}{l} \Gamma_i, x_j : \sigma_{ij}^{\delta_{ij}} \vdash_E e_i : \sigma_i \quad \text{for all } i \\ \Gamma_0, x_j : \sigma_{0j}^{\delta_{0j}} \vdash_E e : \sigma \\ \sigma_i^{\delta_i} = \sigma_{0i}^{\delta_{0i}} \oplus (\bigoplus_j \delta_j \triangleright \sigma_{ji}^{\delta_{ji}}) \quad \text{for all } i \end{array}}{\Gamma_0 \oplus (\bigoplus_j (\delta_j \triangleright \Gamma_j)) \vdash_E \text{letrec } x_i : \sigma_i^{\delta_i} = e_i \text{ in } e : \sigma} (\vdash_E\text{-LETREC}) \\
\\
\frac{\Gamma_1 \vdash_E e_1 : \tau_1^0 \quad \Gamma_2 \vdash_E e_2 : \sigma_2}{\Gamma_1 \oplus \Gamma_2 \vdash_E \text{seq } e_1 e_2 : \sigma_2} (\vdash_E\text{-SEQ}) \\
\\
\frac{\Gamma_1 \vdash_E e : \sigma}{\Gamma_1 \wedge \Gamma_2 \vdash_E e : \sigma} (\vdash_E\text{-SUB-L}) \quad \frac{\Gamma \vdash_E e : \sigma_1 \wedge \sigma_2}{\Gamma \vdash_E e : \sigma_1} (\vdash_E\text{-SUB-R}) \\
\\
\frac{\rho_{ij}^\circ = \rho_{ij}[\overline{v_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } j}{\bigoplus_j x_j : \rho_{ij}^\circ \vdash_E K_i^\nu \overline{v_l} \overline{\tau_k} x_j : (T \overline{v_l} \overline{\tau_k})^\nu} (\vdash_E\text{-CON}) \\
\\
\frac{\begin{array}{l} \Gamma_0 \vdash_E e : (T \overline{v_l} \overline{\tau_k})^{\mathbb{1}} \\ \rho_{ij}^\circ = \rho_{ij}[\overline{v_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } i, j \\ \Gamma_i, x_{ij} : \rho_{ij}^\circ \vdash_E e_i : \sigma \quad \text{all } i \end{array}}{\Gamma_0 \oplus (\wedge_i \Gamma_i) \vdash_E \text{case } e : T \overline{v_l} \overline{\tau_k} \text{ of } K_i \overline{x_{ij}} \rightarrow e_i : \sigma} (\vdash_E\text{-CASE})
\end{array}$$

the function.¹⁰ The more obvious rule

$$\frac{\Gamma_1 \vdash_E e : (\sigma_1^{\nu_1} \rightarrow \sigma_2)^{\mathbb{1}} \quad \Gamma_2 \vdash_E x : \sigma_1}{\Gamma_1 \oplus \Gamma_2 \vdash_E e x : \sigma_2} (\vdash_E\text{-APP-WRONG})$$

does not work because it always demands x exactly once, whatever the value of ν_1 .

Sequencing evaluates but does not use the first argument, and then evaluates the second; the rule is obvious.

There are two forms of subsumption. If e may be evaluated to type σ while making use of environment Γ_1 , then the same expression may be evaluated to the same type while making use of environment $\Gamma_1 \wedge \Gamma_2$, a more general environment which both permits use Γ_1 and permits use Γ_2 . This is rule ($\vdash_E\text{-SUB-L}$). Further, if e may be evaluated to type $\sigma_1 \wedge \sigma_2$ while making use of environment Γ , then e may be evaluated to type σ_1 , a more specific type. This is rule ($\vdash_E\text{-SUB-R}$).

Rule ($\vdash_E\text{-SUB-L}$) is required in order to preserve subject reduction. Observe that (in the $\mathcal{P}(\mathbb{N})$ model):

$$y : \tau^{\{0,1\},\{0,1\}}, z : \tau^{\{0,1\},\{0,1\}} \vdash_E \text{if0 } 0 \text{ then } y \text{ else } z : \tau^{\{1\},\{1\}}$$

reduces to y , and ($\vdash_E\text{-VAR}$) states that

$$y : \tau^{\{1\},\{1\}} \vdash_E y : \tau^{\{1\}}$$

In order to type y in the same context, we must use ($\vdash_E\text{-SUB-L}$), \wedge -ing the context $z : \tau^{\{1\},\{1\}}$ to give

$$y : \tau^{\{0,1\},\{0,1\}}, z : \tau^{\{0,1\},\{0,1\}} \vdash_E y : \tau^{\{1\},\{1\}}$$

as expected.

Recursive letrec bindings are handled by the ($\vdash_E\text{-LETREC}$) rule. For each binding i , the demands and uses it makes of the environment and of each other binding in the recursive group are recorded in Γ_i and $\sigma_{ij}^{\delta_{ij}}$ respectively. Similar information is recorded for the body in Γ_0 and $\sigma_{0j}^{\delta_{0j}}$. The results are then combined to yield an overall environment demand/use and the overall demands/uses of each binding. This combination makes critical use of the guard operator \triangleright in order to model the laziness of the bindings: only if a binding j is actually demanded are its environment demands Γ_j and binding demands and uses $\sigma_{ji}^{\delta_{ji}}$ included in the combination. The equation is recursive because the expression is recursive; in the non-recursive case all the $\sigma_{ij}^{\delta_{ij}}$ are $\tau^{0,0}$ and thus for all i , $\sigma_i^{\delta_i} = \sigma_{0i}^{\delta_{0i}}$; however, the environments are still guarded by their demands.

These rules are very similar to those of $FLIX_1$, replacing 1 with $\mathbb{1}$ and ω with \perp . In ($\vdash_E\text{-ABS}$), the use of \cdot has the same effect as the $|\Gamma(y)| \leq \kappa$ constraint in ($\vdash_1\text{-ABS}$): if the abstraction is used 1, the constraint has no effect (just as $\mathbb{1} \cdot \nu = \nu$); if the abstraction is used ω , the constraint forces all annotations in the environment to ω (just as $\perp \cdot \nu = \perp$). The side condition in ($\vdash_E\text{-LETREC}$) has already been explained in footnote 9, Section 3.3.6. The subtyping relation in ($\vdash_1\text{-SUB}$) has been subsumed by conjunction \wedge , except that this does not descend into the type (perhaps it should; adding this would not be difficult for \wedge , although its meaning for \oplus is less clear!). The remaining rules are identical.

¹⁰Even though the argument's type is taken directly from Γ , subsumption may still be applied. One must simply apply ($\vdash_E\text{-SUB-L}$) below it, rather than ($\vdash_E\text{-SUB-R}$) above it.

C.4.4 Data structures

For data structures we use essentially the same scheme as Chapter 5. There are some crucial differences, however.

Firstly, in Chapter 5 every occurrence of a variable is at the same type; specifically, if a data structure is bound then every use of that data structure is at the same type. This is no longer the case in the system of the present appendix; instead, the types of each occurrence are added together to determine the type at the binding site. This means, *inter alia*, that in $(\vdash_E\text{-CASE})$ the topmost annotation of the scrutinee carries no useful information; it is simply always $\mathbb{1}$. Thus we cannot easily pass the binding site's topmost annotation to recursive instances as we did before.

We resolve this by ignoring the topmost annotation altogether; instead we must use the scheme rejected in Section 5.4.5.1, using an extra usage argument for the usage of the recursive instance.

The annotations on the types of the components of a datum when it is constructed refer to the *total use and demand* of those components, not the use and demand per use of the datum as in some other systems. Consequently, there is no need for a sharing constraint relating the use of the datum with the use of the components, as was required in Section 5.3.2. The quasi-linear type system of Kobayashi [Kob99, §3.1] has the same property, as does the usage type system of Mogensen [Mog97b].

Secondly, we desire that the example of Section C.1 works correctly: the sum of a use of the left component of a pair and a use of the right component should be one use of each, and two uses of the pair itself. This means we must descend inside algebraic data types when computing \oplus . For algebraic data types that are purely sums of products, possibly with more such algebraic data types embedded and recursion, we have the property that each usage argument to the type constructor annotates a component (or components) of a data structure. Thus for such data types we may simply add the arguments pointwise. Functions and non-regular recursion would violate this story, and so pending further work we restrict the annotations of such types in data type declarations to usage/demand *constants*, not variables.

C.4.5 Weakening and contraction

Rule $(\vdash_E\text{-SUB-L})$ is a form of weakening, but it is not quite the same as the weakening of intuitionistic logic. In fact, the explicit counting in the environment makes both weakening and contraction appear rather different.

- *Contraction* in intuitionistic logic is normally present to permit multiple references to a single variable; in the present system multiple references are combined, and the uses added by the operator \oplus .
- *Weakening* in intuitionistic logic is normally present to permit *not* referring to a variable; in the present system this is an equivalence on environments, where no mention is equivalent to a mention with demand and use $\mathbb{0}$.

The need for weakening, at least, arises from *uncertainties* in the number of uses of a variable. This is catered for in the present system by keeping *sets* of annotations, rather than single annotations. Thus a notion of weakening that claims 3 uses may

be weakened to 4 uses makes no sense; instead, we say 3 uses may be weakened to 3 or 4 uses.

Contraction seems to have different character; it is merely about multiple uses, and involves no uncertainty. In the present system, 3 uses and 1 use may be ‘contracted’ to 4 uses, and this happens automatically inside the \oplus operator, used in the appropriate places. In a linear system, though, contraction *does* introduce uncertainty: one cannot determine whether the variable was actually used just once, or more than once.

C.4.6 Sample typing

As an example, here is a well-typed use of the factorial function, assuming subtraction, multiplication, and case on integers with the obvious semantics and $\mathcal{P}(\mathbb{N})$ annotations:

$$\begin{aligned}
 \emptyset \vdash_{\mathbb{E}} \text{letrec } f & : (\text{Int}^{\{1,3\},\{1,3\}} \rightarrow \text{Int}^{\{1\}})^{\delta_f, \nu_f} \\
 & = \lambda^{\{1,3\}} n : \text{Int}^{\{1,3\},\{1,3\}} . \text{case } n \text{ of} \\
 & \quad 0 \rightarrow 1^{\{1\}} \\
 & \quad - \rightarrow \text{letrec } n' : \text{Int}^{\{1,3\},\{1,3\}} = n - 1^{\{1\}} \\
 & \quad \quad \text{in } n * f n' \\
 \\
 ten & : \text{Int}^{\{1,3\},\{1,3\}} \\
 & = \text{Int}^{\{1,3\}} \\
 \text{in } f \text{ ten} & : 10^{\{1\}}
 \end{aligned}$$

The values of δ_f, ν_f must satisfy the following equation from ($\vdash_{\mathbb{E}}\text{-LETREC}$):

$$\delta_f = \{1\} \oplus (\nu_f \cdot \{0, 1\}) = \nu_f$$

to which the only non- \top solution is $\mathbb{N} \setminus \{0\} = \{1, 2, \dots\}$.

C.4.7 Proof of soundness

A proof of soundness has not been completed, but a sketch including key lemmas appears in Appendix D.6.

C.5 Discussion

The work of extending the analysis of the rest of this thesis to deal with strictness and absence is not yet complete. We have given an operational semantics that we believe captures our intuitive understanding of demand and use (and hence strictness and absence). Crucially, in the development of the operational semantics we discovered that it is necessary to distinguish the notions of *demand* and *use*. Further, we have designed a type system and well-typing rules that appear sufficiently powerful to capture most of the behaviour of the operational semantics, and are conjectured to be sound. In order to implement the analysis fully, an inference algorithm must be

designed also. As we saw in Chapter 4, it is also likely that the type system will need to be extended with some form of polymorphism to obtain good results.

Designing an inference algorithm should not be unduly difficult. The constraints would be significantly complicated by the additional operators, especially product and guard, but the fact that all of the operators used in the well-typing rules are monotonic – there are no negative constraints – should ensure the system is well-behaved. The constraints on annotations in $(\vdash_{\text{E-LETREC}})$ are recursive and would require a fixed-point computation in the inference, but if we assume the use of a finite domain such as the Bierman lattice, simple iteration would suffice to find the solution. It is less clear how fast such an inference algorithm would run.

Simple polymorphism of the sort discussed in Chapter 4 could be added to the type system without difficulty (yielding a language $ELIX_2$), but inferring such types is quite another question. As we have already noted (footnote 15 in Section 4.5.3) moving from the simple two-point lattice to a larger annotation domain would increase the number of cases to be considered by the closure algorithm, and it remains entirely unclear whether the algorithm of Section 4.5.4 could be generalised to $ELIX_2$, or if it could, whether it would yield useful results.

Were these all to be developed, extending an $FLIX_2$ implementation to $ELIX_2$ would be a major undertaking. The language $ELIX_2$ has additional structure within its types, distinguishing ρ as well as σ and τ ; the inference algorithm would have to be entirely new, since environments are manipulated in a different manner; and an entirely new constraint solver would also be required. We therefore leave all this to future work.

C.6 Related work

It is important to note that the usage property discovered by the analysis of this appendix differs from that discovered by the previous analyses. Recall from Section 1.3.4 that Gustavsson and Sands [GS01a] showed that their use-once-don’t-drag property is sufficient to prove work- and space-safety of the inlining transformation, but the weaker use-once property is not. Although the analyses \mathcal{T}_1 and \mathcal{T}_2 have the use-once-don’t-drag property (Section 3.6.1), the analysis of the present appendix has only the use-once property: Gustavsson and Sands’ example letrec $x = \bullet 1 + 2$ in $x + (\lambda y . 1) x$ is well-typed in \vdash_{E} , but not in \vdash_2 .

Sestoft [Ses91, c.5] gives a “usage interval analysis”, a single analysis which subsumes strictness analysis and “sharing” (i.e., usage) analysis. However, his annotations are intervals over the set $\{\text{Zero}, \text{One}, \text{Many}\}$ rather than over the naturals; thus they are equivalent to the Bierman lattice (Section C.3.2) without the middle element $\neq 1$. His analysis is a flow analysis, rather than a type-based analysis.

Mogensen [Mog98] extends the analysis of [TWM95a] to the annotation domain $\{0, 1, \infty\}$ (1 denotes a value used at most once). He considers 0 to be necessary to give a good treatment of data structures. Use and demand are not distinguished; in our system guards occur only in $(\vdash_{\text{E-LETREC}})$, but without a separate demand annotation guards must appear in the rules for constructors also. Recursion is treated much more conservatively. Otherwise the rules are very close to those of Figure C.6.

No semantics or proof of soundness is given.

Our treatment of uses ν in rules (\rightarrow -UPDATE) and (\rightarrow -REDUCE) bears a striking resemblance to the corresponding rules in the operational semantics with update marker check intervals given by Gustavsson [Gus98, §4.2]. Although the two are clearly not the same, this suggests a close relationship that may be worthy of further investigation.

Wright's later analysis [Wri96], also linear-style and for call-by-name, is parameterised by an annotation algebra and may be instantiated as a linearity analysis, a strictness analysis, or one of three flavours of usage analysis (affine, affine-plus-linear, and linear with zero).

Marlow's analysis [Mar93], already described in Section 1.3.5, is an abstract interpretation that tracks the use made by an expression of each of its free variables both during evaluation to WHNF and (separately) during subsequent application/deconstruction. This separation corresponds roughly to our separation of demand and use above, except that the latter is represented merely by a set of closures that are referenced by this object, and does not distinguish, *e.g.*, one application from two (curried) applications.

Strictness analysis has been widely studied, especially using abstract interpretation. The earliest type-based strictness analysis was that of Kuo and Mishra [KM89]. Jensen [Jen91] compares the two approaches and shows that they are equivalent. He discusses a more complex type-based strictness analysis in [Jen98], including polymorphism and conditional types, and considers constraint simplification and solution in this context. The Glasgow Haskell Compiler's old abstract-interpretation based strictness (and absence) analysis is described in [PJP93, PJS98a].

Much other related work has already been noted in Section 1.3.5.

Appendix D.

Proofs

In this appendix we give proofs of theorems and lemmas presented without proof in the main text. Since similar proofs are required for each of the five main languages LIX_0 , LIX_1 , LIX_2 , $FLIX_0$, $FLIX_2$, where appropriate we give a full proof for only the most complicated language; proofs for the others may be obtained by omitting the irrelevant cases.

For reference, the relevant rules *etc.* are all collected in Appendix B.

D.1 Type soundness

We begin by proving type soundness for $FLIX_2$.

We make use of some special notation to abbreviate and clarify the proofs, as follows. The symbols Γ and H denote respectively $\overline{x_i : \sigma_i}$ and $\overline{x_i : \sigma_i =^{X_i} e_i}$; that is, Γ is the type environment arising from the heap H , whose variables are $\overline{x_i}$. Similarly for $\Gamma', H', \overline{x'_i}, \overline{\sigma'_i}, \overline{\chi'_i}, \overline{e'_i}$ and $\Gamma'', H'', \overline{x''_i}, \overline{\sigma''_i}, \overline{\chi''_i}, \overline{e''_i}$. Indices start from 1, so “for all i ” does not include $i = 0$; we often use e_0 to denote the control of a configuration. We write V for values, rather than v as elsewhere. We often write “by I.H.” to abbreviate “by the inductive hypothesis”.

In this section (D.1), $\Gamma \vdash_2 e : \sigma$ denotes a *slightly weaker typing judgement* than in Figure B.6 and the main text: we allow any update flag to be $!$, even if it would normally be \bullet . This is necessary to allow Usage Substitution Lemma D.7 to go through. Since the standard (stronger) form implies the weaker form, Theorem D.11 holds as stated under either interpretation.

Our first lemma shows that any type derivation can be converted into a type derivation with exactly one instance of $(\vdash_2\text{-SUB})$ at the bottom (and possibly others higher up), with a non- $(\vdash_2\text{-SUB})$ rule above it.

This allows us to work with general type derivations more easily, since in general there may be zero or more $(\vdash_2\text{-SUB})$ s before we get to the real goods.

Lemma D.1 (Root-normal subsumption)

For all Γ, e, σ , if $\Gamma \vdash_2 e : \sigma$ then there exists a so-called root-normal proof tree, of the form

$$\frac{\displaystyle \frac{\vdots}{\Gamma \vdash_2 e : \sigma'} \text{ (not } (\vdash_2\text{-SUB}))}{\Gamma \vdash_2 e : \sigma} \sigma' \preceq \sigma \text{ (}\vdash_2\text{-SUB)}$$

Proof If $\Gamma \vdash_2 e : \sigma$, then there exists a proof tree with this conclusion. We proceed by induction on the structure of this proof tree. If the bottom rule in the tree is not $(\vdash_2\text{-SUB})$, we may append an instance of $(\vdash_2\text{-SUB})$ at $\sigma' = \sigma$, since \preceq is reflexive, and we are done. If the bottom rule is already $(\vdash_2\text{-SUB})$, with antecedent $\Gamma \vdash_2 e : \sigma''$ where $\sigma'' \preceq \sigma$, remove the bottom rule and by the inductive hypothesis, find a root-normal proof tree for the remainder. Now since \preceq is transitive, $\sigma' \preceq \sigma''$ and $\sigma'' \preceq \sigma$ imply that $\sigma' \preceq \sigma$, and we may replace the two $(\vdash_2\text{-SUB})$ instances with one, and we are done. \square

(In $FLIX_1$, Lemma D.1 is of course a corollary of Lemma D.16).

Our next lemma says that given a well-typed shallow context, the expression in the hole must be well-typed, and can be replaced with another well-typed expression of the same type without affecting the type of the expression as a whole.

Lemma D.2 (Typed hole)

For all Γ, R, e, σ , if $\Gamma \vdash_2 R[e] : \sigma$ then there exists σ' such that $\Gamma \vdash_2 e : \sigma'$ and for all e' , it is the case that $\Gamma \vdash_2 e' : \sigma' \Rightarrow \Gamma \vdash_2 R[e'] : \sigma$.

Proof By Lemma D.1, $\Gamma \vdash_2 R[e] : \sigma$ implies that there is a non- $(\vdash_2\text{-SUB})$ -rooted proof tree for $\Gamma \vdash_2 R[e] : \sigma''$ for some σ'' such that $\sigma'' \preceq \sigma$. We proceed by cases on R .

case $[\cdot] a$

We have $\Gamma \vdash_2 e a : \sigma''$; only one rule other than $(\vdash_2\text{-SUB})$ applies here and hence by $(\vdash_2\text{-APP})$ we have that $\Gamma \vdash_2 e : \sigma'$ where $\sigma' = (\sigma_1 \rightarrow \sigma'')^1$, and $\Gamma \vdash_2 a : \sigma_1$; hence by $(\vdash_2\text{-APP})$ we have $\Gamma \vdash_2 e' a : \sigma''$.

case $[\cdot] \tau$

We have $\Gamma \vdash_2 e \tau : \sigma''$, where $\sigma'' = \tau''^{\kappa''}$; hence by $(\vdash_2\text{-TYAPP})$ we have that $\Gamma \vdash_2 e : \sigma'$ where $\sigma' = (\forall \alpha . \tau')^{\kappa''}$ and $\tau'' = \tau'[\tau/\alpha]$; hence by $(\vdash_2\text{-TYAPP})$ we have $\Gamma \vdash_2 e' \tau : \sigma''$.

case $[\cdot] \kappa$

We have $\Gamma \vdash_2 e \kappa : \sigma''$, where $\sigma'' = \tau''^{\kappa''}$; hence by $(\vdash_2\text{-UAPP})$ we have that $\Gamma \vdash_2 e : \sigma'$ where $\sigma' = (\forall u . \tau')^{\kappa''}$ and $\tau'' = \tau'[\kappa/u]$; hence by $(\vdash_2\text{-UAPP})$ we have $\Gamma \vdash_2 e' \kappa : \sigma''$.

case $[\cdot] : (T \overline{\kappa_l} \overline{\tau_k})^\kappa$ of $\overline{K_i \overline{x_{ij}} \rightarrow e_i}$

We have $\Gamma \vdash_2 \text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i} : \sigma''$; hence by $(\vdash_2\text{-CASE})$ we have that $\sigma' = (T \overline{\kappa_l} \overline{\tau_k})^\kappa$, $\Gamma \vdash_2 e : \sigma'$, and other clauses not involving e ; hence by $(\vdash_2\text{-CASE})$ we have $\Gamma \vdash_2 \text{case } e' : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i} : \sigma''$ as required.

case $[\cdot] + e$

We have $\Gamma \vdash_2 e + e_2 : \sigma''$; hence by (\vdash_2 -PRIMOP) we have that $\sigma'' = \text{Int}^\omega$, $\sigma' = \text{Int}^1$, $\Gamma \vdash_2 e : \sigma'$, $\Gamma \vdash_2 e_2 : \text{Int}^1$; hence by (\vdash_2 -PRIMOP) we have $\Gamma \vdash_2 e' + e_2 : \sigma''$ as required.

case $\text{add}_n [\cdot]$

We have $\Gamma \vdash_2 \text{add}_n e : \sigma''$; hence by (\vdash_2 -PRIMOP-R) we have that $\sigma'' = \text{Int}^\omega$, $\sigma' = \text{Int}^1$, $\Gamma \vdash_2 e : \sigma'$; hence by (\vdash_2 -PRIMOP-R) we have $\Gamma \vdash_2 \text{add}_n e' : \sigma''$ as required.

case $\text{if0} [\cdot] \text{ then } e_1 \text{ else } e_2$

We have $\Gamma \vdash_2 \text{if0 } e \text{ then } e_1 \text{ else } e_2 : \sigma''$; hence by (\vdash_2 -IF0) we have that $\sigma' = \text{Int}^1$, $\Gamma \vdash_2 e : \sigma'$, $\Gamma \vdash_2 e_1 : \sigma''$, $\Gamma \vdash_2 e_2 : \sigma''$; hence by (\vdash_2 -IF0) we have $\Gamma \vdash_2 \text{if0 } e' \text{ then } e_1 \text{ else } e_2 : \sigma''$ as required.

Thus in all cases we have $\Gamma \vdash_2 R[e'] : \sigma''$, and by (\vdash_2 -SUB) we therefore have $\Gamma \vdash_2 R[e'] : \sigma$ as required. \square

The following lemma is useful for establishing the premises required to invoke inductive hypotheses. It is convenient to write it in the form of a rule; whenever (\vdash_2 -WEAK) is invoked, it should be treated as an appeal to Lemma D.3.

Lemma D.3 (Environment weakening)

The following rule is admissible:

$$\frac{\Gamma \vdash_2 e : \sigma \quad \Gamma' \not\leq \Gamma}{\Gamma, \Gamma' \vdash_2 e : \sigma} (\vdash_2\text{-WEAK})$$

Proof By inspection of the well-typing rules. \square

Our next lemma says that the control is well-typed if the configuration is, in a certain environment, and that the environment may be extended and the expression replaced if certain conditions are followed. This allows us to perform each evaluation step of the operational semantics while preserving well-typedness. Recall the definition of trans from Figure B.7, which takes a configuration to an equivalent expression, and the derived typing rule (\vdash_2 -CONFIG) for configurations, which makes use of it. We denote by E a (non-shallow) evaluation context, defined by

$$E ::= [\cdot] \mid R[E]$$

Lemma D.4 (Kitchen sink)

For all S , there exist $k \in \text{Bool}$, H' , E , x' , σ' , e , such that $\text{dom}(H') = \text{dom}(S)$ and for all H , e_0 with $\text{dom}(H) \not\leq \text{dom}(S)$, we have:

(i)

$$\text{trans}\langle H; e_0; S \rangle = \begin{cases} \text{letrec } H, H' \text{ in } E[e_0] & \text{where } H' = \emptyset & \text{if } k = \text{true} \\ (\text{letrec } H, H' \text{ in } e & \text{if } k = \text{false} \\ \text{where } H' \text{ contains } x' : \sigma' =^! E[e_0]) \end{cases}$$

(ii) If $\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H; e_0; S \rangle : \sigma$ then there exists σ_0 such that the following all hold:

- (a) $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i$ for all i ,
- (b) $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_i : \sigma'_i$ for all i ,
- (c) $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma_0$, and
- (d) For all H'', e''_0 with $\text{dom}(H'') \not\subseteq (\text{dom}(H) \cup \text{dom}(S))$, it is the case that if $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_i : \sigma''_i$ for all i and $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_0 : \sigma_0$, then $\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H''; e''_0; S \rangle : \sigma$.

In (i), the two cases correspond to the presence or absence of update frames in S : if k is true, S contains no update frames, and if k is false, S contains one or more update frames. The result $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma_0$ states that in either case, e_0 is typeable in the context provided by the original heap H extended by the pending updates H' . Clause (iid) states that e''_0 may be substituted for e_0 , where e''_0 requires the context provided by an additional heap fragment H'' , provided the configuration is extended with the additional heap fragment.

Proof The proof is by induction on the length of S .

case $S = \varepsilon$

Let $k = \text{true}$, $H' = E = [\cdot]$. (i) follows trivially from the definition of trans. For (ii), we have

$$\begin{aligned}
 & \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H; e_0; \varepsilon \rangle : \sigma \\
 \iff & \{ \text{by } (\vdash_2\text{-CONFIG}) \} \\
 & \overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H \text{ in } e_0 : \sigma \\
 \iff & \{ \text{by Lemma D.1} \} \\
 & \frac{\Gamma, \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \quad \text{all } i \quad \Gamma, \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma''}{\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H \text{ in } e_0 : \sigma''} (\vdash_2\text{-LETREC}) \quad \sigma'' \preceq \sigma \\
 & \frac{\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H \text{ in } e_0 : \sigma''}{\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H \text{ in } e_0 : \sigma} (\vdash_2\text{-SUB}) \\
 \implies & \{ \text{where } \sigma_0 = \sigma \} \\
 & \frac{\Gamma, \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma'' \quad \sigma'' \preceq \sigma}{\Gamma, \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma} (\vdash_2\text{-SUB})
 \end{aligned}$$

Finally, for (iid), for all H'', e''_0 with $\text{dom}(H'') \not\subseteq (\text{dom}(H) \cup \text{dom}(\varepsilon))$ we have:

$$\begin{aligned}
 & \text{from (ii)} \\
 & \frac{\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_i : \sigma''_i \text{ for all } i \quad \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \text{ for all } i \quad \Gamma'' \not\subseteq \Gamma}{\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \text{ for all } i} (\vdash_2\text{-WEAK}) \\
 & \frac{\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_0 : \sigma_0 \quad \Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \text{ for all } i}{\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H, H'' \text{ in } e''_0 : \sigma} (\vdash_2\text{-LETREC}) \\
 & \frac{\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H, H'' \text{ in } e''_0 : \sigma}{\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H''; e''_0; \varepsilon \rangle : \sigma} (\vdash_2\text{-CONFIG})
 \end{aligned}$$

as required.

case $S = \#x : \sigma_x, S'''$

By the inductive hypothesis, for S''' there exists $k''', H''', E''', x''', \sigma''', e'''$. Let $k = \mathbf{false}$, $H' = (H''', x : \sigma_x =^! e_0)$, $E = [\cdot]$, $x' = x$, $\sigma' = \sigma_x$, $e = (k''' ? E'''[x] : e''')$. Then, for all H, e_0 :

(i):

$$\begin{aligned}
& \text{trans}\langle H; e_0; \#x : \sigma_x, S''' \rangle \\
&= \{ \text{by definition of trans} \} \\
& \text{trans}\langle H, x : \sigma_x =^! e_0; x; S''' \rangle \\
&= \{ \text{by I.H.} \} \\
& \begin{cases} \text{letrec } H, x : \sigma_x =^! e_0, H''' \text{ in } E'''[x] & \text{if } k''' = \mathbf{true} \\ \text{letrec } H, x : \sigma_x =^! e_0, H''' \text{ in } e''' & \text{if } k''' = \mathbf{false} \\ \text{where } H''' \text{ contains } x''' : \sigma''' =^! E'''[x] \end{cases} \\
&= \\
& \text{letrec } H, H' \text{ in } e
\end{aligned}$$

as required.

(ii):

$$\begin{aligned}
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H; e_0; \#x : \sigma_x, S''' \rangle : \sigma \\
&\iff \{ \text{by } (\vdash_2\text{-CONFIG}) \text{ and definition of trans} \} \\
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, x : \sigma_x =^! e_0; x; S''' \rangle : \sigma \\
&\implies \{ \text{by I.H.} \} \\
& \begin{aligned} & \Gamma, x : \sigma_x, \Gamma''', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\ & \Gamma, x : \sigma_x, \Gamma''', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma_x \\ & \Gamma, x : \sigma_x, \Gamma''', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i''' : \sigma_i''' \quad \text{for all } i \\ & \Gamma, x : \sigma_x, \Gamma''', \overline{\alpha_l}, \overline{u_m} \vdash_2 x : \sigma_0 \end{aligned} \\
&\implies \{ \text{by definition, and } (\vdash_2\text{-VAR}) \} \\
& \begin{aligned} & \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\ & \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i' : \sigma_i' \quad \text{for all } i \\ & \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma_0 \quad \text{where } \sigma_0 = \sigma_x \end{aligned}
\end{aligned}$$

as required.

(iid): We have by assumption that

$$\begin{aligned}
& \Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i'' : \sigma_i'' \quad \text{for all } i \\
& \Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0'' : \sigma_0 \\
& \Longleftrightarrow \\
& \Gamma, x : \sigma_x, \Gamma''', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i'' : \sigma_i'' \quad \text{for all } i \\
& \Gamma, x : \sigma_x, \Gamma''', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0'' : \sigma_0 \\
& \implies \{ \text{ by I.H., using } \Gamma, x : \sigma_x, \Gamma''', \overline{\alpha_l}, \overline{u_m} \vdash_2 x : \sigma_0 \text{ from above, and} \\
& \quad (\vdash_2\text{-WEAK}), \text{ at } H''_{\downarrow} = (H'', x : \sigma_x =^! e_0'') \} \\
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, x : \sigma_x =^! e_0'', H''; x; S''' \rangle : \sigma \\
& \Longleftrightarrow \{ \text{ by } (\vdash_2\text{-CONFIG}) \text{ and definition of trans } \} \\
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H''; e_0''; \#x : \sigma_x, S''' \rangle : \sigma
\end{aligned}$$

as required.

case $S = R, S'$

By the inductive hypothesis, for S''' there exists $k''', H''', E''', x''', \sigma''', e'''$. Let $k = k'''$, $H' = H'''$, $E = E'''[R[\cdot]]$, $x' = x'''$, $\sigma' = \sigma'''$, $e = e'''$. Then, for all H , e_0 :

(i):

$$\begin{aligned}
& \text{trans} \langle H; e_0; R, S' \rangle \\
& = \{ \text{ by definition of trans } \} \\
& \text{trans} \langle H; R[e_0]; S' \rangle
\end{aligned}$$

and the result follows.

(ii):

$$\begin{aligned}
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H; e_0; R, S' \rangle : \sigma \\
& \Longleftrightarrow \{ \text{ by } (\vdash_2\text{-CONFIG}) \text{ and definition of trans } \} \\
& \overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H; R[e_0]; S' \rangle : \sigma \\
& \implies \{ \text{ by I.H. } \} \\
& \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\
& \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_i : \sigma'_i \quad \text{for all } i \\
& \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 R[e_0] : \sigma''' \\
& \implies \{ \text{ by Lemma D.2 } \}
\end{aligned}$$

$$\begin{aligned}
& \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_i : \sigma_i \quad \text{for all } i \\
& \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_i : \sigma'_i \quad \text{for all } i \\
& \exists \sigma_0 . \Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e_0 : \sigma_0
\end{aligned}$$

as required.

(iid): We have by assumption that

$$\begin{aligned}
& \Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_i : \sigma''_i \quad \text{for all } i \\
& \Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e''_0 : \sigma_0
\end{aligned}$$

$$\implies \{ \text{by Lemma D.2} \}$$

$$\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 R[e''_0] : \sigma'''_0$$

$$\implies \{ \text{by I.H.} \}$$

$$\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H''; R[e''_0]; S' \rangle : \sigma$$

$$\iff \{ \text{by } (\vdash_2\text{-CONFIG}) \text{ and definition of trans} \}$$

$$\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H''; e''_0; R, S' \rangle : \sigma$$

as required. \square

Our next lemma says that an atom of the same type as a variable may be substituted for it. This allows us to perform atomic (i.e., A-normal form) β -reduction. Thanks to the A-normal form restriction, this is fairly easy.

Lemma D.5 (Substitution)

For all $\Gamma, e, \overline{x_i}, \overline{a_i}, \overline{\sigma_i}, \sigma$, if $\Gamma \vdash_2 a_i : \sigma_i$ for all i , and $\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : \sigma$, then $\Gamma \vdash_2 e[\overline{a_i}/\overline{x_i}] : \sigma$.

Proof Proof is by induction on the structure of e . (In retrospect, this would probably have been easier on the structure of the typing proof tree.) For each case, by Lemma D.1, we may assume that the proof tree of $\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : \sigma$ is of the form

$$\frac{\displaystyle \frac{\vdots}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : \sigma''} (\text{not}(\vdash_2\text{-SUB})) \quad \sigma'' \preceq \sigma}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : \sigma} (\vdash_2\text{-SUB})$$

and we need only prove $\Gamma \vdash_2 e[\overline{a_i}/\overline{x_i}] : \sigma''$, since the desired result follows from $(\vdash_2\text{-SUB})$.

case a

An atomic expression must have one of the following forms:

case x_i

By $(\vdash_2\text{-VAR})$, $\sigma'' = \sigma_i$, and by definition $x_i[\overline{a_i}/\overline{x_i}] = a_i$. By assumption $\Gamma \vdash_2 a_i : \sigma_i$, and we are done.

case x , where $x \notin \overline{x_i}$

Here $x[\overline{a_i}/\overline{x_i}] = x$, and by $(\vdash_2\text{-VAR})$ our result follows.

case $a \tau$

By inspection of the type rules, the proof tree must be of the form

$$\frac{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 a : (\forall \alpha . \tau'')^{\kappa''}}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 a \tau : \sigma''} (\vdash_2\text{-TYAPP})$$

where $\sigma'' = (\tau''[\tau/\alpha])^{\kappa''}$. By definition, $(a \tau)[\overline{a_i}/\overline{x_i}] = a[\overline{a_i}/\overline{x_i}] \tau$, and by the inductive hypothesis and $(\vdash_2\text{-TYAPP})$ again our result follows.

case $a \kappa$

Similar to the $a \tau$ subcase.

case n

$n[\overline{a_i}/\overline{x_i}] = n$, and by $(\vdash_2\text{-INT})$ the result follows.

case $K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j}$

By inspection of the type rules, the proof tree must be of the form

$$\frac{\begin{array}{l} \sigma_{ij}^\circ = \sigma_{ij}[\kappa/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } j \\ \Gamma \vdash_2 a_j : \sigma_{ij}^\circ \quad \text{all } j \\ |\sigma_{ij}^\circ| \leq \kappa \quad \text{all } j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} \end{array}}{\Gamma \vdash_2 K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j} : \sigma''} (\vdash_2\text{-CON})$$

where $\sigma'' = (T \overline{\kappa_l} \overline{\tau_k})^\kappa$ and $\chi = \kappa^\dagger$. By definition

$$(K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j})[\overline{a_i}/\overline{x_i}] = K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j}[\overline{a_i}/\overline{x_i}]$$

and by I.H. and $(\vdash_2\text{-CON})$ the result follows.

case $\lambda^{\kappa, \chi} x : \sigma_x . e$, where (wlog) $x \notin \overline{x_i}$, $x \notin \text{dom}(\Gamma)$, $x \notin \text{fv}(\overline{a_i})$

By inspection of the type rules, the proof tree must be of the form

$$\frac{\begin{array}{l} \Gamma, \overline{x_i} : \overline{\sigma_i}, x : \sigma_x \vdash_2 e : \sigma' \\ \text{occur}(x, e) > 1 \Rightarrow |\sigma_x| = \omega \\ \text{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \quad \text{all } y \in \Gamma \\ \text{occur}(x_i, e) > 0 \Rightarrow |\sigma_i| \leq \kappa \quad \text{all } i \end{array}}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 \lambda^{\kappa, \chi} x : \sigma_x . e : \sigma''} (\vdash_2\text{-ABS})$$

where $\sigma'' = (\sigma_x \rightarrow \sigma')^\kappa$ and $\chi = \kappa^\dagger$. By $(\vdash_2\text{-WEAK})$ we may establish the preconditions for applying the I.H., yielding $\Gamma, x : \sigma_x \vdash_2 e[\overline{a_i}/\overline{x_i}] : \sigma'$. Since x is fresh with respect to the substitution, the first *occur* is unaffected. For the second, note that (since $y \notin \overline{x_i}$)

$$\text{occur}(y, e[\overline{a_i}/\overline{x_i}]) = \text{occur}(y, e) + \sum_i (\text{occur}(y, a_i) \cdot \text{occur}(x_i, e))$$

Thus $\text{occur}(y, e[\overline{a_i}/\overline{x_i}]) > 0$ either means $\text{occur}(y, e) > 0$, in which case we're fine, or that for some i , $\text{occur}(y, a_i) = 1$ and $\text{occur}(x_i, e) > 0$, in which case $|\Gamma(y)| = |\sigma_i| \leq \kappa$ and we're also fine. Finally, for the third, note that the number of occurrences may only decrease, not increase, and thus we may derive by (\vdash_2 -ABS) the desired result.

case $e \ a$

By inspection of the type rules, the proof tree must be of the form

$$\frac{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : (\sigma_1 \rightarrow \sigma'')^1 \quad \Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 a : \sigma_1}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e \ a : \sigma''} (\vdash_2\text{-APP})$$

By definition, $(e \ a)[\overline{a_i}/\overline{x_i}] = e[\overline{a_i}/\overline{x_i}] \ a[\overline{a_i}/\overline{x_i}]$, and by two applications of the inductive hypothesis and (\vdash_2 -APP) again our result follows.

case $\Lambda \alpha . e$ where (wlog) $\alpha \notin \text{ftv}(a_i)$

By definition, if α fresh then $(\Lambda \alpha . e)[\overline{a_i}/\overline{x_i}] = \Lambda \alpha . e[\overline{a_i}/\overline{x_i}]$, and the result follows by I.H. and (\vdash_2 -TYABS).

case $e \ \tau$

Identical to the $a \ \tau$ subcase of the a case.

case $\Lambda u . e$

Similar to the $\Lambda \alpha . e$ case.

case $e \ \kappa$

Identical to the $a \ \kappa$ subcase of the a case.

case $e : (T \ \overline{\tau_k})^\kappa$ of $\overline{K_i} \ \overline{x_{ij}} \rightarrow e_i$, where (wlog) $\overline{x_{ij}} \not\downarrow \overline{x_i}$, $\overline{x_{ij}} \not\downarrow \text{dom}(\Gamma)$

By inspection of the type rules, the proof tree must be of the form

$$\frac{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 e : (T \ \overline{\tau_k})^\kappa \quad \Gamma, \overline{x_i} : \overline{\sigma_i}, x_{ij} : t_{ij}[\overline{t_k}/\overline{\alpha_k}] \vdash_2 e_i : \sigma'' \quad \text{all } i \quad \text{occur}(x_{ij}, e_i) > 1 \Rightarrow \kappa = \omega \quad \text{all } i, j}{\Gamma, \overline{x_i} : \overline{\sigma_i} \vdash_2 \text{case } e : (T \ \overline{\tau_k})^\kappa \text{ of } \overline{K_i} \ \overline{x_{ij}} \rightarrow e_i : \sigma''} (\vdash_2\text{-CASE})$$

We apply the inductive hypothesis to e , and to each of the e_i (using (\vdash_2 -WEAK) to establish $\Gamma, \overline{x_l} : \overline{\sigma_l}, x_{ij} : t_{ij}[\overline{t_k}/\overline{\alpha_k}] \vdash_2 a_l : \sigma_l$ in each case). The occur clause is unchanged due to the freshness of the $\overline{x_{ij}}$. By (\vdash_2 -CASE) then we are done.

case $e_1 + e_2$

By I.H. twice and (\vdash_2 -PRIMOP).

case $\text{add}_n \ e_2$

By I.H. and (\vdash_2 -PRIMOP-R).

case $\text{if0 } e \text{ then } e_1 \text{ else } e_2$

By I.H. three times and (\vdash_2 -IF0).

case letrec $\overline{x'_i : \sigma'_i =_{\chi_i} e_i}$ in e where (wlog) $\overline{x'_i} \not\leq \overline{x_i}$, $\overline{x'_i} \not\leq \text{dom}(\Gamma)$, $\overline{x'_i} \not\leq \text{fv}(\overline{a_i})$.

By inspection of the type rules, the proof tree must be of the form

$$\frac{\begin{array}{l} \Gamma, \overline{x'_j : \sigma'_j} \vdash_2 e_i : \sigma'_i \quad \text{for all } i \\ \Gamma, \overline{x'_j : \sigma'_j} \vdash_2 e : \sigma'' \\ \left(\text{occur}(x'_i, e) + \sum_{j=1}^n \text{occur}(x'_i, e_j) \right) > 1 \Rightarrow |\sigma'_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \vdash_2 \text{letrec } \overline{x'_i : \sigma'_i =_{\chi_i} e_i} \text{ in } e : \sigma''} \quad (\vdash_2\text{-LETREC})$$

where $\chi_i = |\sigma'_i|^\dagger$ for all i . By $(\vdash_2\text{-WEAK})$ and I.H. we may establish the first two clauses; the third is unchanged due to the freshness of the $\overline{x_i}$. By $(\vdash_2\text{-LETREC})$, then, we are done. \square

We also prove similar results for type and usage substitution.

Lemma D.6 (Type substitution)

For all $\Gamma, \alpha, e, \tau_1^\kappa, \tau$, if $\Gamma, \alpha \vdash_2 e : \tau_1^\kappa$, then $\Gamma \vdash_2 e[\tau/\alpha] : (\tau_1[\tau/\alpha])^\kappa$.

Proof By induction on the structure of the proof tree of $\Gamma, \alpha \vdash_2 e : \tau_1^\kappa$. \square

Lemma D.7 (Usage substitution)

For all $\Gamma, u, e, \tau^{\kappa_1}, \kappa$, if $\Gamma, u \vdash_2 e : \tau^{\kappa_1}$ where $u \notin (\text{fv}(\Gamma) \cup \text{fv}(\kappa))$, then $\Gamma \vdash_2 e[\kappa/u] : (\tau[\kappa/u])^{\kappa_1}$.

Proof By induction on the structure of the proof tree of $\Gamma, \alpha \vdash_2 e : \tau_1^\kappa$. The only interesting case is in $(\vdash_2\text{-ABS})$, $(\vdash_2\text{-LETREC})$, and $(\vdash_2\text{-CON})$, where a usage annotation κ_0 is used to compute an update flag $\chi = \kappa_0^\dagger$. If $\kappa_0 \in \{1, \omega\}$ then the substitution causes no change, but if $\kappa_0 = u$ then $\kappa_0[\kappa/u] = \kappa$. Now when $\kappa = 1$ the weakened form of the typing judgement (Section D.1, p. 275) becomes necessary, for we require $1^\dagger = !$. \square

Lemma D.8 (Pruning)

If $\Gamma, x_0 : \sigma_0 \vdash_2 e : \sigma$ and $\text{occur}(x_0, e) = 0$ and $\text{occur}(x_0, e_i) = 0$ for all i , then $\Gamma \vdash_2 e : \sigma$.

Proof Proof is by induction on the structure of the proof tree of $\Gamma, x_0 : \sigma_0 \vdash_2 e : \sigma$. Assume for contradiction that the mapping for x_0 is required. Only $(\vdash_2\text{-VAR})$ inspects the environment, and if this refers to x_0 then x_0 occurs once in that expression. By the definition of occur , this propagates upwards, and so at least one of $\text{occur}(x_0, e)$, $\text{occur}(x_0, e_i)$ is non-zero, which is a contradiction. \square

Our next lemma says that a well-typed shallow context containing a value is always reducible and the result is always well-typed. This allows us to perform reductions.

Lemma D.9 (Delta progress)

For all R, V, Γ, σ , if $\Gamma \vdash_2 R[V] : \sigma$ then there exists e such that $R[V] \mapsto_\delta e$, and $\Gamma \vdash_2 e : \sigma$.

Proof By cases on R . As in the proof of Lemma D.5, we use Lemma D.1 to assume that the root of the proof tree of $\Gamma \vdash_2 R[V] : \sigma$ is not an instance of $(\vdash_2\text{-SUB})$; we leave this step implicit below to avoid clutter.

case $[\cdot] a$

We have $\Gamma \vdash_2 V a : \sigma$; hence by $(\vdash_2\text{-APP})$ we have that $\Gamma \vdash_2 V : \sigma_1 \rightarrow \sigma$ and $\Gamma \vdash_2 a : \sigma_1$. By inspection of the type rules, V must be of the form $\lambda^\kappa x : \sigma_1 . e_0$, and by $(\vdash_2\text{-ABS})$ we have that $\Gamma, x : \sigma_1 \vdash_2 e_0 : \sigma$. Thus by $(\rightarrow_\delta\text{-APP})$, $e = e_0[a/x]$, and by Lemma D.5 the result holds.

case $[\cdot] \tau$

We have $\Gamma \vdash_2 V \tau : \sigma$; hence by $(\vdash_2\text{-TYAPP})$ we have that $\Gamma \vdash_2 V : (\forall \alpha . \tau_1)^\kappa$ and $\sigma = (\tau_1[\tau/\alpha])^\kappa$. By inspection of the type rules, V must be of the form $\Lambda \alpha . V'$, and by $(\vdash_2\text{-TYABS})$ we have that $\Gamma, \alpha \vdash_2 V' : \tau_1^\kappa$. Thus by $(\rightarrow_\delta\text{-TYAPP})$, $e = V'[\tau/\alpha]$, and by Lemma D.6 the result holds.

case $[\cdot] \kappa$

Analogously, by $(\vdash_2\text{-UAPP})$, $(\vdash_2\text{-UABS})$, $(\rightarrow_\delta\text{-UAPP})$, and Lemma D.7.

case $[\cdot] : (T \overline{\tau_k})^\kappa$ of $\overline{K_i \overline{x_{ij}} \rightarrow e_i}$

We have $\Gamma \vdash_2 \text{case } V : (T \overline{\tau_k})^\kappa \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i} : \sigma$; hence by $(\vdash_2\text{-CASE})$ we have that $\Gamma \vdash_2 V : (T \overline{\tau_k})^\kappa$ and $\Gamma, \overline{x_{ij}} : \overline{\sigma_{ij}^\circ} \vdash_2 e_i : \sigma$. By inspection of the type rules, V must be of the form $K_i^{\kappa, \chi} \overline{\kappa_l} \overline{\tau_k} \overline{a_j}$, and by $(\vdash_2\text{-CON})$ we have that $\Gamma \vdash_2 a_j : \sigma_{ij}^\circ$. Thus by $(\rightarrow_\delta\text{-CASE})$, $e = e_i[\overline{a_j}/\overline{x_{ij}}]$, and by Lemma D.5 the result holds.

case $[\cdot] + e$

We have $\Gamma \vdash_2 V + e_2 : \sigma$; hence by $(\vdash_2\text{-PRIMOP})$ we have that $\sigma = \text{Int}$, $\Gamma \vdash_2 V : \text{Int}$, and $\Gamma \vdash_2 e_2 : \text{Int}$. By inspection of the type rules, V must be of the form n . Thus by $(\rightarrow_\delta\text{-PRIMOP-L})$, $e = \text{add}_n e_2$, and by $(\vdash_2\text{-PRIMOP-R})$ the result holds.

case $\text{add}_n [\cdot]$

We have $\Gamma \vdash_2 \text{add}_n V : \sigma$; hence by $(\vdash_2\text{-PRIMOP-R})$ we have that $\sigma = \text{Int}$, $\Gamma \vdash_2 V : \text{Int}$. By inspection of the type rules, V must be of the form n' . Thus by $(\rightarrow_\delta\text{-PRIMOP-R})$, $e = n_3$ where $n_3 = n + n'$, and by $(\vdash_2\text{-LIT})$ the result holds.

case $\text{if0 } [\cdot] \text{ then } e_1 \text{ else } e_2$

We have $\Gamma \vdash_2 \text{if0 } V \text{ then } e_1 \text{ else } e_2 : \sigma$; hence by $(\vdash_2\text{-IF0})$ we have that $\Gamma \vdash_2 V : \text{Int}$, $\Gamma \vdash_2 e_1 : \sigma$, $\Gamma \vdash_2 e_2 : \sigma$. By inspection of the type rules, V must be of the form n . Thus by rule $(\rightarrow_\delta\text{-IF0})$, if $n = 0$ then $e = e_1$ else $e = e_2$; in either case the result holds. \square

The next result is the key lemma. It says that a well-typed configuration is either reducible to a well-typed configuration, or terminal or a black hole. This allows us to prove the desired result by simple induction on the length of the computation.

Lemma D.10 (Progress)

For all $FLIXC_2$ configurations C , if $\overline{\alpha_l}, \overline{u_m} \vdash_2 C : \sigma$ then either (i) $C \in \text{Value} \cup \text{BlackHole}$, or (ii) $\exists C' . C \rightarrow_{\overline{\alpha_l}, \overline{u_m}} C'$ and $C \rightarrow_{\overline{\alpha_l}, \overline{u_m}} C' \Rightarrow \overline{\alpha_l}, \overline{u_m} \vdash_2 C' : \sigma$.

Proof By induction on the number of outermost type and usage abstractions in the control of the configuration C , and then by cases on C .

case $\langle H; R[e]; S \rangle$

$\langle H; R[e]; S \rangle \mapsto_{\overline{\alpha_l}, \overline{u_m}} \langle H; e; R, S \rangle$; typing follows immediately by definition of trans.

case $\langle H; \text{letrec } H' \text{ in } e; S \rangle$

$\langle H; \text{letrec } H' \text{ in } e; S \rangle \mapsto_{\overline{\alpha_l}, \overline{u_m}} \langle H, H'; e; S \rangle$. By Lemma D.4, there exists H'', σ_0 such that $\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H' \text{ in } e : \sigma_0$, and by (\vdash_2 -LETREC) we have $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_i : \sigma'_i$ for all i and $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m} \vdash_2 e : \sigma_0$. Now choose $\overline{\alpha_l}, \overline{u_m}$ fresh, and let $\Gamma^{o'} = x_j^{o'} : \forall \overline{u_m} . \forall \overline{\alpha_l} . \sigma'_j$ and $e_j^{o'} = \Lambda \overline{u_m} . \Lambda \overline{\alpha_l} . e'_j[x_i^{o'} \overline{\alpha_l} \overline{u_m} / x_i']$ and $e^o = e[x_i^{o'} \overline{\alpha_l} \overline{u_m} / x_i']$. By (\vdash_2 -UAPP) and (\vdash_2 -TYAPP) we have $\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'}, \overline{\alpha_l}, \overline{u_m} \vdash_2 x_i^{o'} \overline{\alpha_l} \overline{u_m} : \sigma'_i$, and by (\vdash_2 -WEAK) we have $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'}, \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_j : \sigma'_j$ for all j , and $\Gamma, \Gamma', \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'}, \overline{\alpha_l}, \overline{u_m} \vdash_2 e : \sigma_0$. Then by Substitution Lemma D.5 we have that $\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'}, \overline{\alpha_l}, \overline{u_m} \vdash_2 e'_j[x_i^{o'} \overline{\alpha_l} \overline{u_m} / x_i'] : \sigma'_j$ for all j , and $\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'}, \overline{\alpha_l}, \overline{u_m} \vdash_2 e[x_i^{o'} \overline{\alpha_l} \overline{u_m} / x_i'] : \sigma_0$. By (\vdash_2 -UABS) and (\vdash_2 -TYABS) we then have $\Gamma, \Gamma'', \overline{\alpha_l}, \overline{u_m}, \Gamma^{o'} \vdash_2 e^{o'} : \sigma^{o'}$ for all j . Finally, by Lemma D.4 we have that $\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, H'; e^o; S \rangle : \sigma$ as required.

case $\langle H; \Lambda \alpha . e; S \rangle$

$\langle H; \Lambda \alpha . e; S \rangle \mapsto_{\overline{\alpha_l}, \overline{u_m}} \langle H'; \Lambda \alpha' . e'; S' \rangle$ if $\langle H; e[\alpha' / \alpha]; S \rangle \mapsto_{\overline{\alpha_l}, \alpha', \overline{u_m}} \langle H'; e'; S' \rangle$, α' fresh, e not a value. Apply I.H.; if a value then C is a value, if a black hole then C is a black hole. Otherwise, typing follows by (\vdash_2 -TYABS).

case $\langle H; \Lambda u . e; S \rangle$

Similar to previous case.

case $\langle H, x_0 : \sigma_0 =^\bullet e; x_0; S \rangle$

$\langle H, x_0 : \sigma_0 =^\bullet e; x_0; S \rangle \mapsto_{\overline{\alpha_l}, \overline{u_m}} \langle H; e; S \rangle$. By Lemma D.4 we know that there exists H', E, x', σ', e' such that either $\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H, x_0 : \sigma_0 =^\bullet e, H' \text{ in } E[x_0] : \sigma$ where $H' = \emptyset$, or $\overline{\alpha_l}, \overline{u_m} \vdash_2 \text{letrec } H, x_0 : \sigma_0 =^\bullet e, H' \text{ in } e' : \sigma$ where H' contains the binding $x' : \sigma' =^! E[x_0]$. By (\vdash_2 -LETREC) we have that $\Gamma, x_0 : \sigma_0, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e : \sigma_0$, and so by Lemma D.4 we have that $\overline{\alpha_l}, \overline{u_m} \vdash_2 \langle H, x_0 : \sigma_0 =^\bullet e; e_0; S \rangle : \sigma$. Now by (\vdash_2 -LETREC) we also know that, respectively, $(\text{occur}(x_0, E[x_0]) + \sum_{j=1}^n \text{occur}(x_0, e_j) + \text{occur}(x_0, e) + \sum_{j=1}^{n'} \text{occur}(x_0, e'_j)) \leq 1$ or $(\text{occur}(x_0, e') + \sum_{j=1}^n \text{occur}(x_0, e_j) + \text{occur}(x_0, e) + \sum_{j=1}^{n'} \text{occur}(x_0, e'_j)) \leq 1$ where one of the e'_j is $E[x_0]$. In either case, we know that $\text{occur}(x_0, E[x_0]) \geq 1$, and so all the other $\text{occur}(x_0, \cdot)$ are zero. Thus by the Pruning Lemma D.8 we may omit the binding for x_0 , obtaining the desired result.

case $\langle H, x_0 : \sigma_0 =^! e; x_0; S \rangle$

$\langle H, x_0 : \sigma_0 =^! e; x_0; S \rangle \mapsto_{\overline{\alpha_l}, \overline{u_m}} \langle H; e; \#x_0 : \sigma_0, S \rangle$; typing follows immediately by definition of trans.

case $\langle H; x_0; S \rangle$ where $x_0 \notin \text{dom}(H)$ but $x_0 \in \text{dom}(S)$
 $C \in \text{BlackHole}$.

case $\langle H; x_0; S \rangle$ where $x_0 \notin \text{dom}(H) \cup \text{dom}(S)$

By Lemma D.4 and $(\vdash_2\text{-LETREC})$ we have that $x_0 \in \text{dom}(H, H')$; but since by the lemma we also have $\text{dom}(H') = \text{dom}(S)$, we have a contradiction, and so this case does not occur.

case $\langle H; V; \#x_0 : \sigma_0, S \rangle$

By Lemma D.4, $(\vdash_2\text{-LETREC})$, and definition of trans we have that one of the bindings in H' is $x_0 : \sigma_0 =^! V$.

If $|\sigma_0|$ is ω or u for some usage variable u , then by $(\vdash_2\text{-LETREC})$ and $(\vdash_2\text{-SUB})$ the topmost annotation of the type of V must also be ω or u ; and by $(\vdash_2\text{-LIT})$, $(\vdash_2\text{-CON})$, $(\vdash_2\text{-ABS})$, $(\vdash_2\text{-TYABS})$, $(\vdash_2\text{-UABS})$ we have that $|V|$ must be $!$, satisfying the side condition of $(\rightarrow\text{-UPDATE})$.

Alternatively, if $|\sigma_0|$ is 1, then by the same argument as case $(\rightarrow\text{-VAR-ONCE})$ above, there is no reference to x_0 in the configuration apart from the update frame. This also satisfies the side condition of $(\rightarrow\text{-UPDATE})$.

Hence in either case,¹ $\langle H; V; \#x_0 : \sigma_0, S \rangle \rightarrow_{\overline{\alpha_l}, \overline{u_m}} \langle H, x_0 : \sigma_0 =^! V; x_0; S \rangle$. Typing follows immediately by definition of trans.

case $\langle H; V; R, S \rangle$

Since $\text{trans}\langle H; V; R, S \rangle = \text{trans}\langle H; R[V]; S \rangle$, by $(\vdash_2\text{-CONFIG})$ and Lemma D.4 there exists Γ' such that $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 R[V] : \sigma_0$. Now by Lemma D.9 there exists e such that $R[V] \rightarrow_\delta e$ and $\Gamma, \Gamma', \overline{\alpha_l}, \overline{u_m} \vdash_2 e : \sigma_0$; thus $\langle H; V; R, S \rangle \rightarrow_{\overline{\alpha_l}, \overline{u_m}} \langle H; e; S \rangle$, and is well-typed by Lemma D.4 as required.

case $\langle H; V; \varepsilon \rangle$

$C \in \text{Value}$. □

Finally, we reach the theorem. This theorem states that a well-typed program translates into an LX configuration which runs forever, or terminates in *Value* or *BlackHole*; i.e., it never goes *Wrong*. This is the desired result, showing that “well-typed programs do not go wrong”.

Theorem D.11 (Type soundness)

For all $e \in FLIX_2$, if $\emptyset \vdash_2 e : \sigma$ and there exists a configuration C' such that $(e)^{\langle ; ; \rangle} \downarrow C'$, then $C' \in \text{Value} \cup \text{BlackHole}$.

Proof By trans and $\langle ; ; \rangle$ and $(\vdash_2\text{-CONFIG})$ we have $\vdash_2 (e)^{\langle ; ; \rangle} : \sigma$. We may proceed by induction on the length of derivation of $(e)^{\langle ; ; \rangle} \downarrow C'$. By Lemma D.10, the type σ is preserved by reductions. At the final step, (ii) (of Lemma D.10) cannot apply and so (i) implies $C \in \text{Value} \cup \text{BlackHole}$. □

¹The $|\sigma_0| = 1$ case never occurs in $FLIX_1$, because it arises only when a usage variable becomes instantiated to 1.

D.2 Correctness

Theorem D.12 (Correctness)

For all $FLIX_2$ target programs e , where $\emptyset \vdash_2 e : \sigma$, let M be the corresponding FL_0 source program $(e)^\natural$. Then we have

- (i) $(e)^{\langle \cdot \rangle} \downarrow \Leftrightarrow (\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow$
(i.e., the $FLIX_2$ program e terminates iff the FL_0 program M does); and
- (ii) If $(e)^{\langle \cdot \rangle} \downarrow C'$ and $(\mathcal{T}_0 \llbracket M \rrbracket)^{\langle \cdot \rangle} \downarrow C''$, then all the following hold:
 - (a) $C' \in \text{BlackHole} \Leftrightarrow C'' \in \text{BlackHole}$
 - (b) $C' \in \text{Value} \Leftrightarrow C'' \in \text{Value}$
 - (c) $C' = \langle H; n; \varepsilon \rangle \Leftrightarrow C'' = \langle H'; n; \varepsilon \rangle$

(i.e., if the two programs terminate, they both terminate in the same way, viz., black hole, non-ground value, or the same ground value).

Proof By Correspondence Lemma 5.3 we may ignore the instrumentation, and consider the two FLX terms $M_2 = (e)^\flat$ and $M_0 = (M)^\flat$. Call the configuration $(M_2)^{\langle \cdot \rangle}$ the initial *left* configuration, and $(M_0)^{\langle \cdot \rangle}$ the initial *right* configuration. For the purposes of the proof, we allow update frames in the right configuration to be optionally *marked* (thus each frame is either marked or unmarked). By the Progress Lemma D.10 neither side ever gets stuck, so we need consider only valid reductions, and *Value* and *BlackHole*. Furthermore, reduction is deterministic up to the choice of fresh variables.

case “ \Rightarrow ”

Proof proceeds by induction on the length of the left reduction sequence, and we will simulate each left reduction by one or more right reductions. Reduction will preserve the invariants that the left and right configurations differ only in that (i) some update flags on the left are \bullet whereas on the right they are all $!$, (ii) any marked update frames on the right are absent on the left, and (iii) the right heap may have bindings for variables that are not bound in the left heap. Otherwise (iv) the heaps, controls, and stacks are identical.

Initially, the invariants hold trivially.

If the left configuration can be reduced, we proceed by induction on the structure of the derivation of the reduction. The reduction must be by one of the following rules:

case (\rightarrow -UNWIND), (\rightarrow -LETREC), (\rightarrow -VAR-MANY)

In these cases, the same rule applies to the right configuration and the invariants are all preserved.

case $(\rightarrow\text{-TYLAM}), (\rightarrow\text{-ULAM})$

In these cases, the result is established by the inductive hypothesis. The same rule applies to the right configuration and the invariants are all preserved.

case $(\rightarrow\text{-VAR-ONCE})$

On the left the binding for x bears the flag \bullet , but on the right it bears the flag $!$. We reduce the right configuration by rule $(\rightarrow\text{-VAR-MANY})$, but *mark* the resulting update frame. This preserves the invariants even though $(\rightarrow\text{-VAR-ONCE})$ does not push an update frame.

case $(\rightarrow\text{-UPDATE}), (\rightarrow\text{-REDUCE})$

If there are any marked update frames on top of the right stack, we perform an $(\rightarrow\text{-UPDATE})$ on the right configuration first, for each one. The invariants still all hold; we simply have extra bindings in the right heap. Now we perform the $(\rightarrow\text{-UPDATE})$ or $(\rightarrow\text{-REDUCE})$ which was performed on the left, again preserving the invariants.

This simulates the left reduction by one or more right reductions, preserving the invariants.

If the left configuration cannot be reduced, then since we cannot get stuck, this must be because it is in either *Value* or *BlackHole*. If there are any marked update frames on top of the right stack, we perform an $(\rightarrow\text{-UPDATE})$ on the right configuration first, for each one. If the left configuration is in *Value*, it is easy to see that the right configuration must be also. If the left configuration is in *BlackHole*, then there is a variable x in the control which appears in $\text{dom}(S)$ but not in $\text{dom}(H)$. In the right configuration, by the invariants, the same variable must appear in the control and in $\text{dom}(S)$ (by Progress we know that we never get two or more update frames for the same variable, marked or unmarked), but not in $\text{dom}(H)$ (by Progress we know that $\text{dom}(H) \not\subseteq \text{dom}(S)$ is preserved); thus the right configuration is also in *BlackHole*. Finally, if the left configuration is $\langle H; n; \varepsilon \rangle$ then the invariants imply that the right configuration is $\langle H'; n; \varepsilon \rangle$ as required.

case “ \Leftarrow ”

Proof proceeds by induction on the length of the right reduction sequence, and we will simulate each right reduction by zero or one left reductions. Reduction will preserve the same invariants as before.

Initially, the invariants hold trivially.

If the right configuration can be reduced, we proceed by induction on the structure of the derivation of the reduction. The reduction must be by one of the following rules:

case $(\rightarrow\text{-UNWIND}), (\rightarrow\text{-LETREC}), (\rightarrow\text{-REDUCE})$

In these cases, the same rule applies to the left configuration and the invariants are all preserved.

case $(\rightarrow\text{-TYLAM}), (\rightarrow\text{-ULAM})$

In these cases, the result is established by the inductive hypothesis.

The same rule applies to the right configuration and the invariants are all preserved.

case $(\rightarrow\text{-VAR-ONCE})$

This rule never occurs in the right reduction sequence.

case $(\rightarrow\text{-VAR-MANY})$

If the binding for x in the left configuration bears the flag \bullet , mark the frame pushed in the right configuration, and perform $(\rightarrow\text{-VAR-ONCE})$ in the left configuration. This preserves the invariants even though $(\rightarrow\text{-VAR-ONCE})$ does not push a frame.

case $(\rightarrow\text{-UPDATE})$

The side condition $|V| = !$ is satisfied in the left configuration because otherwise the left configuration would become stuck, and we know by Progress that this does not occur.

If the update frame is unmarked, perform the same update on the left configuration. If the update frame is marked, do nothing to the left configuration. This preserves the invariants.

This simulates the right reduction by zero or one left reductions, preserving the invariants.

If the right configuration cannot be reduced, the same argument as before applies to show that both configurations agree on *Value*, *BlackHole*, and literals, as required (this time there is no need to process marked update frames in the right configuration, since none occur). \square

D.3 Constraints

Constraints were informally introduced in Section 3.5.1. Informally, a constraint is simply a set of equalities $\langle \kappa = \kappa' \rangle$ or inequalities $\langle \kappa \leq \kappa' \rangle$ which constrain the valid assignments to the variables. A solution to a constraint is an assignment that satisfies all the equalities and inequalities.

We may define this more formally as follows. We are indebted to [OSW98] for much of this formulation.

Let A be the set of *constants* $\{1, \omega\}$. Let \leq be the *partial order* defined in Figure 3.3, restricted to $A \times A$. Let U be an infinite set of *variables*. We write κ to denote an element of $A \cup U$.

A *substitution* S is a total, idempotent function from U to $A \cup U$, which is the identity on all but a finite number of variables. For all $\kappa \in A \cup U$, define $S\kappa = S(\kappa)$ if $\kappa \in U$ and κ otherwise.

An *atomic constraint* $\langle \kappa \leq \kappa' \rangle$ is an element of the set $\Omega = (A \cup U) \times (A \cup U)$. A *constraint* C is an element of $\mathcal{P}_{fin}(\Omega)$; i.e., a finite set of atomic constraints. We write \emptyset to denote the empty constraint, and $C \wedge D$ to denote the union of constraints C and D ; we confuse atomic constraints $\langle \kappa \leq \kappa' \rangle$ with their singleton constraints $\{\langle \kappa \leq \kappa' \rangle\}$. Substitution is extended to constraints: $SC = \{\langle S\kappa \leq S\kappa' \rangle \mid \langle \kappa \leq \kappa' \rangle \in C\}$.

The *atomic entailment relation* $\vdash^e \in \mathcal{P}_{fin}(\Omega) \times \Omega$ is defined as follows: $C \vdash^e \langle \kappa \leq \kappa' \rangle$ holds iff for all substitutions S we have $(\forall \langle \kappa_1 \leq \kappa'_1 \rangle \in C . S\kappa_1 \leq S\kappa'_1) \Rightarrow$

$S\kappa \leq S\kappa'$. The entailment relation $\vdash^e \in \mathcal{P}_{fin}(\Omega) \times \mathcal{P}_{fin}(\Omega)$ is defined as follows: $C \vdash^e D$ iff $\forall \langle \kappa \leq \kappa' \rangle \in D . C \vdash^e \langle \kappa \leq \kappa' \rangle$. We abbreviate $\emptyset \vdash^e C$ by $\vdash^e C$. Constraints C and D are considered *equal*, denoted $C =^e D$, iff $C \vdash^e D$ and $D \vdash^e C$.

A constraint C induces a *partial order* on $A \cup U$. We denote this partial order \leq_C , defined by $\kappa \leq_C \kappa' = C \vdash^e \langle \kappa \leq \kappa' \rangle$. For convenience, we sometimes write $\kappa \leq_C^+ \kappa'$ for $\kappa \leq_C \kappa'$ and $\kappa \leq_C^- \kappa'$ for $\kappa' \leq_C \kappa$.

A *solution* of a constraint C is a substitution S such that $\vdash^e SC$; i.e., for all $\langle \kappa \leq \kappa' \rangle \in C$ it is the case that $S\kappa \leq S\kappa'$. A constraint is *satisfiable* iff it has a solution; it is *unsatisfiable* otherwise.

For notational convenience, we may write the constraint $\{\langle \kappa \leq \kappa' \rangle, \langle \kappa' \leq \kappa \rangle\}$ as $\langle \kappa = \kappa' \rangle$. If B is true (respectively false), then $\{B \Rightarrow C\}$ denotes C (respectively \emptyset). Note that this is *not* a conditional constraint; B must statically be either true or false. We may write $\{\sigma'_1 \preceq \sigma_1\}$ to denote the least set of atomic constraints such that $\sigma'_1 \preceq \sigma_1$ is derivable by the rules given in Figure 3.3 (reading $\langle \kappa \leq \kappa' \rangle$ for $\kappa \leq \kappa'$); note that this means that structure-mismatched constraints such as $\{\text{Int}^{u_1} \preceq (\text{Int}^{u_2} \rightarrow \text{Int}^{u_3})^{u_4}\}$ are undefined.

D.4 Soundness of inference phase 1

We now prove some results about the inference.

We may define the *annotations* of a type in a similar manner to the free usage variables (Figure 5.8). Define $\text{ann}^\varepsilon(\sigma)$ to be the set of ε -ve annotations of σ , as shown in Figure B.10. We extend the definition of $\text{ann}^\varepsilon(\cdot)$ to environments as follows:

$$\text{ann}^\varepsilon(\Gamma) = \bigcup_{x \in \Gamma} \text{ann}^\varepsilon(\Gamma(x))$$

If ε is omitted, we compute the union of the positive and the negative annotations.

Lemma D.13 (Closure operation)

For all $C, \Gamma, \overline{\sigma_i}$ such that $\exists S' . \vdash^e S'C$ (i.e., C is satisfiable),

(i) The closure operation $\text{Clos}(C, \Gamma, \overline{\tau_i \kappa_i}) = (C', \overline{u_i}, S)$ is well-defined,

and we have the following results, where $F_0 \triangleq (\text{fuv}(\Gamma) \cup \text{fuv}(\overline{\kappa_i}))$:

(ii) $C' =^e SC$

(i.e., a solution of the residual constraint, applied to the substituted term, satisfies all the original constraints);

(iii) $\exists S' . \vdash^e S'C'$

(i.e., the residual constraint is satisfiable);

(iv) For all substitutions S', S'' such that $S'|_{U \setminus \overline{u_i}} = S''|_{U \setminus \overline{u_i}}$ (where U is the set of all usage variables), we have that $\vdash^e S'C' \Leftrightarrow \vdash^e S''C'$

(i.e., the $\overline{u_i}$ may safely take any value; alternatively, the residual constraint is independent of the values of the $\overline{u_i}$);

- (v) $\text{dom}(S) \subseteq \text{fuv}(C) \setminus F_0$, and for all $x \in F_0$ and $\kappa \in \{1, \omega\}$, if $\exists S' . \vdash^e S'(C \wedge \langle x = \kappa \rangle)$ then $\exists S' . \vdash^e S'(C' \wedge \langle x = \kappa \rangle)$.
(i.e., the substitution does not attempt to touch variables to which it is not applied, and neither does C' constrain them further); and
- (vi) $\overline{u_i} \subseteq (\text{fuv}(S\overline{\tau_i}) \setminus F_0)$
(i.e., the variables $\overline{u_i}$ are all abstractable).
- (vii) $\forall u \in \text{fuv}^+(\overline{\tau_i}) . Su \neq 1$
(i.e., no positive annotation is forced to 1).

Proof (i) holds by inspection of the algorithm (up to the choice of representative of each equivalence class); the only possible source of failure is the invocation of *TransitiveClosure*, but by assumption C is satisfiable.

Observe that $C =^e C_0$, since *TransitiveClosure* merely builds a data structure to represent C_0 and tests for satisfiability.

(ii) holds trivially, by definition.

For (iii) and (iv), since C' is simply C under the mapping S , it is sufficient to show that (a) no constraint $\langle x \leq y \rangle$ in C can be mapped by S to one of the failure candidates $\langle 1 \leq \omega \rangle$, $\langle u_i \leq \omega \rangle$, $\langle 1 \leq u_j \rangle$, or $\langle u_i \leq u_j \rangle$ where $i \neq j$, and (b) that no variable is mapped to more than one variable or constant.

(b) holds because the conditions on x in the definition of S are mutually exclusive, and if there are two pairs of variables u_i^-, v_i^+ , $i = 1, 2$ such that $u_i \leq_C x \wedge x \leq_C v_i$, it follows by transitivity that $u_1 \leq_C v_2$ and thus $u_1 \sim v_2 \sim u_2$, and thus $[u_1]_{(\sim)} = [u_2]_{(\sim)}$. Since we choose a single representative from each class, x must be mapped to the same representative in each case.

We show (a) by cases, assuming each failure candidate and deriving a contradiction in each case.

case $x \mapsto 1, y \mapsto \omega$ and $x \mapsto u_i, y \mapsto \omega$

We have $\exists u^- \in G' . u \leq_C x$ and $\neg \exists u_*^- \in G' . u_* \leq_C y$ by definition of the mapping. But $u \leq_C x \leq_C y$ implies $u \leq_C y$, a contradiction.

case $x \mapsto u_i, y \mapsto u_j, i \neq j$

We have $\exists u^- \in G' . u \leq_C x$, $\exists u_*^- \in G' . u_* \leq_C y$, $\exists v_*^+ \in G' . y \leq_C v_*$ by definition of the mapping. But $u \leq_C x \leq_C y \leq_C v_*$ implies $u \leq_C v_*$, and hence $u \sim v_*$. But $u_* \leq_C y \leq_C v_*$, and thus $u_* \sim v_*$ also; hence $u \sim u_*$ and $[u]_{(\sim)} = [u_*]_{(\sim)}$ and finally $u_i = u_j$, a contradiction.

case $x \mapsto 1, y \mapsto u_j$

We have $\neg \exists v^+ \in G' . x \leq_C v$ and $\exists v_*^+ \in G' . y \leq_C v_*$ by definition of the mapping. But $x \leq_C y \leq_C v_*$ implies $x \leq_C v_*$, a contradiction.

Since no constraint in C is mapped to $\langle 1 \leq \omega \rangle$, no matter what value is given to the $\overline{u_i}$, both (iii) and (iv) are satisfied.

For (v), observe that $F_0 \setminus F \subseteq \text{dom}(S_0)$.

For the first part, assume for contradiction that there is an $x \in F_0$ that is also in $\text{dom}(S)$. If $x \in \text{dom}(S_0)$ then we directly contradict the definition of S , so we may assume $x \in F$. Then by definition of S we have $\exists u^- \in G' . u \leq_C x$. But since $G' = \text{gfp}(\Phi)$ we have that $G' = \Phi(G')$, and specifically from the third clause of the definition of Φ that $\neg \exists x \in F . x \leq_C u$, i.e., $\neg \exists x \in F . u \leq_C x$, a contradiction.

For the second part, assume for contradiction that there is an $x \in F_0$ and a $\kappa \in \{1, \omega\}$ such that $\exists S' . \vdash^e S'(C \wedge \langle x = \kappa \rangle)$ but $\neg \exists S' . \vdash^e S'(C' \wedge \langle x = \kappa \rangle)$. From the latter we may derive that $C' \vdash^e \langle x = \bar{\kappa} \rangle$, where $\bar{1} = \omega$ and $\bar{\omega} = 1$. Consider first the case where $\kappa = \omega$ and $\bar{\kappa} = 1$. For this to occur, a mapping $x' \mapsto 1$ must occur in the third conjunct of the definition of S , for some variable x' (possibly the same variable) with $x' \notin \text{dom}(S_0)$ and $x' \leq_C x$. Now if $x \in \text{dom}(S_0)$ then $C \vdash^e \langle x = \omega \rangle$, and so $C \vdash^e \langle x' = \omega \rangle$ also, and $x' \in \text{dom}(S_0)$, a contradiction. So $x \notin \text{dom}(S_0)$. Furthermore, since $u^- \leq_C x' \leq_C x$, we have $x \mapsto 1$ also. So we need only consider the case where $x = x'$. Now, by definition of S , we have $\exists u^- \in G' . u \leq_C x$. But since $G' = \text{gfp}(\Phi)$ we have that $G' = \Phi(G')$, and specifically from the third clause of the definition of Φ that $\neg \exists x \in F . x \leq_C u$, i.e., $\neg \exists x \in F . u \leq_C x$, a contradiction. A symmetric argument applies in case $\kappa = 1$ and $\bar{\kappa} = \omega$.

For (vi), we prove firstly that each u_i occurs at least once in SC or $S\bar{\tau}_i$, and secondly that no u_i occurs in F_0 . Observe that each u_i is the representative element of the equivalence class $[u_i]_{(\sim)}$, and by the definition of \mathcal{U} we have that for some ε , $u_i^\varepsilon \in G'$.

For the first, observe that by the second clause of the definition of Φ , for each u_i we have that there exists some $v_i^\varepsilon \in G'$ such that $u_i \leq_C^\varepsilon v_i$. By the definition of S , this implies that $Su_i = u_i$. Now by the first clause of the definition we know $G' \subseteq G$, and so $u_i \in G' \subseteq G \subseteq G_0 = fuv(\bar{\tau}_i)$. Since $Su_i = u_i$, we have that $u_i \in fuv(S\bar{\tau}_i)$ as required.

For the second, assume for contradiction that some $u_i \in F_0$. Since $u_i \in G' \subseteq G = G_0 \setminus \text{dom}(S_0)$, $u_i \in \text{dom}(S_0)$ leads to an immediate contradiction, and so we need consider only $u_i \in F$. Now observe that by the third clause of the definition of Φ we have that $\neg \exists x \in F . x \leq_C^\varepsilon u_i$. But by reflexivity of (\leq_C) and the assumption, this is a contradiction.

Finally, for (vii), assume for contradiction that for some $u_* \in fuv^+(\bar{\tau}_i)$ it is the case that $Su_* = 1$. Now clearly $u_*^+ \in G_0$, and in fact (since it is in the domain of S and hence not in the domain of S_0), $u_*^+ \in G$. But by definition of S , $\exists u^- \in G' . u \leq_C u_*$ and $\neg \exists v^+ \in G' . u_* \leq_C v$; specifically, since $u_* \leq_C u_*$, we have $u_*^+ \notin G'$. The third clause of Φ states that $\forall u^\varepsilon \in G' . \neg \exists x \in (F \cup \{v \mid v^\varepsilon \in (G \setminus A)\}) . x \leq_C^\varepsilon u$, and note that u_*^+ is in the set of possible values for x . Since $u_* \leq_C u$, we have that $u^- \notin G'$, a contradiction. \square

Theorem D.14 (Soundness of inference phase 1)

For all Γ in $FLIX_2$ (possibly with free usage variables) and M, t in L_0 such that $(\Gamma)^\natural \vdash_0 M : t$ and $1 \notin \text{ann}^+(\Gamma)$, and all annotated data type declarations satisfying $1 \notin \text{ann}(\bar{\sigma}_{ij})$ (see Section 5.4.3),

- (i) $\blacktriangleright_2 (\Gamma, M) = (e', \sigma', C, V)$ is well defined²
(i.e., the algorithm \blacktriangleright_2 is deterministic and does not fail);
- (ii) $(e')^\sharp = M$ and $(\sigma')^\sharp = t$
(i.e., the inference algorithm merely annotates the source term, and does not alter it or its source type);
- (iii) $\forall x \in \Gamma. V(x) = \text{occur}(x, e')$
(i.e., V correctly implements the *occur* function);
- (iv) $\forall S. \vdash^e SC \Rightarrow S\Gamma \vdash_2 Se' : S\sigma'$
(i.e., all solutions of the resulting constraint are well-typed); and
- (v) $\exists S. \vdash^e S(C \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv(\sigma'))} \langle u = \omega \rangle)$ and $1 \notin \text{ann}^+(\sigma')$
(i.e., the resulting constraint has at least one solution, and permits all possible future uses).

Proof

The proof proceeds by induction on the structure of M . Note that in each case, at most one \vdash_0 rule and one \blacktriangleright_2 rule applies; thus the structure of the proof trees for these relations is fixed by M . In every case, (ii) and (iii) are shown trivially by inspection and the inductive hypothesis, and so we omit the details.

case A

case x

- (i) Since $(\Gamma)^\sharp \vdash_0 x : t$, by $(\vdash_0\text{-VAR})$ we have $x : \sigma \in \Gamma$ where $(\sigma)^\sharp = t$. In general, $\sigma = (\forall \overline{u_i}. \tau)^\kappa$, τ a usage-monotype. Thus by $(\blacktriangleright_2\text{-VAR})$, the algorithm is deterministic up to fresh variables, and does not fail.
- (iv) Since $C = \emptyset$, we must show that for all S , $S\Gamma \vdash_2 S(x \overline{v_i}) : S(\tau[\overline{v_i}/\overline{u_i}])^\kappa$ where $\Gamma(x) = (\forall \overline{u_i}. \tau)^\kappa$; i.e., $S\Gamma \vdash_2 x \overline{Sv_i} : (S\tau[\overline{Sv_i}/\overline{u_i}])^{S\kappa}$, which holds by $(\vdash_2\text{-VAR})$ and $(\vdash_2\text{-UAPP})$.
- (v) Let $S = \lambda u. \omega$; then $\vdash^e S(\bigwedge_{u \in (fuv(\Gamma) \cup fuv((\tau[\overline{v_i}/\overline{u_i}])^\kappa))} \langle u = \omega \rangle)$; since $C = \emptyset$, the first part follows. Since by assumption $1 \notin \text{ann}^+(\Gamma)$, we have $1 \notin \text{ann}^+((\forall \overline{u_i}. \tau)^\kappa)$, and so $1 \notin \text{ann}^+(\tau^\kappa)$, and since $\overline{v_i}$ fresh, this implies $1 \notin \text{ann}^+((\tau[\overline{v_i}/\overline{u_i}])^\kappa)$ as required.

case $A t$

- (i) By $(\blacktriangleright_2\text{-TYAPP})$, $(\vdash_0\text{-TYAPP})$, and I.H.
- (iv) By I.H. and definition of substitution, $S\Gamma \vdash_2 Se : (\forall \alpha. S\tau_1)^{S\kappa}$. By $(\vdash_2\text{-TYAPP})$, $S\Gamma \vdash_2 Se S\tau_2 : (S\tau_1[S\tau_2/\alpha])^{S\kappa}$ as required.
- (v) By I.H., $\exists S. \vdash^e S(C \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv((\forall \alpha. \tau)^\kappa) \langle u = \omega \rangle))$ and $1 \notin \text{ann}^+((\forall \alpha. \tau)^\kappa)$. Since $\text{ann}(\tau_2)$ are all fresh usage variables, and

²It is well defined modulo the names of fresh variables; we have already noted that we are omitting the details of fresh variable management. Here this also means that the list of usage variables in scope occurs in Γ of $\Gamma \vdash_2 e : \sigma$ but not in Γ of $\blacktriangleright_2 (\Gamma, M)$, since in the latter the free usage variables are managed separately.

$ann^+((\tau_1[\tau_2/\alpha])^\kappa) \subseteq ann^+((\forall\alpha . \tau)^\kappa) \cup ann(\tau_2)$, we may form $S' = S \cup \{ann(\tau_2) \mapsto \omega\}$, establishing the desired results.

case n

All parts trivial by $(\vdash_0\text{-LIT})$, $(\blacktriangleright_2\text{-LIT})$, $(\vdash_2\text{-LIT})$.

case $K_i \overline{t_k} \overline{A_j}$

(i) $(\blacktriangleright_2\text{-CON})$ applies. By $(\vdash_0\text{-CON})$ we have $\Gamma \vdash_0 A_j : t_{ij}^\circ$ where $t_{ij}^\circ = t_{ij}[\overline{t_k}/\overline{\alpha_k}]$, enabling us to apply the I.H. By I.H.(ii) we have that $(\sigma'_j)^\natural = t_{ij}^\circ$, and since $(\sigma_{ij}^\circ)^\natural = t_{ij}^\circ$ the types are compatible in the subtype constraints of C_1 . Thus $(\blacktriangleright_2\text{-CON})$ is deterministic and does not fail.

(iv) By I.H. we have $\forall S . \vdash^e SC_1^j \Rightarrow ST \vdash_2 Sa_j : S\sigma'_j$; since $C \vdash^e SC_1^j$ we have $\vdash^e SC \Rightarrow ST \vdash_2 Sa_j : S\sigma'_j$. We also have $C \vdash^e \{\sigma'_j \preceq \sigma_{ij}^\circ\}$ and $C \vdash^e \{|\sigma_{ij}^\circ| \leq v\}$. and by $(\vdash_2\text{-CON})$ and $(\vdash_2\text{-SUB})$ the result follows.

(v) By I.H., we have S_j such that $\vdash^e S(C_1^j \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv(\sigma'_j))} \langle u = \omega \rangle)$ and $1 \notin ann^+(\sigma'_j)$. Now since all the S_j agree on $fuv(\Gamma)$, and (by induction) $\text{dom}(S_j) \setminus fuv(\Gamma)$ is fresh, we can form $S = \bigcup_j S_j \cup \{x \mapsto \omega \mid x \in fuv(\overline{\tau_k}, v, \overline{v_l})\}$ and this satisfies all the C_1^j . Furthermore, it satisfies C_1 since all usage variables in σ_{ij} are mapped to ω and $1 \notin ann(\sigma_{ij})$ by assumption (Section 5.4.3), and it satisfies C_2 similarly. The result follows.

case $\lambda x : t . M$

(i) By $(\blacktriangleright_2\text{-ABS})$, $(\vdash_0\text{-ABS})$, and I.H.

(iv) By I.H. we have that for all S such that $\vdash^e SC_1, ST, x : S\sigma_1 \vdash_2 Se : S\sigma_2$. Now $C_2 \vdash^e \{V(x) > 1 \Rightarrow \langle |\sigma_1| = \omega \rangle\}$, and $C_3 \vdash^e \bigwedge_{y \in \Gamma} \{V(y) > 0 \Rightarrow \langle |\Gamma(y)| = \omega \rangle\}$. So by $(\vdash_2\text{-ABS})$ we have $ST \vdash_2 S(\lambda^{v,v^\dagger} x : \sigma_1 . e) : S((\sigma_1 \rightarrow \sigma_2)^v)$ as required.

(v) By I.H., $\exists S . \vdash^e S(C \wedge \bigwedge_{u \in (fuv(\Gamma, x:\sigma_1) \cup fuv(\sigma_2))} \langle u = \omega \rangle)$ and $1 \notin ann^+(\sigma_2)$. If we extend S to map $v, fuv(\sigma_1)$ to ω , we satisfy all the desired constraints, and since $1 \notin ann^+(\sigma_2)$, $1 \notin ann^-(\sigma_1)$, and $1 \neq v$, the result follows.

case $M A$

(i) $(\blacktriangleright_2\text{-APP})$ applies. By $(\vdash_0\text{-APP})$ we have $\Gamma \vdash_0 M : t_1 \rightarrow t_2$ and $\Gamma \vdash_0 A : t_1$, enabling us to apply the I.H. By I.H.(ii) we have that $(\sigma'_1)^\natural = t_1$ and $((\sigma_1 \rightarrow \sigma_2)^\kappa)^\natural = t_1 \rightarrow t_2$, and so the types are compatible in the subtype constraint C_3 . Thus $(\blacktriangleright_2\text{-APP})$ is deterministic and does not fail.

(iv) Let S be a solution of $C_1 \wedge C_2 \wedge C_3$. Then by I.H., $ST \vdash_2 SM : S((\sigma_1 \rightarrow \sigma_2)^\kappa)$ and $ST \vdash_2 SA : S(\sigma'_1)$; we also have by C_3 that $S\sigma'_1 \preceq S\sigma_1$. Thus by $(\vdash_2\text{-SUB})$ and $(\vdash_2\text{-APP})$, the result follows.

(v) By I.H., we have S_1, S_2 such that $\vdash^e S(C_1 \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv((\sigma_1 \rightarrow \sigma_2)^\kappa))} \langle u = \omega \rangle)$, $\vdash^e S(C_1 \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv(\sigma'_1))} \langle u = \omega \rangle)$, and $1 \notin ann^+((\sigma_1 \rightarrow \sigma_2)^\kappa)$, $1 \notin ann^+(\sigma'_1)$. Now since both substitutions agree on $fuv(\Gamma)$, and (by induction) $\text{dom}(S_i) \setminus fuv(\Gamma)$ is fresh, we can form $S = S_1 \cup S_2$, and this satisfies C_1 and C_2 . Assume (for contradiction) that it does not satisfy C_3 ; that is,

$S\sigma'_1 \not\leq S\sigma_1$. Then either at a positive position $S\sigma'_1$ bears the annotation 1 and $S\sigma_1$ bears the annotation ω , or at a negative position $S\sigma'_1$ bears the annotation ω and $S\sigma_1$ bears the annotation 1. The former is not possible because all free variables of σ'_1 are mapped to ω and $1 \notin \text{ann}^+(\sigma'_1)$. The latter is also not possible because all free variables of σ_1 are mapped to ω and $1 \notin \text{ann}^-(\sigma_1) \subseteq \text{ann}^+(\sigma_1 \rightarrow \sigma_2)^\kappa$. Thus we have a contradiction and the result follows. That $1 \notin \text{ann}^+(\sigma_2)$ also follows.

case $\Lambda\alpha . M$

(i) By (\blacktriangleright_2 -TYABS), (\vdash_0 -TYABS), and I.H.

(iv) By I.H. we have that for all S such that $\vdash^e SC, ST, \alpha \vdash_2 Se : S(\tau^\kappa)$. Hence by (\vdash_2 -TYABS) we may derive $ST \vdash_2 S(\Lambda\alpha . e) : S((\forall\alpha . \tau)^\kappa)$ as required.

(v) By I.H. and inspection.

case $M t$

As case $A t$ of case A above.

case case $M : T \overline{t_k}$ of $\overline{K_i \overline{x_{ij}} \rightarrow \overline{M_i}}$

(i) (\blacktriangleright_2 -CASE) applies. By (\vdash_0 -CASE) we have $\Gamma \vdash_0 M : T \overline{t_k}$ and $\Gamma, \overline{x_{ij} : t_{ij}^\circ} \vdash_0 M_i : t$ where $t_{ij}^\circ = t_{ij}[\overline{t_k}/\overline{\alpha_k}]$, enabling us to apply the I.H. By I.H.(ii) we have that the type returned from M is of the right shape, and that $(\sigma_i)^\natural = t$. This latter means that the types are compatible in the *FreshLUB* of C_3 . Thus (\blacktriangleright_2 -CASE) is deterministic and does not fail.

(iv) Let S be a solution of $C_1 \wedge C_2 \wedge C_3 \wedge C_4$. By I.H. we have that $ST \vdash_2 Se : S((T \overline{\kappa_l} \overline{\tau_k})^\kappa)$, and also $ST \vdash_2 Se_i : S\sigma_i$. Furthermore, $C_3 \vdash^e S\sigma_i \leq \sigma$, and $C_4 \vdash^e \{V_i(x_{ij}) > 1 \Rightarrow \langle S|\sigma_{ij}^\circ| = \omega \rangle\}$. By (\vdash_2 -SUB) we may derive $ST \vdash_2 Se_i : S\sigma$, and by (\vdash_2 -CASE) we may obtain the desired result.

(v) By I.H., we have S_1 such that $\vdash^e S_1(C_1 \wedge \bigwedge_{u \in (\text{fuv}(\Gamma) \cup \text{fuv}((T \overline{\kappa_l} \overline{\tau_k})^\kappa))} \langle u = \omega \rangle)$ and S_2^i such that $\vdash^e S_2^i(C_2^i \wedge \bigwedge_{u \in (\text{fuv}(\Gamma, \overline{x_{ij} : \sigma_{ij}^\circ}) \cup \text{fuv}(\sigma_i))} \langle u = \omega \rangle)$. Observe

that these agree on $\text{fuv}(\Gamma)$, and also on $\text{fuv}(\overline{\sigma_{ij}^\circ})$, and on $\text{fuv}((T \overline{\kappa_l} \overline{\tau_k})^\kappa)$; remaining variables are fresh by induction. So we may form $S = S_1 \cup \bigcup_i S_2^i \cup \{\text{fuv}(\sigma) \mapsto \omega\}$, and this satisfies C_1 and C_2 . Furthermore, it satisfies C_3 : if it did not, then there is an i and either a positive position in which σ_i bears a 1 annotation and σ a ω , or a negative position in which σ_i bears a ω annotation and σ a 1. The former case cannot happen since $\text{fuv}(\sigma_i) \mapsto \omega$ and $1 \notin \text{ann}^+(\sigma_i)$; in the latter case, by definition of *FreshLUB*, if a negative position in σ bears a 1 annotation then all the σ_i must have 1 there also, a contradiction. Finally, C_4 follows from the fact that $|\sigma_{ij}^\circ|$ is a positive annotation, and by definition of $\text{fuv}^\varepsilon(\cdot)$, it must come from a positive annotation in $(T \overline{\kappa_l} \overline{\tau_k})^\kappa$, and all these are mapped to ω by S . The result follows.

case $M_1 + M_2$

(i) (\blacktriangleright_2 -PRIMOP) applies. By (\vdash_0 -PRIMOP) we have $\Gamma \vdash_0 M_i : \text{Int}$, enabling us to apply the I.H. By I.H.(ii) we have that $(\sigma_i)^\natural = \text{Int}$, and so (\blacktriangleright_2 -PRIMOP) is deterministic and does not fail.

(iv) Let S be a solution of $C_1 \wedge C_2$. Then by I.H., $S\Gamma \vdash_2 SM_i : S(\text{Int}^{\kappa_i})$. Thus by (\vdash_2 -SUB) and (\vdash_2 -PRIMOP), the result follows.

(v) By I.H., we have S_i such that $\vdash^e S(C_i \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv(\text{Int}^{\kappa_i}))} \langle u = \omega \rangle)$, and $1 \notin \text{ann}^+(\text{Int}^{\kappa_i})$. Now since both substitutions agree on $fuv(\Gamma)$, and (by induction) $\text{dom}(S_i) \setminus fuv(\Gamma)$ is fresh, we can form $S = S_1 \cup S_2$, and this satisfies C_1 and C_2 . By inspection, the result follows.

case if0 M then M_1 else M_2

Analogous to case case (but simpler). In (iv), we must insert a (\vdash_2 -SUB) above e as well as the two for the e_i justifying the *FreshLUB*; this is because if0 requires the condition to have type Int^1 , whereas case does not restrict the annotation on its scrutinee.

case letrec $\overline{x_i : t_i} = \overline{M_i}$ in M

(i) (\blacktriangleright_2 -LETREC) applies. By (\vdash_0 -LETREC) we have $\Gamma, \overline{x_j : t_j} \vdash_0 M_i : t_i$ and $\Gamma, \overline{x_j : t_j} \vdash_0 M : t$, enabling us to apply the I.H. By I.H.(ii) we have that $(\sigma'_i)^\sharp = t_i = (\lceil t_i \rceil_\sigma^{\text{fresh}})^\sharp$, and so the types are compatible in the subtyping constraint of C_1 . By I.H.(v) the C_1^i are satisfiable and agree on $fuv(\Gamma, x_j : \tau_j^{v_j})$, and so we may combine the substitutions; letting $fuv(\tau_i^{v_i}) \mapsto \omega$ means that the substitution also entails $\sigma'_i \preceq \tau_i^{v_i}$, for if not, there is a positive annotation of σ_i that is 1, a contradiction. Hence C_1 is satisfiable, and by Lemma D.13 the *Closoperation* is well-defined. Now by Lemma D.13(vii) we also have that $1 \notin fuv^+(S\tau_j)$ and hence $1 \notin \text{ann}^+(\overline{(\forall \overline{u_k} . S\tau_j)^{v_j}})$. So by I.H., the body inference is well-defined, and so (\blacktriangleright_2 -LETREC) is deterministic and does not fail.

(iv) Let S' be a solution of $C'_1 \wedge C_2 \wedge C_3$. By Lemma D.13(ii) and I.H. we have that $S'\Gamma, x_j : S'S\tau_j^{S'Sv_j} \vdash_2 S'Se_i : S'S\sigma'_i$ (note from Lemma D.13(v) that S does not scope over Γ). Now $C'_1 \vdash^e S\sigma'_i \preceq S\tau_i^{Sv_i}$, and so by (\vdash_2 -SUB) we have $S'\Gamma, x_j : S'S\tau_j^{S'Sv_j} \vdash_2 S'Se_i : S'S\tau_i^{S'Sv_i}$. Now by Lemma D.13(vi) we may apply (\vdash_2 -UABS), deriving $S'\Gamma, x_j : S'S\tau_j^{S'Sv_j} \vdash_2 S'\Lambda \overline{u_k} . Se_i : S'\forall \overline{u_k} . S\tau_i^{S'Sv_i}$, and by weakening (lemma omitted) and Lemma D.5 we have $S'\Gamma, y_j : (S'\forall \overline{u_k} . S\tau_j)^{S'Sv_j} \vdash_2 S'\Lambda \overline{u_k} . Se_i[y_j \overline{u_k}/x_j] : S'\forall \overline{u_k} . S\tau_i^{S'Sv_i}$. The result follows by (\vdash_2 -LETREC).

(v) We have already shown in (i) that C_1 is satisfiable; by Lemma D.13(iii) this shows that C'_1 is satisfiable; by (v) we have that $fuv(\Gamma)$ may be mapped to ω as required, as can $\overline{v_i}$. This substitution is S'_1 . Again, in (i) above we have shown that $1 \notin \text{ann}^+(\overline{(\forall \overline{u_k} . S\tau_j)^{v_i}})$. Thus by I.H. we may obtain S_2 satisfying C_2 and mapping the appropriate variables to ω . Since $fuv(\overline{(\forall \overline{u_k} . S\tau_j)^{v_i}}) \subseteq fuv(\tau_j^{v_j})$, it is clear that S'_1 and S_2 agree on all variables they have in common, and so we may combine them. Finally, this substitution trivially satisfies C_3 . The result follows. \square

We now prove a lemma about subtyping.

Lemma D.15 (Tycon subtyping)

Given a type ψ (either a τ - or a σ -type), two vectors of usage annotations $\overline{\kappa_l}$ and $\overline{\kappa'_l}$,

and two vectors of τ -types $\overline{\tau}_k$ and $\overline{\tau}'_k$ such that

$$\begin{aligned} u_l \in fuv^\varepsilon(\psi) &\Rightarrow \kappa_l \leq^\varepsilon \kappa'_l \quad \text{for all } \varepsilon, l \\ \alpha_k \in ftv^\varepsilon(\psi) &\Rightarrow \tau_k \preceq^\varepsilon \tau'_k \quad \text{for all } \varepsilon, k \end{aligned}$$

we have that $\psi[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq \psi[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$.

Proof The proof is by induction on the structure of ψ .

case $\psi = \sigma_1 \rightarrow \sigma_2$

We have by definition of $ftv^\varepsilon(\cdot)$ and assumptions, that $\alpha_k \in ftv^\varepsilon(\sigma_1) \Rightarrow \alpha_k \in ftv^\varepsilon(\sigma_1 \rightarrow \sigma_2) \Rightarrow \tau'_k \preceq^\varepsilon \tau_k$ and $\alpha_k \in ftv^\varepsilon(\sigma_2) \Rightarrow \alpha_k \in ftv^\varepsilon(\sigma_1 \rightarrow \sigma_2) \Rightarrow \tau_k \preceq^\varepsilon \tau'_k$ for all ε and k , and similarly for usage. Thus, by induction we have that $\sigma_1[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k] \preceq \sigma_1[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k]$ and $\sigma_2[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq \sigma_2[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$. Hence, by (\preceq -ARROW) we have that $(\sigma_1 \rightarrow \sigma_2)[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq (\sigma_1 \rightarrow \sigma_2)[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$, as required.

case $\psi = \forall \alpha' . \tau$

Assume (without loss of generality) that α' is distinct from all the α_k . We have that $\alpha_k \in ftv^\varepsilon(\tau) \Rightarrow \alpha_k \in ftv^\varepsilon(\forall \alpha' . \tau) \Rightarrow \tau_k \preceq^\varepsilon \tau'_k$ for all ε and all k , and similarly for usage. Thus by induction, $\tau[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq \tau[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$, and so by (\preceq -FORALL) we have $(\forall \alpha' . \tau)[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq (\forall \alpha' . \tau)[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$, as required.

case $\psi = \forall u' . \tau$

Similar to the $\forall \alpha' . \tau$ case.

case $\psi = \alpha$

If $\alpha \neq \alpha_k$, all k , then the result is trivial. If $\alpha = \alpha_k$ for some k , then clearly $\alpha \in ftv^+(\alpha)$, and thus $\tau_k \preceq \tau'_k$ as required.

case $\psi = \tau^u$

We have that $\alpha_k \in ftv^\varepsilon(\tau) \Rightarrow \alpha_k \in ftv^\varepsilon(\tau^u) \Rightarrow \tau_k \preceq^\varepsilon \tau'_k$ for all ε, k , and similarly for usage. Thus by induction, $\tau[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq \tau[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$. If $u \neq u_l$, all l , then trivially $u = u$. If $u = u_l$ for some l , then clearly $u \in fuv^+(u)$, and thus $\kappa_l \leq \kappa'_l$. In either case, $u[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq u[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$, and so by (\preceq -ANNOT) we have $(\tau^u)[\overline{\kappa}_l/\overline{u}_l, \overline{\tau}_k/\overline{\alpha}_k] \preceq (\tau^u)[\overline{\kappa}'_l/\overline{u}_l, \overline{\tau}'_k/\overline{\alpha}_k]$, as required.

case $\psi = T \overline{\kappa}_{l'}^\diamond \overline{\tau}_{k'}^\diamond$ where $\text{data } (T \overline{u}_{l'}^\diamond \overline{\alpha}_{k'}^\diamond)^{u^\diamond} = \overline{K_i} \overline{\sigma_{ij}^\diamond}$

For each k' , we have that if $\alpha_{k'}^\diamond \in ftv^{\varepsilon'}(\sigma_{ij}^\diamond)$, then

- for each k , $\alpha_k \in ftv^\varepsilon(\tau_{k'}^\diamond) \Rightarrow \alpha_k \in ftv^{\varepsilon \cdot \varepsilon'}(T \overline{\kappa}_{l'}^\diamond \overline{\tau}_{k'}^\diamond)$ by definition of $ftv^\varepsilon(\cdot)$, and by the assumptions this in turn implies $\tau_k \preceq^{\varepsilon \cdot \varepsilon'} \tau'_k$.
- for each l , $u_l \in fuv^\varepsilon(\tau_{k'}^\diamond) \Rightarrow u_l \in fuv^{\varepsilon \cdot \varepsilon'}(T \overline{\kappa}_{l'}^\diamond \overline{\tau}_{k'}^\diamond)$ by definition of $fuv^\varepsilon(\cdot)$, and by the assumptions this in turn implies $\kappa_l \leq^{\varepsilon \cdot \varepsilon'} \kappa'_l$.

Furthermore, for each l' , we have that if $u_{l'}^\diamond \in fw^{\varepsilon'}(\sigma_{ij}^\diamond)$, then

- for each l , $u_l \in fw^\varepsilon(\kappa_{l'}^\diamond) \Rightarrow u_l \in fw^{\varepsilon \cdot \varepsilon'}(T \overline{\kappa_{l'}^\diamond} \overline{\tau_{k'}^\diamond})$ by definition of $fw^\varepsilon(\cdot)$, and by the assumptions this in turn implies $\kappa_l \leq^{\varepsilon \cdot \varepsilon'} \kappa_{l'}'$.
- for each k , $\alpha_k \notin ftv^\varepsilon(\kappa_{l'}^\diamond) = \emptyset$.

Hence by induction we have

- $\alpha_{k'}^\diamond \in ftv^{\varepsilon'}(\sigma_{ij}^\diamond) \Rightarrow \tau_{k'}^\diamond[\overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \preceq^{\varepsilon'} \tau_{k'}^\diamond[\overline{\kappa_{l'}}/\overline{u_l}, \overline{\tau_{k'}}/\overline{\alpha_k}]$ for all ε' , k' , and
- $u_{l'}^\diamond \in fw^{\varepsilon'}(\sigma_{ij}^\diamond) \Rightarrow \kappa_{l'}^\diamond[\overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \leq^{\varepsilon'} \kappa_{l'}^\diamond[\overline{\kappa_{l'}}/\overline{u_l}, \overline{\tau_{k'}}/\overline{\alpha_k}]$ for all ε' , l' .

Finally by (\preceq -TYCON) we have $T \overline{\kappa_{l'}^\diamond[\overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}]} \overline{\tau_{k'}^\diamond[\overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}]} \preceq T \overline{\kappa_{l'}^\diamond[\overline{\kappa_{l'}}/\overline{u_l}, \overline{\tau_{k'}}/\overline{\alpha_k}]} \overline{\tau_{k'}^\diamond[\overline{\kappa_{l'}}/\overline{u_l}, \overline{\tau_{k'}}/\overline{\alpha_k}]}$ and thus $(T \overline{\kappa_{l'}^\diamond} \overline{\tau_{k'}^\diamond})[\overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \preceq (T \overline{\kappa_{l'}^\diamond} \overline{\tau_{k'}^\diamond})[\overline{\kappa_{l'}}/\overline{u_l}, \overline{\tau_{k'}}/\overline{\alpha_k}]$ as required. \square

We now prove completeness of inference phase 1 for $FLIX_1$ only (the result does not hold for $FLIX_2$). The first lemma and proof are printed in landscape format in order to accommodate the exhibited proof tree fragments.

Lemma D.16 (Canonical proof tree)

If $\Gamma \vdash_2 e : \sigma$ then there exists a proof tree whose conclusion is $\Gamma \vdash_2 e : \sigma$ (except that in case expressions the type annotation of the scrutinee may be replaced by a more specific type) and in which the rule $(\vdash_2\text{-SUB})$ occurs in each of the locations described in Sections 3.5.2 and 5.5.1 and listed below, and nowhere else:

- immediately above $(\vdash_2\text{-IF0})$ in any branch,
- immediately above $(\vdash_2\text{-PRIMOP})$ and $(\vdash_2\text{-PRIMOP-R})$ in any branch,
- immediately above $(\vdash_2\text{-APP})$ in branch e but restricted to the topmost annotation only (i.e., $\sigma' = \tau^{k'} \preceq \tau^k = \sigma$),
- immediately above $(\vdash_2\text{-APP})$ in branch a ,
- immediately above $(\vdash_2\text{-LETREC})$ in branches \bar{e}_i (not branch e),
- immediately above $(\vdash_2\text{-CON})$ in any branch,
- immediately above $(\vdash_2\text{-CASE})$ in branches \bar{e}_i (not branch e), and
- at the root.

Proof Since the proof tree is finite, it must contain only finitely many occurrences of $(\vdash_2\text{-SUB})$, and have only finite height. Thus we may take each occurrence in turn and push it downwards through each rule until it reaches one of the positions listed above, or it reaches another occurrence of $(\vdash_2\text{-SUB})$. If we reach another occurrence of $(\vdash_2\text{-SUB})$ we may merge the two, by transitivity of \preceq , and continue pushing the rule downwards. Once all occurrences of $(\vdash_2\text{-SUB})$ are at one of the locations listed above, we may insert trivial $(\vdash_2\text{-SUB})$ instances at any locations that do not have a $(\vdash_2\text{-SUB})$ occurrence, by reflexivity of \preceq , and we are done.

We demonstrate that $(\vdash_2\text{-SUB})$ may be pushed downwards through all rules and positions not listed above.

case $(\vdash_2\text{-ABS})$

We have that

$$\frac{\frac{\Gamma, x : \sigma_1 \vdash_2 e : \sigma'_2 \quad \sigma'_2 \preceq \sigma_2}{\Gamma, x : \sigma_1 \vdash_2 e : \sigma_2} (\vdash_2\text{-SUB}) \quad \frac{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa \quad \text{for all } y \in \Gamma \quad \begin{array}{l} \text{occure}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \text{occure}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \end{array}}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-ABS})}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-ABS})$$

Now by $(\preceq\text{-ANNOT})$ and $(\preceq\text{-ARROW})$, if $\sigma'_2 \preceq \sigma_2$ then $(\sigma_1 \rightarrow \sigma'_2)^\kappa \preceq (\sigma_1 \rightarrow \sigma_2)^\kappa$. Hence we may derive

$$\frac{\frac{\Gamma, x : \sigma_1 \vdash_2 e : \sigma'_2 \quad \begin{array}{l} \text{occure}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \text{occure}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \end{array} \quad \text{for all } y \in \Gamma}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma'_2)^\kappa} (\vdash_2\text{-ABS}) \quad \frac{(\sigma_1 \rightarrow \sigma'_2)^\kappa \preceq (\sigma_1 \rightarrow \sigma_2)^\kappa}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-SUB})}{\Gamma \vdash_2 \lambda^{\kappa, \kappa^\dagger} x : \sigma_1 . e : (\sigma_1 \rightarrow \sigma_2)^\kappa} (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-APP})$, branch e

We have that

$$\frac{\frac{\Gamma \vdash_2 e : (\sigma'_1 \rightarrow \sigma'_2)^{\kappa'} \quad (\sigma'_1 \rightarrow \sigma'_2)^{\kappa'} \preceq (\sigma_1 \rightarrow \sigma_2)^1}{\Gamma \vdash_2 e : (\sigma_1 \rightarrow \sigma_2)^1} (\vdash_2\text{-SUB}) \quad \Gamma \vdash_2 a : \sigma_1}{\Gamma \vdash_2 e a : \sigma_2} (\vdash_2\text{-APP})$$

Now by $(\preceq\text{-ANNOT})$ and $(\preceq\text{-ARROW})$, if $(\sigma'_1 \rightarrow \sigma'_2)^{\kappa'} \preceq (\sigma_1 \rightarrow \sigma_2)^1$ then $\kappa' \leq 1$, $\sigma_1 \preceq \sigma'_1$, and $\sigma'_2 \preceq \sigma_2$. Hence we may derive

$$\frac{\frac{\Gamma \vdash_2 e : (\sigma'_1 \rightarrow \sigma'_2)^{\kappa'} \quad (\sigma'_1 \rightarrow \sigma'_2)^{\kappa'} \preceq (\sigma'_1 \rightarrow \sigma'_2)^1}{\Gamma \vdash_2 e : (\sigma'_1 \rightarrow \sigma'_2)^1} (\vdash_2\text{-SUB}) \quad \frac{\Gamma \vdash_2 a : \sigma_1 \quad \sigma_1 \preceq \sigma'_1}{\Gamma \vdash_2 a : \sigma'_1} (\vdash_2\text{-SUB})}{\frac{\Gamma \vdash_2 e : (\sigma'_1 \rightarrow \sigma'_2)^1 \quad \Gamma \vdash_2 a : \sigma'_1}{\Gamma \vdash_2 e a : \sigma'_2} (\vdash_2\text{-APP}) \quad \sigma'_2 \preceq \sigma_2}{\Gamma \vdash_2 e a : \sigma_2} (\vdash_2\text{-SUB})$$

as required. If the a branch already has a $(\vdash_2\text{-SUB})$ instance at its root then we should merge this with the new instance; otherwise we should insert the new instance just as shown. In either case, this is a legal position to leave a $(\vdash_2\text{-SUB})$ instance, and so since it will be pushed downwards no further, it does not affect the termination result.

case $(\vdash_2\text{-LETREC})$, branch e

We have that

$$\frac{\Gamma, \overline{\sigma_j} : \sigma_j \vdash_2 e_i : \sigma_i \quad \text{for all } i}{\left(\text{occure}(x_i, e) + \sum_{j=1}^n \text{occure}(x_i, e_j) \right) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i} \frac{\Gamma, \overline{\sigma_j} : \sigma_j \vdash_2 e : \sigma' \quad \sigma' \preceq \sigma}{\Gamma \vdash_2 \text{letrec } x_i : \sigma_i = |\sigma_i|^\dagger e_i \text{ in } e : \sigma} (\vdash_2\text{-LETREC})$$

We may derive

$$\frac{\begin{array}{l} \Gamma, \overline{\sigma_j} : \sigma_j \vdash_2 e_i : \sigma_i \quad \text{for all } i \\ \Gamma, \overline{\sigma_j} : \sigma_j \vdash_2 e : \sigma' \\ \left(\text{occure}(x_i, e) + \sum_{j=1}^n \text{occure}(x_i, e_j) \right) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \vdash_2 \text{letrec } x_i : \sigma_i = |\sigma_i|^\dagger e_i \text{ in } e : \sigma'} (\vdash_2\text{-LETREC}) \quad \sigma' \preceq \sigma \quad (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-UABS})$

We have that

$$\frac{\Gamma, u \vdash_2 e : \tau'^{\kappa'} \quad \tau'^{\kappa'} \preceq \tau^\kappa}{\Gamma, u \vdash_2 e : \tau^\kappa} (\vdash_2\text{-SUB}) \quad \frac{u \notin (fuw(\Gamma) \cup fuw(\kappa))}{\Gamma \vdash_2 \Lambda u . e : (\forall u . \tau)^\kappa} (\vdash_2\text{-UABS})$$

Now by $(\preceq\text{-ANNOT})$ and $(\preceq\text{-ALL-U})$, if $\tau'^{\kappa'} \preceq \tau^\kappa$ then $(\forall u . \tau')^{\kappa'} \preceq (\forall u . \tau)^\kappa$. Furthermore, if $u \notin fuw(\kappa)$ and $\kappa' \leq \kappa$

then $u \notin fw(\kappa')$. Hence we may derive

$$\frac{\frac{\Gamma, u \vdash_2 e : \tau'^{\kappa'} \quad u \notin (fw(\Gamma) \cup fw(\kappa'))}{\Gamma \vdash_2 \Lambda u . e : (\forall u . \tau')^{\kappa'}} (\vdash_2\text{-UABS})}{\Gamma \vdash_2 \Lambda u . e : (\forall u . \tau)^{\kappa}} (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-UAPP})$

We have that

$$\frac{\frac{\Gamma \vdash_2 e : (\forall u . \tau')^{\kappa''} \quad (\forall u . \tau')^{\kappa''} \preceq (\forall u . \tau)^{\kappa}}{\Gamma \vdash_2 e : (\forall u . \tau)^{\kappa}} (\vdash_2\text{-SUB})}{\Gamma \vdash_2 e \kappa' : (\tau[\kappa'/u])^{\kappa}} (\vdash_2\text{-UAPP})$$

Now by $(\preceq\text{-ANNOT})$ and $(\preceq\text{-ALL-U})$, if $(\forall u . \tau')^{\kappa''} \preceq (\forall u . \tau)^{\kappa}$ then $\tau' \preceq \tau$ and $\kappa'' \leq \kappa$. By definition of \preceq and \leq , then, $\tau'[\kappa'/u] \preceq \tau[\kappa'/u]$, since at each occurrence of u in τ (respectively τ') the other has either u also or the constant ω (1), making the relation hold. Hence we may derive

$$\frac{\frac{\Gamma \vdash_2 e : (\forall u . \tau')^{\kappa''}}{\Gamma \vdash_2 e \kappa' : (\tau'[\kappa'/u])^{\kappa''}} (\vdash_2\text{-UAPP})}{\Gamma \vdash_2 e \kappa' : (\tau[\kappa'/u])^{\kappa}} (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-TYABS})$

We have that

$$\frac{\frac{\Gamma, \alpha \vdash_2 e : \tau'^{\kappa'} \quad \tau'^{\kappa'} \preceq \tau^{\kappa}}{\Gamma, \alpha \vdash_2 e : \tau^{\kappa}} (\vdash_2\text{-SUB})}{\Gamma \vdash_2 \Lambda \alpha . e : (\forall \alpha . \tau)^{\kappa}} (\vdash_2\text{-TYABS})$$

Now by $(\preceq\text{-ANNOT})$ and $(\preceq\text{-ALL})$, if $\tau'^{\kappa'} \preceq \tau^\kappa$ then $(\forall\alpha . \tau')^{\kappa'} \preceq (\forall\alpha . \tau)^\kappa$. Hence we may derive

$$\frac{\frac{\Gamma, \alpha \vdash_2 e : \tau'^{\kappa'} \quad \alpha \notin \text{fv}(\Gamma)}{\Gamma \vdash_2 \Lambda\alpha . e : (\forall\alpha . \tau')^{\kappa'}} \quad (\vdash_2\text{-TYABS}) \quad (\forall\alpha . \tau')^{\kappa'} \preceq (\forall\alpha . \tau)^\kappa}{\Gamma \vdash_2 \Lambda\alpha . e : (\forall\alpha . \tau)^\kappa} (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-TYAPP})$

We have that

$$\frac{\frac{\Gamma \vdash_2 e : (\forall\alpha . \tau_1')^{\kappa'} \quad (\forall\alpha . \tau_1')^{\kappa'} \preceq (\forall\alpha . \tau_1)^\kappa}{\Gamma \vdash_2 e : (\forall\alpha . \tau_1)^\kappa} \quad (\vdash_2\text{-SUB})}{\Gamma \vdash_2 e \tau_2 : (\tau_1[\tau_2/\alpha])^\kappa} (\vdash_2\text{-TYAPP})$$

Now by $(\preceq\text{-ANNOT})$, $(\preceq\text{-ALL})$, $(\preceq\text{-TYVAR})$, and reflexivity of \preceq , if $(\forall\alpha . \tau_1')^{\kappa'} \preceq (\forall\alpha . \tau_1)^\kappa$ then we have $(\tau_1'[\tau_2/\alpha])^{\kappa'} \preceq (\tau_1[\tau_2/\alpha])^\kappa$. Hence we may derive

$$\frac{\frac{\Gamma \vdash_2 e : (\forall\alpha . \tau_1')^{\kappa'}}{\Gamma \vdash_2 e \tau_2 : (\tau_1'[\tau_2/\alpha])^{\kappa'}} \quad (\vdash_2\text{-TYAPP}) \quad (\tau_1'[\tau_2/\alpha])^{\kappa'} \preceq (\tau_1[\tau_2/\alpha])^\kappa}{\Gamma \vdash_2 e \tau_2 : (\tau_1[\tau_2/\alpha])^\kappa} (\vdash_2\text{-SUB})$$

as required.

case $(\vdash_2\text{-CASE})$, branch e

We have that

$$\frac{\frac{\Gamma \vdash_2 e : (T \overline{\kappa_l'} \overline{\tau_k'})^{\kappa'} \quad (T \overline{\kappa_l'} \overline{\tau_k'})^{\kappa'} \preceq (T \overline{\kappa_l} \overline{\tau_k})^\kappa}{\Gamma \vdash_2 e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa} \quad (\vdash_2\text{-SUB}) \quad \frac{\begin{array}{l} \sigma_{ij}^\circ = \sigma_{ij}[\kappa'/u, \overline{\kappa_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \quad \text{all } i, j \\ \Gamma, x_{ij} : \sigma_{ij}^\circ \vdash_2 e_i : \sigma \quad \text{all } i \\ \text{occurs}(x_{ij}, e_i) > 1 \Rightarrow |\sigma_{ij}^\circ| = \omega \quad \text{all } i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i} \overline{\sigma_{ij}} \end{array}}{\Gamma \vdash_2 \text{case } e : (T \overline{\kappa_l} \overline{\tau_k})^\kappa \text{ of } \overline{K_i} \overline{x_{ij}} \rightarrow e_i : \sigma} (\vdash_2\text{-CASE})$$

Now by $(\preceq\text{-TYCON})$ and $(\preceq\text{-ANNOT})$, if $(T \overline{\kappa'_l} \overline{\tau'_k})^{\kappa'} \preceq (T \overline{\kappa_l} \overline{\tau_k})^\kappa$ then

$$\begin{aligned} \alpha_k \in \text{fv}^{\varepsilon}(\sigma_{ij}) &\Rightarrow \tau'_k \preceq^{\varepsilon} \tau_k && \text{for all } k, \varepsilon, i, j \\ u_l \in \text{fv}^{\varepsilon}(\sigma_{ij}) &\Rightarrow \kappa'_l \leq^{\varepsilon} \kappa_l && \text{for all } l, \varepsilon, i, j \\ \kappa' &\leq \kappa \end{aligned}$$

Since u occurs only positively in σ_{ij} (by definition; see Sections 5.3.1 and 5.3.4.3), we may apply Lemma D.15 to each type σ_{ij} and the vector pairs κ', κ'_l and κ, κ_l , and τ'_k and τ_k , yielding

$$\sigma_{ij}^{\circ} = \sigma_{ij}[\kappa'_l/u, \overline{\kappa'_l/\overline{u_l}}, \overline{\tau'_k/\overline{\alpha_k}}] \preceq \sigma_{ij}[\kappa_l/u, \overline{\kappa_l/\overline{u_l}}, \overline{\tau_k/\overline{\alpha_k}}] = \sigma_{ij}^{\circ}$$

If we can now establish $\Gamma, \overline{x_{ij}} : \sigma_{ij}^{\circ} \vdash_2 e_i : \sigma$ for all i , then we are done, for since by $(\preceq\text{-ANNOT})$ we have $|\sigma_{ij}^{\circ}| = \omega \Rightarrow |\sigma_{ij}^{\circ}| = \omega$, we have

$$\begin{aligned} \Gamma \vdash_2 e : (T \overline{\kappa'_l} \overline{\tau'_k})^{\kappa'} & \\ \sigma_{ij}^{\circ} = \sigma_{ij}[\kappa'_l/u, \overline{\kappa'_l/\overline{u_l}}, \overline{\tau'_k/\overline{\alpha_k}}] & \quad \overline{\tau'_k/\overline{\alpha_k}} \quad \text{all } i, j \\ \Gamma, x_{ij} : \sigma_{ij}^{\circ} \vdash_2 e_i : \sigma & \quad \text{all } i \\ \text{occurs}(x_{ij}, e_i) > 1 \Rightarrow |\sigma_{ij}^{\circ}| = \omega & \quad \text{all } i, j \\ \text{where data } (T \overline{u_l} \overline{\alpha_k})^u = \overline{K_i \overline{\sigma_{ij}}} & \\ \Gamma \vdash_2 \text{case } e : (T \overline{\kappa'_l} \overline{\tau'_k})^{\kappa'} \text{ of } \overline{K_i \overline{x_{ij}} \rightarrow e_i} : \sigma & \quad (\vdash_2\text{-CASE}) \end{aligned}$$

as required (notice that the type annotation on e is now more specific). We may establish the required typing by, at every occurrence of the $\overline{x_{ij}}$, replacing the rule instance

$$\frac{}{\Gamma', \overline{x_{ij}} : \sigma_{ij}^{\circ} \vdash_2 x_{ij} : \sigma_{ij}^{\circ}} (\vdash_2\text{-VAR})$$

with

$$\frac{\frac{\Gamma', \overline{x_{ij}} : \sigma_{ij}^{\circ} \vdash_2 x_{ij} : \sigma_{ij}^{\circ}}{\Gamma', \overline{x_{ij}} : \sigma_{ij}^{\circ} \vdash_2 x_{ij} : \sigma_{ij}^{\circ}} (\vdash_2\text{-VAR})}{\Gamma', \overline{x_{ij}} : \sigma_{ij}^{\circ} \vdash_2 x_{ij} : \sigma_{ij}^{\circ}} (\vdash_2\text{-SUB})$$

Proving termination is now a little trickier. The essential observation is that when a $(\vdash_2\text{-SUB})$ instance is pushed down from the *scrutinee* branch, new $(\vdash_2\text{-SUB})$ instances are inserted into the case *alternative* branches, and the furthest they may be pushed downwards is to just above this $(\vdash_2\text{-CASE})$ instance. Thus no recursive insertion can occur, and the process will eventually terminate.

□

Theorem D.17 (Soundness and completeness of monomorphic inference phase 1)

For all Γ in $FLIX_1$ (necessarily without free usage variables) and M, t in FL_0 such that $(\Gamma)^\sharp \vdash_0 M : t$ and $1 \notin ann^+(\Gamma)$, and all annotated data type declarations satisfying $1 \notin ann(\overline{\sigma_{ij}})$ (see Section 5.4.3),

- (i) $\blacktriangleright_1 (\Gamma, M) = (e', \sigma', C, V)$ is well defined³
(i.e., the algorithm \blacktriangleright_1 is deterministic and does not fail);
- (ii) $(e')^\sharp = M$ and $(\sigma')^\sharp = t$
(i.e., the inference algorithm merely annotates the source term, and does not alter it or its source type);
- (iii) $\forall x \in \Gamma. V(x) = occur(x, e')$
(i.e., V correctly implements the *occur* function);
- (iv) $\forall S. \vdash^e SC \Rightarrow S\Gamma \vdash_1 Se' : S\sigma'$
(i.e., all solutions of the resulting constraint are well-typed); and
- (v) $\exists S. \vdash^e S(C \wedge \bigwedge_{u \in (fuv(\Gamma) \cup fuv(\sigma'))} \langle u = \omega \rangle)$ and $1 \notin ann^+(\sigma')$
(i.e., the resulting constraint has at least one solution, and permits all possible future uses).
- (vi) For all e, σ such that $(e)^\sharp = M$, $(\sigma)^\sharp = t$, and $\Gamma \vdash_1 e : \sigma$, there exists a substitution S such that $\vdash^e SC$ and $Se' = e$, $S\sigma' = \sigma$.
(i.e., all well-typed annotations of the source term are solutions of the resulting constraint).

Proof Proofs of (i) through (v) are essentially the same as those of Theorem D.14 for $FLIX_2$, except in the variable and letrec cases, which are similar but simpler. Completeness, (vi), holds only for $FLIX_1$.

Consider the proof tree of $S\Gamma \vdash_1 Se' : S\sigma'$ constructed by the proof of (iv) (in fact, since $fuv(\Gamma) = \emptyset$, $S\Gamma = \Gamma$, and so it is a proof tree of $\Gamma \vdash_1 Se' : S\sigma'$). In the case of $FLIX_1$, this will contain no occurrences of $(\vdash_2\text{-UABS})$ or $(\vdash_2\text{-UAPP})$, since $FLIX_1$ has no usage polymorphism. Now observe that this proof tree is in the canonical form defined in Lemma D.16: occurrences of $(\vdash_1\text{-SUB})$ are restricted to certain specific locations.

We know that $\Gamma \vdash_1 e : \sigma$; thus we may apply Lemma D.16 to obtain a canonical proof tree for this result. This proof tree will be of exactly the same shape as that constructed by the proof of (iv) above, and thus of the same shape as the inference proof tree of $\Gamma \blacktriangleright_1 M \rightsquigarrow e : \sigma; C; V$. That is, to each step of the inference algorithm corresponds a non- $(\vdash_1\text{-SUB})$ node of the canonical proof tree, possibly with instances of $(\vdash_1\text{-SUB})$ immediately above.

Now we may proceed by induction on the structure of these isomorphic proof trees, that of $\Gamma \blacktriangleright_1 M \rightsquigarrow e : \sigma; C; V$ on the left and that of $\Gamma \vdash_1 e : \sigma$ on

³It is well defined modulo the names of fresh variables; we have already noted that we are omitting the details of fresh variable management. Here this also means that the list of usage variables in scope occurs in Γ of $\Gamma \vdash_1 e : \sigma$ but not in Γ of $\blacktriangleright_1 (\Gamma, M)$, since in the latter the free usage variables are managed separately.

the right. At each node we may augment the substitution so as to yield a proof subtree from (iv) that is identical to the canonical proof subtree on the right; this is possible because for each rule the conditions on the right are sufficient to satisfy the constraints on the left, and fresh variables are always sufficiently fresh that substitutions never interfere. The final result is a substitution S with $Se' = e$, $S\sigma' = \sigma$ that satisfies the constraints, as required. \square

D.5 Soundness and pseudo-completeness of inference phase 2

Lemma D.18 (Solver invariant)

Let the data structure of CS be interpreted as described in Section 3.5.4 and repeated below: Firstly, interpret each equivalence class having k members as a conjunction of $k - 1$ atomic equality constraints forming a spanning tree of members of the class. Secondly, interpret each mapping from a root variable u_i to a constant κ as an equality constraint $\langle u_i = \kappa \rangle$. Thirdly, interpret each mapping from a root variable u_i to a pair of bounds $(\overline{u}_k, \overline{v}_l)$ as a conjunction of inequality constraints $\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle$. The conjunction of these three sets of constraints is the constraint denoted by the data structure. Algorithm failure is interpreted as the unsatisfiable constraint $\langle 1 \leq \omega \rangle$. Further input is permitted after failure, but yields only repeated failure.

Now let C_1, \dots, C_n be atomic constraints. After these are input to the algorithm, the constraint denoted by the data structure is equal to $\bigwedge_{i=1}^n C_i$.

Proof The proof proceeds by induction on n . (In fact, strong induction on the total cost of the constraints C_1, \dots, C_n as defined in the proof of Theorem 3.6 is required. By storing \$1 on each constant mapping, \$2 on each variable except the first in each equivalence class, and \$2 on each variable in a bound, it is possible to recreate an arbitrary data structure with the specified cost. This allows the induction to go through. I have not worked out the full details of this in the proof below; instead I have merely assumed that the algorithm terminates and equivalently that the induction is well-founded.)

In the base case ($n = 0$), the data structure is empty and thus the resulting constraint is simply \emptyset , as required.

For the inductive step, we may assume that n constraints have already been input and the constraint denoted by the data structure is equal to $\bigwedge_{i=1}^n C_i$; we now input a further constraint C_{n+1} and wish to establish the result.

As described in Section 3.5.4, the constraint C_{n+1} must be either trivial and we do nothing (in which case the result follows trivially), false and we fail immediately (in which case again the result follows trivially), or equivalent to one of four possibilities, which we now consider.

case $\langle u_i = 1 \rangle$ or $\langle u_i = \omega \rangle$

If u_i maps to 1 or ω respectively, then there is already a constraint $\langle u_i = 1 \rangle$ or $\langle u_i = \omega \rangle$, and we need do nothing. Conversely, if u_i maps to ω or 1 respectively, we have an unsatisfiable constraint and failure is the correct behaviour. Otherwise we have $\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle$, and note that $\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle \wedge \langle u_i = 1 \rangle =^e \bigwedge_l \langle v_l = 1 \rangle \wedge \langle u_i = 1 \rangle$ (similarly for ω respectively). The result follows by a nested induction (well-founded by Theorem 3.6).

case $\langle u_i = u_j \rangle$

If u_i maps to 1, then there is already a constraint $\langle u_i = 1 \rangle$. Observe that $\langle u_i = 1 \rangle \wedge \langle u_i = u_j \rangle =^e \langle u_i = 1 \rangle \wedge \langle u_j = 1 \rangle$, justifying the algorithm's behaviour; the result follows by a nested induction. Proceed similarly for ω , or for the mapping of u_j . Otherwise we have

$$\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle \wedge \bigwedge_k \langle u'_k \leq u_j \rangle \wedge \bigwedge_l \langle u_j \leq v'_l \rangle$$

where u_i and u_j are in equivalence classes U_1 and U_2 . Merging the equivalence classes amounts to adding the single equality constraint $\langle u_i = u_j \rangle$; the new mapping is simply a rearrangement of the large constraint already given.

case $\langle u_i \leq u_j \rangle$

If u_i maps to 1, then there is already a constraint $\langle u_i = 1 \rangle$. Since $\langle u_i = 1 \rangle \wedge \langle u_i \leq u_j \rangle =^e \langle u_i = 1 \rangle \wedge \langle u_j = u_0 \rangle$, the result follows by a nested induction. If u_i maps to ω , then there is already a constraint $\langle u_i = \omega \rangle$. Since $\langle u_i = \omega \rangle \wedge \langle u_i \leq u_j \rangle =^e \langle u_i = \omega \rangle$, the result follows trivially. The dual cases apply to u_j . Otherwise we have

$$\bigwedge_k \langle u_k \leq u_i \rangle \wedge \bigwedge_l \langle u_i \leq v_l \rangle \wedge \bigwedge_k \langle u'_k \leq u_j \rangle \wedge \bigwedge_l \langle u_j \leq v'_l \rangle \wedge \langle u_i \leq u_j \rangle$$

and the result follows by simple rearrangement.

This concludes the proof. □

Theorem D.19 (Soundness and pseudo-completeness of inference phase 2)

For all constraints C ,

- (i) Algorithm \mathcal{CS} terminates.
- (ii) If there exists an S such that $\vdash^e SC$, then \mathcal{CS} succeeds.
- (iii) If \mathcal{CS} succeeds with substitution S , then $\vdash^e SC$, and for all S' such that $\vdash^e S'C$, $S' \sqsubseteq S$ (i.e., S is the best solution to C).

Proof (i) follows from Theorem 3.6. (ii) follows from Lemma D.18 and the observation that the derivation of S always succeeds. (iii) also follows from Lemma D.18: for the first part, we can see by inspection of the data structure that the constraint is satisfied by S , and for the second part, assume for

Figure D.1 Contexts.

$$\mathbb{C} ::= [\cdot] \mid x \mid n \mid \lambda x : \tau^{d,u} . \mathbb{C} \quad (\text{D.1})$$

$$\mid \mathbb{C} x \mid \mathbb{C}_1 + \mathbb{C}_2 \mid \text{let! } x : \tau^{d,u} = \mathbb{C}_1 \text{ in } \mathbb{C}_2 \quad (\text{D.2})$$

$$\mid \text{letrec } x_i : \tau_i^{d_i, u_i} = \mathbb{C}_i \text{ in } \mathbb{C}_0 \quad (\text{D.3})$$

$$\mid C_i \overline{\tau_k} \overline{y_j} \quad (\text{D.4})$$

$$\mid \text{case } \mathbb{C}_0 \text{ of } C_i x_{ij} : \tau_{ij}^{d_{ij}, u_{ij}} \mathbb{C}_i \quad (\text{D.5})$$

the sake of contradiction that there exists a substitution S' satisfying C but mapping a variable v to 1 that S maps to ω . Since S maps v to ω , C must contain the constraint $\langle v = \omega \rangle$; but then S' cannot satisfy C and we have our contradiction. Therefore S is indeed the best solution of C . \square

D.6 Extended system

We believe the extended type system of Appendix C can be proven sound, along the lines of a proof we have conducted for an earlier system. We sketch this earlier proof here.

We define *contexts* and *applicative contexts* along the lines of Moran and Sands [Mor98, MS99], and we define open evaluation for contexts (in fact this machinery is not necessary for the present setting). Contexts are defined in Figure D.1.

Lemma D.20 (Trans)

For all heaps H , applicative contexts \mathbb{A} , and stacks S , $\text{trans}\langle H; \mathbb{A}; S \rangle$ is equal to either

$$\text{trans}\langle H; \mathbb{A}'; \varepsilon \rangle$$

or

$$\begin{aligned} &\text{trans}\langle (H, x_1 : \tau_1^{d_1, u_1} = \mathbb{A}', x_2 : \tau_2^{d_2, u_2} = \mathbb{A}_1[x_1], x_3 : \tau_3^{d_3, u_3} = \mathbb{A}_2[x_2], \\ &\quad \dots, x_n : \tau_n^{d_n, u_n} = \mathbb{A}_{n-1}[x_{n-1}], \mathbb{A}_n[x_n], \varepsilon \rangle, \end{aligned}$$

for some $\mathbb{A}', n, x_i : \tau_i^{d_i, u_i}, \mathbb{A}_i$, where if the annotations in S are non-empty then for all i , d_i is nonempty and $0 \notin d_i$. Further, $\text{use}(\mathbb{A}, \text{use}(S, u_0))$ is equal to either $\text{use}(\mathbb{A}', u_0)$ or $\text{use}(\mathbb{A}', \text{use}(\mathbb{A}_1, \text{use}(\mathbb{A}_2, \dots \text{use}(\mathbb{A}_{n-1}, \text{use}(\mathbb{A}_n, u_0)) \dots)))$, respectively.

Proof By induction on S . \square

Lemma D.21 (\mathbb{A})

For any applicative context \mathbb{A} with hole type τ , and any expression e such that $\Gamma \vdash_{E'} e : \tau^{\mathbb{1}, \text{use}(\mathbb{A}, u_0)}$, there exist τ_0, u_0, Δ such that $\Delta \oplus \Gamma \vdash_{E'} \mathbb{A}[e] : \tau_0^{\mathbb{1}, u_0}$.

Proof By induction on \mathbb{A} . \square

Lemma D.22 (Value)

For all values v such that $\Gamma' \vdash_{E'} v : \tau^{\mathbb{1}, u'}$, it is the case that there exist Δ', Γ such that $\Gamma' = \Delta' \wedge (u' \cdot \Gamma)$, and for all Δ, u , $\Delta' \wedge (u \cdot \Gamma) \vdash_{E'} v : \tau^{\mathbb{1}, u}$.

Proof (omitted). □

Lemma D.23 (Paste)

If $\Gamma \vdash_{E'} \mathbb{C}[e] : \tau_0^{\mathbb{1}, u_0}$ and $\Delta \vdash_{E'} e : \tau^{\mathbb{1}, u}$, then $\Delta \vdash_{E'} e' : \tau^{\mathbb{1}, u} \Rightarrow \Gamma \vdash_{E'} \mathbb{C}[e'] : \tau_0^{\mathbb{1}, u_0}$.

Proof Proof is by induction over \mathbb{C} . □

Lemma D.24 (Substitution)

If $\Gamma, x : \tau^{d, u} \vdash_{E'} e : \tau_0^{\mathbb{1}, u_0}$ then $\Gamma \oplus y : \tau^{d, u} \vdash_{E'} e[y/x] : \tau_0^{\mathbb{1}, u_0}$.

Proof Trivial if $y \not\in \text{dom}(\Gamma)$; otherwise by induction on e . □

Theorem D.25 (Subject reduction)

If $\emptyset \vdash_{E'} \langle H; e; S \rangle : \tau_0^{\mathbb{1}, u_0}$ (and no d or u annotation in H , e , S is empty) and $\langle H; e; S \rangle \mapsto \langle H'; e'; S' \rangle$, then $\emptyset \vdash_{E'} \langle H'; e'; S' \rangle : \tau_0^{\mathbb{1}, u_0}$ and no d or u annotation in H' , e' , S' is empty.

Proof By induction on the reduction sequence, by cases on each reduction step.

(details omitted). □

Theorem D.26 (Termination condition)

If $\emptyset \vdash_{E'} \langle H; v; \varepsilon \rangle : \tau_0^{\mathbb{1}, u_0}$ (and no d or u annotation in H , v is empty) then every d and u annotation in H (at top level) contains a 0 (assuming u_0 is zero).

Proof Use (trans), ($\vdash_{E'}$ -LETREC), (Value), and fixpoint/monotonicity arguments with $d_{0j} = \emptyset$ by (Value). □

Bibliography

- [Abr90] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990. Cited on: 5.
- [AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *18th ACM Symposium on Principles of Programming Languages (POPL91)*, Orlando, Florida, pages 104–118. ACM Press, 1991. Cited on: 117, 144, 165.
- [AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Cited on: 117, 118, 145, 156, 165.
- [ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991. Extended version of paper at 16th POPL89. Cited on: 131.
- [ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Cited on: 131.
- [AF94] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Computer Science Dept, University of Oregon, October 1994. Cited on: 27.
- [AFM⁺95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In POPL [POP95]. Cited on: 5.
- [Amt94] Torben Amtoft. Local type reconstruction by means of symbolic fixed point iteration. In ESOP [ESO94]. Cited on: 119.
- [Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Chalmers University, Göteborg, Sweden, 1987. Cited on: 9.
- [AvGP92] Peter Achten, John van Groningen, and Rinus Plasmeijer. High-level specification of I/O in functional languages. In John Launchbury and Patrick Sansom, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, Ayr, Scotland, Workshops in Computing, pages 1–17. Springer-Verlag, November 1992. Revised version. Available <ftp://ftp.cs.kun.nl/pub/Clean/papers/1992/achp92-HighLevelIO.ps.gz>. Cited on: 70.
- [AVL62] G. M Adel'son-Vel'skiĭ and Y. M. Landis. An algorithm for the organisation of information. *Soviet Mathematics – Doklady*, 3:1259–1262, 1962. Cited on: 147.
- [AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In FPCA [FPC93], pages 31–41. Cited on: 74, 117.

- [AWP97] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping (extended abstract). In *Proceedings of TACS'97, International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan*, number 1281 in Lecture Notes in Computer Science, pages 47–76. Springer-Verlag, September 1997. Journal version appears as [AWP99]. Cited on: 118.
- [AWP99] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping (extended abstract). *Higher-Order and Symbolic Computation*, 12(3), October 1999. Journal version of [AWP97].
- [Bar81] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981. Cited on: 27.
- [Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Clarendon Press, Oxford, 1992. Cited on: 168.
- [BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ -programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985. Cited on: 140, 156.
- [BH98] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, May 1998. Cited on: 117, 145, 146, 156, 165.
- [BHY88] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1:147–164, 1988. Cited on: 50.
- [Bie91] Gavin Bierman. Type systems, linearity and functional languages (short summary). In *Montreal Workshop on Programming Language Theory*, December 1991. Unpublished. Available <http://www.cl.cam.ac.uk/users/gmb/Publications/tflfl.dvi.Z>. Cited on: 14, 257.
- [Bie92] Gavin Bierman. Type systems, linearity and functional languages. In *CLICS Workshop, Aarhus*, pages 71–91, 1992. Appears as Technical Report DAIMI PB-397-I, Department of Computer Science, Aarhus University. Cited on: 14, 257.
- [Bir84] G. M. Birtwhistle. *Simula begin*. Lund & Bromley, second edition, 1984. Cited on: 131.
- [BJ96] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages (IFL96)*, number 1268 in Lecture Notes in Computer Science, Bad Godesberg, Germany, September 1996. Springer-Verlag. Cited on: 13.
- [Bli89] Wayne D. Blizard. Multiset theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66, Winter 1989. Cited on: 55.
- [BM96] François Bourdoncle and Stephan Merz. On the integration of functional programming, class-based object oriented programming, and multi-methods. Technical Report 26, Centre de Mathématiques Appliquées, École des Mines de Paris, October 25 1996. Revised. Cited on: 165.

- [BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 302–315. ACM Press, January 1997. Cited on: 165.
- [BM98] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Proceedings of Mathematics of Program Construction, 4th International Conference, MPC'98, Marstrand, Sweden*, number 1422 in Lecture Notes in Computer Science, pages 52–67. Springer-Verlag, June 1998. Cited on: 159.
- [BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM Lisp Conference*, pages 136–143, 1980. Cited on: 128.
- [BP97] Andrew Barber and Gordon Plotkin. Dual intuitionistic linear logic. Unpublished; available <http://www.dcs.ed.ac.uk/home/agb/research.html>, October 15 1997. Cited on: 72.
- [BS93a] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, Computing Science Institute, University of Nijmegen, The Netherlands, December 1993. Extended abstract appeared as [BS93b]. Cited on: 70, 152.
- [BS93b] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Shyamasundar, editor, *Proceedings of 13th Conference on the Foundations of Software Technology and Theoretical Computer Science (FST&TCS13), Bombay, India*, number 761 in Lecture Notes in Computer Science, pages 41–51. Springer-Verlag, 1993. Short version of [BS93a]. Cited on: 70.
- [BS95a] Erik Barendsen and Sjaak Smetsers. A derivation system for uniqueness typing. In Corradini and Montanari, editors, *Proceedings of Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRA'95), Volterra, Pisa, Italy*, Electronic Notes in Theoretical Computer Science, pages 151–158. Elsevier, 1995. Cited on: 70.
- [BS95b] Erik Barendsen and Sjaak Smetsers. Uniqueness typing in theory and practice. In *PLILP'95*, 1995. LNCS 982. Available <http://www.cs.kun.nl/~clean/download/papers/1995/bare95-unitypeinference.ps.gz>. Cited on: 70, 111, 140.
- [BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996. Cited on: 70, 111, 123, 140, 141, 152.
- [BS98] Erik Barendsen and Sjaak Smetsers. Strictness typing. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages, University College London, UK*, pages 101–115, September 1998. Informal proceedings. Not in LNCS version. Cited on: 48, 70.
- [BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *POPL [POP96]*. Cited on: 227.
- [Bul84] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984. Cited on: 117, 123.

- [Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2), April 1987. Revised 21 June 1988. Cited on: 128.
- [Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL88)*, San Diego, California, pages 70–79. ACM, January 1988. Cited on: 95, 116, 117.
- [Car93] Luca Cardelli. An implementation of $F_{<}$. SRC Research Report 97, Digital Equipment Corporation, Systems Research Center, February 23 1993. Revised version of June 1997 available <http://www.luca.demon.co.uk/>. Cited on: 119.
- [Car96] Luca Cardelli. Type systems. In *CRC Handbook of Science and Engineering*, chapter 140. CRC Press, 1996. Cited on: 28.
- [CC92] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In *Nineteenth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL92)*, January 19–22, 1992, Albuquerque, New Mexico, pages 83–94. ACM Press, 1992. Cited on: 144.
- [CF91] Robert Cartwright and Mike Fagan. Soft typing. *ACM SIGPLAN Notices*, 26(6):278–292, June 1991. Cited on: 20.
- [CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in F_{\leq} . *Mathematical Structures in Computer Science*, 2:55–91, 1992. Cited on: 86, 116.
- [Chu33] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2(33, 34):346–366, 839–864, 1932–33. Cited on: 3.
- [Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Number 6 in Annals of Mathematics Studies. Princeton University Press, 1941. Cited on: 140.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Engineering and Computer Science Series. MIT Press / McGraw–Hill, 1990. Ninth printing, 1993. Cited on: 57, 62, 63.
- [CMMS94] Luca Cardelli, John C. Mitchell, Simone Martini, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1/2):4–56, February 1994. Cited on: 86, 116.
- [Cra99] Karl Cray. A simple proof technique for certain parametricity results. In ICFP [ICF99], pages 82–89. Cited on: 131.
- [Cur90] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Palo Alto Research Center, February 1990. Cited on: 74, 117.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985. Cited on: 4, 86, 95, 116.
- [DHM95] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Mycroft [Myc95]. Extended version, formerly available from third author’s home page. Cited on: 91, 111, 123.

- [DM58] J. C. E. Dekker and J. Myhill. Some theorems on classes of recursively enumerable sets. *Transactions of the American Mathematical Society*, 89(1):25–59, September 1958. Cited on: 17.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982. Cited on: 28, 71, 89, 128.
- [dMJB⁺99] Paul de Mast, Jan-Marten Jansen, Dick Bruin, Jeroen Fokker, Pieter Koopman, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Functional programming in CLEAN – part I: Introduction on functional programming. Draft of work in progress, available http://www.cs.kun.nl/~clean/Manuals/Clean_Book/clean_book.html, July 1999. Cited on: 48, 70.
- [Dun86] J. M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume III. D. Reidel, 1986. Cited on: 14.
- [EHM⁺99] Peter Harry Eidorff, Friz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. AnnoDomini: From type theory to Year 2000 conversion tool. In POPL [POP99], pages 1–14. Cited on: 16.
- [ESO94] *European Symposium on Programming (ESOP'94)*, number 788 in Lecture Notes in Computer Science. Springer-Verlag, April 1994.
- [EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of OOPSLA'95*, pages 169–184, 1995. Cited on: 118.
- [Fag90] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, December 1990. Cited on: 20.
- [Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In Mycroft [Myc95]. Cited on: 14, 69, 70.
- [Fax97] Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, 1997. Appears as Research Report TRITA-IT R 97:08. Cited on: 14.
- [FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999. Cited on: 71, 118, 119.
- [FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 72(2):155–175, June 1990. Cited on: 118.
- [FPC93] *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, Copenhagen, Denmark. ACM Press, 1993.
- [FPC95] ACM. *Conference Record of FPCA'95 SIGPLAN–SIGARCH–WG2.8 Conference on Functional Programming Languages and Computer Architecture, La Jolla, California, 25–28 June 1995*, 1995.
- [Fre97] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, Paris, France, number 1302 in Lecture Notes in Computer Science, pages 265–277. Springer-Verlag, September 1997. Cited on: 117.

- [FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, pages 237–247, June 1993. SIGPLAN Notices 28(6). Cited on: 27.
- [FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 1987: Proceedings*, number 274 in Lecture Notes in Computer Science, pages 34–45. Springer-Verlag, 1987. Cited on: 9, 39.
- [GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, June 1990. Cited on: 13, 48, 70.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. Cited on: 4.
- [Gil96] Andrew John Gill. *Cheap Deforestation for Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, January 1996. Cited on: 5, 6, 12, 13, 77, 138, 186, 188.
- [Gir72] Jean-Yves Girard. *Interpretation Fonctionnelle Et Elimination Des Coupures De L'Arithmétique D'Ordre Supérieur*. These d'état, Université Paris VII, 1972. Cited on: 7, 81, 116, 128.
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987. Cited on: 13.
- [Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993. Cited on: 72.
- [Gir95] Jean-Yves Girard. Linear logic: Its syntax and semantics. In Girard, Lafont, and Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Notes Series. Cambridge University Press, 1995. Cited on: 13.
- [GLP00] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 221–231, Montréal, Canada, September 2000. ACM Press. Cited on: 117, 165.
- [GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In FPCA [FPC93], pages 223–232. Cited on: 138, 186, 188.
- [Gol87] Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, number 274 in Lecture Notes in Computer Science, pages 408–425. Springer-Verlag, September 1987. Cited on: 12, 176.
- [Gor94] Andrew D. Gordon. A tutorial on co-induction and functional programming. In K. Hammond, D.N. Turner, and P. Sansom, editors, *Proceedings of the 1994 Glasgow Functional Programming Workshop, Ayr, Scotland, 12–14 September 1994*, Workshops in Computing. Springer-Verlag, 1994. Cited on: 144.
- [GS00a] Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of Higher Order Operational Techniques in Semantics (HOOTS99)*,

- volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2000. Cited on: 12.
- [GS00b] Jörgen Gustavsson and Josef Sveningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Implementation of Functional Languages 12th International Workshop (IFL00)*, Aachen, Germany, September 2000. Springer. LNCS 2011. Technical report, Department of Computer Science, RWTH Aachen. Extended version appears as [Gus01a]. Cited on: 14, 15, 27, 40, 69, 72, 74, 79, 91, 120, 141, 164.
- [GS01a] Jörgen Gustavsson and David Sands. On usage analyses for work and space-safe inlining. In *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages* [Gus01a], paper II. Cited on: 12, 15, 39, 272.
- [GS01b] Jörgen Gustavsson and Josef Sveningsson. Constraint abstractions. In *Proceedings of the Symposium on Programs as Data Objects II*, number 2053 in Lecture Notes in Computer Science. Springer, May 2001. Cited on: 69, 120, 228.
- [GSSS01] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Søndergaard. Boolean constraints for binding-time analysis. In Olivier Danvy and Andrzej Filinski, editors, *Proceedings of the Second Symposium on Programs as Data Objects (PADO'01)*, number 2053 in LNCS, pages 36–92. Springer, 2001. Cited on: 123.
- [GTW78] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology IV: Data Structuring*, chapter 5, pages 80–149. Prentice-Hall, 1978. Cited on: 128.
- [Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. MIT Press, 1992. Cited on: 50.
- [Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In ICFP [ICF98], pages 39–50. Extended version appears as [Gus99]. Cited on: 12, 14, 15, 26, 39, 42, 68, 69, 70, 72, 176, 214, 273.
- [Gus99] Jörgen Gustavsson. *A Type Based Sharing Analysis for Update Avoidance and Optimisation*. Licentiate thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, April 1999. Also appears as [Gus01b]; page numbers refer to that edition. Cited on: 14, 44, 64, 69, 70, 164, 165, 176.
- [Gus01a] Jörgen Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages*. Department of Computing Science, Chalmers University of Technology, Göteborg University, May 2001. PhD thesis. Cited on: 14, 21, 113.
- [Gus01b] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages* [Gus01a], paper III.
- [Haw88] Stephen W. Hawking. *A Brief History of Time: From the Big Bang to Black Holes*. Bantam Books, 1992 reprint edition, 1988. Cited on: 180.
- [Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [Hug91], pages 448–472. Cited on: 71.

- [Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, April 1993. Cited on: 91, 110, 159.
- [Hen96] Fritz Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report D-293, DIKU, University of Copenhagen, May 1996. Cited on: 121.
- [Hen99] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. *Theory and Practice of Object Systems (TAPOS)*, 5(1):57–72, 1999. Also TOPPS Report D-396, DIKU, University of Copenhagen. Cited on: 118.
- [HFA⁺96] Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Meemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. van Groningen, Kevin Hammond, Bogumil Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Rojemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with ‘Pseudoknot’, a float-intensive benchmark. *Journal of Functional Programming*, 6(4):612–655, July 1996. Cited on: 6.
- [HHPJW94] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. In ESOP [ESO94], pages 241–256. Cited on: 91.
- [Hin69] J. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. Cited on: 42, 66, 71, 128.
- [HJ94] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... an experiment in software prototyping productivity. Unpublished manuscript, July 4 1994. Available <http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul.html>. Cited on: 3, 5.
- [HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In ESOP [ESO94], pages 287–301. Slightly revised version, DIKU Technical Report D-198. Cited on: 50, 113, 123.
- [HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In POPL [POP95]. Cited on: 4.
- [Hop71] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. Cited on: 118, 119.
- [HPT98] Haruo Hosoya, Benjamin C. Pierce, and David N. Turner. Datatypes and subtyping. Unpublished manuscript. Available <http://www.cis.upenn.edu/~bcpierce/papers/index.html>, July 24 1998. Cited on: 165.
- [HR98] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *Proc. 25th Int’l Coll. on Automata, Languages and Programming (ICALP)*, Aalborg, Denmark, number 1443 in Lecture Notes in Computer Science, pages 616–627. Springer-Verlag, July 1998. Also TOPPS Report D-394, DIKU, University of Copenhagen. Cited on: 117, 118.
- [HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. Cited on: 17, 118.

- [Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2), February 1989. Cited on: 3, 5.
- [Hug91] John Hughes, editor. *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, Cambridge, Massachusetts, number 523 in LNCS. Springer, August 1991.
- [HW90] Paul Hudak and Philip (editors) Wadler. Report on the programming language Haskell, a non-strict purely functional language. Technical Report YALEU/DCS/RR777, Department of Computer Science, Yale University, April 1990. Available <http://www.haskell.org/definition/>. Cited on: 6.
- [ICF97] ACM. *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, 1997. Appears as ACM SIGPLAN Notices 32(8), August 1997.
- [ICF98] *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, 1998.
- [ICF99] *International Conference on Functional Programming (ICFP'99)*, Paris, France, September 1999.
- [Ing61] P. Z. Ingerman. Thunks. *Communications of the ACM*, 4(1):55–58, January 1961. Cited on: 9.
- [Jac94] Bart Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994. Cited on: 14, 71.
- [Jen91] Thomas P. Jensen. Strictness analysis in logical form. In Hughes [Hug91]. Extended version appears as [Jen95]. Cited on: 16, 123, 273.
- [Jen95] Thomas P. Jensen. Conjunctive type systems and abstract interpretation of higher-order functional programs. *Journal of Logic and Computation*, 5(4):397–421, 1995.
- [Jen98] Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'98)*, January 1998, San Diego, California. ACM Press, 1998. Cited on: 273.
- [Jim96] Trevor Jim. What are principal typings and what are they good for? In *Twenty-third ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, pages 42–53. ACM Press, 1996. Cited on: 18, 42, 66.
- [Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University, Göteborg, Sweden, 1987. Cited on: 9.
- [Jon92] Richard E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992. Cited on: 34.
- [Jon99] Mark P. Jones. Type classes and functional dependencies. Available <http://www.cse.ogi.edu/~mpj/fds.html>, September 10 1999. Cited on: 6.
- [JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Available <http://www.cs.purdue.edu/homes/palsberg/publications.html>, June 1999. Cited on: 117, 118, 124.
- [JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997. Cited on: 144.

- [JW95] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In Mycroft [Myc95], pages 207–224. Cited on: 16, 117, 123.
- [KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of FPCA'89, Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989. Cited on: 273.
- [KMM91] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991. Cited on: 110.
- [Kob99] Naoki Kobayashi. Quasi-linear types. In POPL [POP99], pages 29–42. Cited on: 15, 48, 50, 270.
- [Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964. Cited on: 3, 33, 39, 128.
- [Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. Cited on: 3.
- [Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina, January 10–13, 1993*, pages 144–154. ACM Press, 1993. Cited on: 14, 27, 34, 36, 39.
- [Lei83] Daniel Leivant. Polymorphic type inference. In *POPL'83*, 1983. Reference from CiteSeer. Cited on: 89.
- [LGH⁺92] John Launchbury, Andy Gill, John Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In Launchbury and Sansom [LS93]. Cited on: 14, 70.
- [LKS00] John Launchbury, Sava Krstić, and Timothy E. Sauerwein. Zip fusion with hyperfunctions. Available <http://www.cse.ogi.edu/~krstic/>, 2000. Cited on: 158.
- [LM92] Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Conference Record of POPL '92: 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, 1992. ACM Press. Cited on: 117, 118.
- [LNS82] J.-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, May 1982. Cited on: 144.
- [LP96] John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In Nielson [Nie96]. Cited on: 131.
- [LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press. Extended version appears as [LPJ95]. Cited on: 114.
- [LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, 1995. Cited on: 189.
- [LS93] John Launchbury and Patrick M. Sansom, editors. *Functional Programming, Glasgow 1992, Workshops in Computing*, Berlin, 1993. Springer-Verlag.

- [LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In FPCA [FPC95]. Cited on: 138.
- [LSML00] Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. In *Twenty-seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL00)*, Boston, Massachusetts. ACM Press, January 2000. To appear. Cited on: 6, 168.
- [LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998. Cited on: 28, 128.
- [MAE⁺62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 17 August 1962. Cited on: 3.
- [Mar93] Simon Marlow. Update avoidance analysis by abstract interpretation. In *Glasgow Workshop on Functional Programming*, Ayr, Springer Verlag Workshops in Computing Series, July 1993. Cited on: 9, 10, 12, 13, 68, 273.
- [MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In FPCA [FPC95], pages 324–333. Cited on: 158.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978. Cited on: 4, 14, 17, 28, 31, 36, 71, 89, 93, 128.
- [Mit84] John C. Mitchell. Coercion and type inference (summary). In POPL [POP84], pages 175–185. Cited on: 117.
- [Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, 1991. Cited on: 117.
- [Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996. Cited on: 50, 55.
- [MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982. Cited on: 117, 121.
- [Mog97a] Torben Æ. Mogensen. Types for 0, 1 or many uses. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages*, St. Andrews, Scotland, pages 157–165, September 1997. Appears as LNCS 1467. Cited on: 12, 14, 15.
- [Mog97b] Torben Æ. Mogensen. Types for 0, 1 or many uses. Updated version of [Mog97a]. Obtained by email from author, 1997. Cited on: 270.
- [Mog98] Torben Æ. Mogensen. Types for 0, 1 or many uses. Updated and corrected version of [Mog97a]. Obtained from author, 1998. Cited on: 14, 141, 272.
- [Mon93] B. Monsuez. Polymorphic types and widening operators. In P. Cousot, M. Falaschi, G. File, and A. Ranzy, editors, *Static Analysis: Proceedings of the Third International Workshop (WSA '93)*, Padova, Italy, number 724 in Lecture Notes in Computer Science, pages 267–281. Springer-Verlag, September 22–24 1993. Cited on: 16.
- [Mor98] Andrew Keith Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, September 1998. Cited on: 310.

- [Mos93] Christian Mossin. Polymorphic binding time analysis. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, July 21 1993. Cited on: 50, 70, 123.
- [MOTW95] John Maraist, Martin Odersky, David Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus (extended abstract). In *11th International Conference on the Mathematical Foundations of Programming Semantics*, April 1995. Cited on: 14, 71.
- [MPJMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *PLDI'01*, 2001. Cited on: 7.
- [MPS84] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *POPL [POP84]*, pages 165–174. Cited on: 95.
- [MS99] A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *POPL [POP99]*. To appear. Cited on: 26, 39, 310.
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML — Revised*. MIT Press, 1997. Cited on: 3, 85, 87, 89, 128.
- [MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP [ICF97]*. Appears as ACM SIGPLAN Notices 32(8), August 1997. Cited on: 119.
- [MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):528–569, May 1999. Cited on: 4, 83.
- [Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer-Verlag, April 1984. Citation from [Oka97]. Cited on: 91, 159.
- [Myc95] Alan Mycroft, editor. *Proceedings of the Second International Static Analysis Symposium (SAS), Glasgow, Scotland*, number 983 in *Lecture Notes in Computer Science*. Springer-Verlag, September 25–27 1995.
- [Nie96] Hanne Riis Nielson, editor. *Programming Languages and Systems—ESOP '96: 6th European Symposium on Programming, Linköping, Sweden, April 1996*, number 1058 in *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. Cited on: 15, 16, 52.
- [Nor98] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *ICFP [ICF98]*. Cited on: 66, 120, 121, 165.
- [Nor99] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, May 1999. Cited on: 120.
- [Nor02] Johan Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2–3):93–127, May-June 2002. Cited on: 120.
- [Nuu94] Esko Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52:207–213, 1994. Cited on: 105, 110.
- [Oka97] Chris Okasaki. Catenable double-ended queues. In *ICFP [ICF97]*. Appears as ACM SIGPLAN Notices 32(8), August 1997. Cited on: 159.

- [Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. Cited on: 159.
- [Oka99] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In ICFP [ICF99], pages 28–35. Cited on: 159.
- [OL96] Martin Odersky and Konstantin Läuffer. Putting type annotations to work. In POPL [POP96]. Cited on: 114, 187.
- [OSW98] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 1998. To appear. Cited on: 119, 123, 124, 290.
- [Par93] Will D. Partain. The `nofib` benchmark suite of Haskell programs. In Launchbury and Sansom [LS93], page 195. Cited on: 10, 19, 190.
- [PH⁺96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non-strict, purely functional language, version 1.3. Technical Report YALEU/DCS/TR1106, Yale University, May 1 1996. Cited on: 253.
- [Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, 20 December 1991. Available as technical report CMU-CS-91-205. Cited on: 140.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002. Cited on: 4.
- [Pit01] Andrew M. Pitts. Nominal Logic: A first order theory of names and binding. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium (TACS 2001), Sendai, Japan*, number 2215 in Lecture Notes in Computer Science, pages 219–242. Springer, October 2001. Invited talk. Cited on: 27.
- [PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992. Cited on: 6, 8, 9, 19, 50, 65.
- [PJ96] Simon L. Peyton Jones. Compilation by transformation: A report from the trenches. In Nielson [Nie96], pages 18–44. Cited on: 12.
- [PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent haskell. In POPL [POP96]. Cited on: 6.
- [PJH⁺99] Simon Peyton Jones, John Hughes, et al. Report on the programming language Haskell 98, a non-strict, purely functional language, February 1 1999. Cited on: 3, 6, 128, 168.
- [PJJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the 2nd ACM Haskell Workshop*, May 2 1997. Available <http://research.microsoft.com/Users/simonpj/Papers/type-class-design-space/>. Cited on: 6, 168.
- [PJL91] Simon L. Peyton Jones and David Lester. A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, May 1991. Cited on: 12, 13.
- [PJL92] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992. Out of print. Available <http://research.microsoft.com/Users/simonpj/Papers/papers.html>. Cited on: 9.

- [PJM97] Simon L. Peyton Jones and Erik Meijer. Henk: a typed intermediate language. In *Types in Compilation Workshop*, June 1997. Available: <http://www.cse.ogi.edu/~simonpj/>. Cited on: 168.
- [PJM99] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *Workshop on Implementing Declarative Languages*, 1999. Revised version to appear in *Journal of Functional Programming*. Available <http://research.microsoft.com/Users/simonpj/Papers/inlining/index.htm>. Cited on: 11.
- [PJP93] Simon L. Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In Kevin Hammond and John O'Donnell, editors, *Functional Programming, Glasgow 1993*, Springer Verlag Workshops in Computing, pages 201–220, 1993. Cited on: 273.
- [PJPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: Moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, New York, 1996. ACM Press. Cited on: 12, 83.
- [PJS98a] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998. Cited on: 4, 6, 8, 27, 81, 128, 252, 273.
- [PJS98b] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. In *Science of Computer Programming* [PJS98a], pages 3–47. Cited on: 12.
- [PJTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2001 Haskell Workshop, Florence, Italy*, pages 203–233, September 2001. Cited on: 186.
- [POP84] *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah*. ACM Press, 1984.
- [POP95] *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, New York, 1995. ACM Press.
- [POP96] *Twenty-third ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), January 1996*. ACM Press, 1996.
- [POP99] *Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), January 20–22, 1999, San Antonio, Texas*. ACM Press, 1999.
- [POP01] *Twenty-eighth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), 17–19 January 2001, The Royal Society, London*. ACM Press, 2001.
- [Pot98] François Pottier. *Type Inference in the Presence of Subtyping: From Theory to Practice*. PhD thesis, Université Paris VII, July 3 1998. Also available in French. Cited on: 118.
- [Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000. Cited on: 118.
- [Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, November 2001. Cited on: 71, 95, 118.

- [PS92] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, pages 175–180, 1992. Cited on: 16.
- [PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, 1993. Full version to appear in *MSCS*. Cited on: 117, 144.
- [PS96] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996. Cited on: 16, 117.
- [PS97a] Benjamin Pierce and Martin Steffen. Corrigendum to “higher-order subtyping”. *Theoretical Computer Science*, 184:247, 1997. Corrigendum to [PS97b].
- [PS97b] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176:235–282, 1997. Corrigendum appears as [PS97a].
- [PT97] Benjamin C. Pierce and David N. Turner. Local type inference. CSCI Technical Report #493, Indiana University Computer Science Department, November 12 1997. Cited on: 119, 121.
- [PvE98] Rinus Plasmeijer and Marko van Eekelen. Language report: Concurrent Clean version 1.3. HILT and University of Nijmegen. Available <http://www.cs.kun.nl/~clean/>, April 1998. Cited on: 48, 69, 123, 152.
- [R⁺01] Eric S. Raymond et al. The jargon lexicon (4.3.0). Available <http://www.tuxedo.org/~esr/jargon/>, June 2001. Cited on: 155.
- [Ray91] Eric S. Raymond, editor. *The New Hacker's Dictionary*. MIT Press, 1991. Published version of the Jargon File, <http://www.tuxedo.org/~esr/jargon/>. Cited on: 9, 56.
- [Rey74] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, number 19 in *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974. Cited on: 7, 81, 116, 128.
- [Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, 1983. Cited on: 131.
- [RF01] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL [POP01]*, pages 54–66. Cited on: 56, 120, 228.
- [Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953. Cited on: 17.
- [RM99] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35:191–221, 1999. Cited on: 71.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. Cited on: 117.
- [SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–129, La Jolla, CA, 1995. Cited on: 4.

- [San95] André Luís de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, July 1995. Available as Technical Report TR-1995-17. Cited on: 12, 187.
- [Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, October 3 1991. Available as DIKU Research Report 92/6. Cited on: 12, 13, 15, 272.
- [Ses97] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997. Cited on: 25, 26, 31, 36, 39.
- [Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π -calculus. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming: 25th International Colloquium, ICALP'98, Aalborg, Denmark*, number 1443 in Lecture Notes in Computer Science, pages 695–706. Springer-Verlag, June 1998. Cited on: 71.
- [SM01] Mark Shields and Erik Meijer. Type-indexed rows. In POPL [POP01], pages 261–275. Cited on: 168.
- [SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC-99-009, University of South Australia, School of Computer and Information Science, July 1999. Available <http://ps.uni-sb.de/~mmueller/papers.html>. Cited on: 116, 123, 124.
- [SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping. Technical Report LFCS-ECS-94-280, University of Edinburgh, February 1994. Also Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94. Revised version with Fun subtyping. Journal version appears as [PS97b, PS97a]. Cited on: 86, 116, 165.
- [Str67] Christopher Strachey. Fundamental concepts in programming languages. Lecture notes, International Summer School in Computer Programming, Copenhagen, Programming Research Group, Oxford University. Reprinted as [Str00], August 1967. Cited on: 128, 131.
- [Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, May 2000. Reprints [Str67].
- [Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association of Computing Machinery*, 22(2):215–225, April 1975. Cited on: 57.
- [Tar94] Quentin Tarantino. *Pulp Fiction*. Faber and Faber, London, 1994. Cited on: 51.
- [TGT01] Andrew Tolmach and The GHC Team. An external representation for the ghc core language (draft for ghc5.02). Available <http://www.haskell.org/ghc/docs/papers/core.ps.gz>, September 6 2001. Cited on: 8.
- [Tiu97] Jerzy Tiuryn. Subtyping over a lattice. In *Gödel Colloquium*, Vienna, 1997. Updated August 1997. Cited on: 117, 120.
- [TMC⁺96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, 1996. Cited on: 27.

- [TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the Third International Symposium on Static Analysis (SAS'96)*, Aachen, Germany, number 1145 in Lecture Notes in Computer Science, pages 349–265. Springer-Verlag, September 1996. Cited on: 44, 67, 71, 117, 118, 119, 124.
- [TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pages 188–201, 1994. Cited on: 16.
- [TvL84] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the Association of Computing Machinery*, 31(2):245–281, April 1984. Cited on: 57.
- [TWM95a] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In FPCA [FPC95], pages 1–11. Cited on: 10, 13, 14, 15, 26, 39, 42, 45, 50, 66, 68, 121, 137, 141, 152, 164, 169, 176, 224, 225, 272.
- [TWM95b] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. Technical Report TR-1995-8, Computing Science Department, University of Glasgow, 1995. Extended version of [TWM95a]. Available <http://www.dcs.gla.ac.uk/people/old-users/dnt/>. Cited on: 14, 48, 68, 72, 225.
- [Var94] Various. Music from the motion picture Pulp Fiction. Compact disc, 1994. MCAD11103. MCA Records, Universal City, CA 91608, USA. Cited on: 51.
- [Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971. Cited on: 5, 8.
- [Wad89] Philip Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989, June 1989. Slightly revised version. Cited on: 131.
- [Wad90a] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. Earlier version appeared in ESOP'88, LNCS 300. Cited on: 5, 138.
- [Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990. Cited on: 13.
- [Wad91] Philip Wadler. Is there a use for linear logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM)*, Yale University, June 1991. Available <http://www.research.avayalabs.com/user/wadler/topics/linear-logic.html>. Cited on: 13.
- [Wad93a] Philip Wadler. A syntax for linear logic. In Brookes et al., editors, *Mathematical Foundations of Programming Semantics*, New Orleans, April 1993, *proceedings*, number 802 in Lecture Notes in Computer Science. Springer-Verlag, 1993. Cited on: 13.
- [Wad93b] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, Gdansk, August–September 1993, *proceedings*, number 711 in Lecture Notes in Computer Science. Springer-Verlag, 1993. Cited on: 13.
- [Wad97] Philip Wadler. A HOT opportunity. *Journal of Functional Programming*, 7(2), March 1997. Editorial. Cited on: 4, 16.

- [Wad98] Philip Wadler. Functional programming: An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, February 1998. Cited on: 3.
- [Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticæ*, 10:115–122, 1987. Cited on: 117.
- [WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *16th ACM Symposium on Principles of Programming Languages (POPL'89)*, Austin, Texas, January 1989, 1989. Cited on: 168.
- [WBF93] David A. Wright and Clement A. Baker-Finch. Usage analysis with natural reduction types. In *Third International Workshop on Static Analysis, Padora*, pages 254–266, 1993. Appears as LNCS 724. Cited on: 14.
- [Wel94] J. B. Wells. Typability and type checking in the second-order λ -calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, Paris, July 4–7 1994. Cited on: 114.
- [WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Cited on: 17, 20, 29, 31.
- [WJ98] Andrew Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998. Cited on: 117, 123.
- [WPJ98] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. Technical Report TR-1998-19, Department of Computing Science, University of Glasgow, 1998. Cited on: 130, 164, 165.
- [WPJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In POPL [POP99]. Cited on: 42, 70, 78, 115, 130, 164, 165, 169, 183, 197, 214, 226.
- [WPJ00] Keith Wansbrough and Simon L. Peyton Jones. Simple usage polymorphism. In Robert Harper, editor, *Third Workshop on Types In Compilation (TIC 2000)*, Montreal, Canada, September 21 2000. Preliminary proceedings. Cited on: x, 123, 169.
- [Wri96] David A. Wright. Linear, strictness and usage logics. In *Proceedings of CATS'96 (Computing: The Australasian Theory Symposium)*, Melbourne, Australia, January 1996. Cited on: 14, 273.

Index

Page numbers in bold denote the primary or defining occurrence of the term. An appended *fn* denotes an occurrence in a footnote. For notation, see also Appendix A.

- A-normal form, 27, 34, 131, 141, 175, 177, 187, 215, 266, 267, 281
- absence, 7, 14, 229, 251–253, 257
- absent argument transformation, 252
- abstract interpretation, 12, 13, 16, 123, 273
- abstract machine, 6–9, 31, 39
- abstraction, 45, 267
- accounting method, 62, 308
- accuracy
 - requires zero annotation, 253
- addition \oplus , 256, 267, 269, 270
- add_n , 29
- adjacent lambdas, 65, 65*fn*, 121
- administrative operations, 81
- affine linear logic, 14, 48, 71–72, 273
- affine usage, 257
- algebra, 256
- algebraic data type, 14, 42, 77, 115, **128**, 129, 135–147, 169, 214, 224, 226, 228, 264, 266, 270
 - declaration, **129**, 131, 164, 266
 - mutually recursive, 156*fn*, 182
 - notation, *see* annotation scheme
 - properties of, 137
 - typical, 147
- Algorithm \mathcal{M} , 128
- Algorithm \mathcal{W} , 71, 117, 121, 128
- All (analysis parameter setting), 195
- all alike, *see* recursion, mutual
- allocation, 19, 199–200, 205, 214
- α -conversion, 27, 79, 81, 82, 86, 123, 129, 135, 161
- Anal (analysis parameter setting), 204
- analysis, 15
 - extending, 251–273
 - improvement of, 209
- “and” function, 77
- annotated data type declaration, 138, **139**, 147–159, 161, 182
 - restriction on, 150, 153, 163
- annotation, 11, 14–16, 18, 39, 45, 52, 71, 81, 93, 130, *see also* topmost
 - annotation, ground annotation, usage annotation
 - ordering, *see* primitive ordering
 - static, 211
 - type, *see* type annotation
- annotation algebra, 273
- annotation category, 183
- annotation domain, 11, 14, 15, 39, 58, 70, 100, 111, 254–257, 272
- annotation lattice, 83
- annotation position, 44, 56, 67, 148
- annotation scheme, 78, 137, 148–159, 163–165, 169, 197, 224, 226, 228
 - elaboration of, 182
- annotations
 - of a type, 291
 - too large, 264
- application, 267
 - constructor, 149
 - constrained, 141
 - unsaturated, 177
 - function, 149
- application site, 65, 74, 75, 77, 93, 109, 113, 117, 119, *see also* call site
- applicative context, **310**
- approximation, 17, 48, 51, 89–91, 94, 105, 109, 111, 113, 115, 119, 121, 124–125, 149, 177, 224
 - desirable, 73
 - explicit, 123
- apteryx australis*, 62*fn*
- are, *see* in a
- arity, 184
 - of constructor, 131
- arrow, 44, 266
- asymmetry, 155, 156

- at most once, 10, 43, 47, 223, 229, 251,
 see also thunk, single-entry
- atom, 27, 81, **81**, 131, 281
- atomic, *see* A-normal form
- atomic constraint, **290**
- augment*, 77*fn*
- AVL tree, 147, 152
- Bad*, **35**
- BadBinding*, **35**, 59, 106, 259
- BadDemand*, 259
- BadUsage*, 259
- BadValue*, **35**, 59, 106, 259
- bailing out, 159
- balance, 225
- balanced binary tree, 145*fn*, 159
- Barendregt Variable Convention, **27**
- base types, 128
- benchmark, *see* NoFib
- benefit, 212, 213
- best solution, *see* goodness ordering
- β -reduction, 34, 83*fn*
- Bierman lattice, 14, 257, 272
- big-step semantics, 39
- binary size, 200
- binders
 - list of, 185
- binding, 177, 211, 254, 255, 257, 269
 - avoiding, 130*fn*
 - usage of, 50, 68
- binding group, 50, 83, 89, 110, 164, 176,
 177, 269
- binding-time analysis, 50, 111, 113, 123
- bivariant, 103*fn*, 158, 182
- black hole, **35**, 173
- BlackHole*, **35**, 59, 106, 259*fn*, 287
- blackholing, 12, **34**, 39, 191
- book-keeping, 131
- books, 230
- boring, *see* usage variable,
 boring/interesting
- \perp , 180
- bottom element, 83
- bottom-line impact, 193
- bottom-propagation, **179**
- bounded polymorphism, 116, 119, 123,
 165
- bowtie, 96
- boyer, 191, 198, 210–213
- branch, **130**, 130*fn*, 142, 161, 212
- bspt, 191, 204, 205, 215
- build*, 77*fn*
- build* hack, 188, 200
- cacheprof, 191, 198, 199, 215
- CAF, 187, 192, 211
- call site, 172, 174, 176, *see also*
 application site
- call-by-name, **4**, 6, 14
- call-by-need, *see* lazy evaluation
- call-by-value, **4**, 6, 14, 15, 251, 253
- candidate variable, **90**, 99–101, 112
- canonical proof tree, 55, 87, 300
- capability ordering, 68
- capture, 82
- case, 7, 70, 128, 130*fn*, 131, 142, 161,
 168, 177, 178, 182, 188, 270
- case studies, 209–212
- chief architect, 168
- cichelli, 195, 214
- clausify, 191, 205
- Clean, **69**, 123, 140, 141, 151
- clean separation, 138
- cloning, *see* specialisation
- closed
 - heap binding, 82
 - type, 118
- closure, *see* thunk
- closure conversion, 83
- closure operation, 89, **89**, 94, 96, 100,
 101–106, 108, 121, 121*fn*, 224,
 228, 272, 291
 - trivial, 104
- cluster, 90, **103**, 105, 121
- code generation, 81, 95, 115, 187, 227
- code size, 77, 114
- coercion, 50, 70, 123, 169, **179**
- coinduction, 117, 144, 165
- common pattern, 78
- common subexpression elimination,
 171*fn*
- compilation time, 200
- compiler, 1, 4, 6, 12, 18, 51, 70, 81,
 167–215, 226
- compiler options, 172
- completeness, 17, 18, 61, 66, 119
 - lack of, 108, 109
 - of inference, 307
- complexity, 73, 77, 110, 164
 - amortized, 62
 - theorem, 63
- component, 138–140, 142, 146, 266, 270
- compose*, 78, 114
- computability, 17

- conditional, **27**, 34
- conditional constraint, 291
- conditional type, 273
- cones
 - upward and downward, 105
- configuration, **33**, 37, 259
 - remains closed, 59
 - stuck, **35**
 - terminal, **35**
 - value, **35**
- conjunction
 - of constraints, 108
- conjunction \wedge , 257, 267, 269
- conservative extension, 78
- const*, 135
- constrained polymorphism, 66, 69, 70, 74, 91, 94, 111, 113, 118, 123, 149, 165, 224, 225, 228
 - related work, 116–120
- constraint, 14, 16, 52, **53**, 55, **55**, 56, 57, 61, 74, 78, 87, 89, 90, 92, 93, 96, 99, 101, 105, 117, 119–121, 140, 165, **290–291**
 - covariant of, 52
 - insoluble, 121
 - of candidate, 100
 - residual, 90
 - unsatisfiable, 57
- constraint abstraction, 120
- constraint set, 172
- constraint solver, **57–58**, 61, 64, 69, 71, 214, 272, 308
 - implementation of, 206
- constraint-based analysis, **16**
- constraints, 197
- constructor, 68, 128, 129, 131, 141, 161, 168, 181, 200, *see also* datum
 - encoding of, 140
- constructor argument, 177, 181, 182
- constructor function, 78
- context, *see* evaluation context
- context sensitivity, 74
- continuation-passing, 157
- contraction, 270
- contravariant, **44**, 44*fn*, 50, 79, 143, 146
- contributions, 1–3
- control, **33**, 39
- copying, 131
- Core, **7**, **168**, 177–184, 187, **216–217**
- correctness, 162, 173, 226, 288
 - theorem, 60
- correlation between effectiveness and opportunity, 193
- Correspondence Lemma, **38**, 107, **162**, 288
- cost, 149, 151, 153
 - of analysis, 206
- cost-centre profiling, 192*fn*
- covariant, **44**, 44*fn*, 79, 118, 146
- creation site, *see* usage propagation
- cross-checks, 192
- cryptarithm2, 190, 199, 200, 205
- curried function, 64, 70, 74, 77, 78, 181, 273
- cylindrical constraint system, 124
- dangling pointer, 12
- data, **129**
- data constructor, *see* constructor
- data flow analysis, **15**, 16
- data system, 156*fn*
- data type, *see* algebraic data type
- data types, *see* algebraic data types
 - (►-DATA-ALL), 183, 195, 197
 - (►-DATA-ALL-BAD), **148**, 153, 183
 - (►-DATA-CLEAN), **152**
 - (►-DATA-EQUAL), **150**, 159, 164, 169, 183, 197
 - (►-DATA-FULL), **151**, 153, 183, 195, 197
 - (►-DATA-MANY), **148**, 183
 - (►-DATA-RESTR), **153**
- datum, **128**, 135*fn*, 138–142
- dead binding, 34, 51
- decidability, 77
- deconstruction, 128, 130
- deeply unlifted type, **180**
- DEFAULT, 168, **178**
- definition site, 65, 174
- definitions, **237–250**
- deforestation, 5, 77*fn*, 138, 186, 188, 210
- delta progress, 284
- demand, 5, 9, 10, 36, 48–50, 65, 69, 83, 179, 181, 192, **254**, 259, 272, 273
- demand annotation, 255, **266**
- demand/use distinction, 254
- dependency, 75, 77, 79, 95, 99, 100, **103**, 111, 113, 121, 135, 181
- depth subtyping, 165
- derivation, 44, 87, 275
- Design Patterns, 4
- design space, 137, 138, 150, 173, 224
- destruction, *see* deconstruction

- destructor function, 78
- development, **225**
- dictionary, 153, 183, 197
- direction of constraints, 99*fn*
- dissertation, *see OED*, thesis, sense 5
- dollar, *see* accounting method
- don't care, *see* usage annotation, irrelevant
- duality, 50
- dummy usage annotation, 169
- duplication, 182

- Edison, 183
- effect-based analysis, 16
- effectiveness, 19, 115, 190, **192**, 193, 200, 204, 213, 229
 - comparison of analyses, 195
 - fundamental limitation of, 211, 213
- efficiency, 18
- elephant, 180
- ELIX*₂, 272
- ELX*, 255
- encoding of constructor, *see* constructor, encoding of
- entailment, 55, 118, 119, 123, **291**
- environment, 28, 85, 90, 101, 172, 266, 267
 - initial, 89
 - manipulation of, 68, 72, 140*fn*, 253, 254, 267, 272
- environment variable, 112
- environments
 - separated, 72
- equal
 - constraint, 55
- Equal (analysis parameter setting), 197
- equivalence class, 105
- erasure, **31**, 38, 44, 79, 81, 106*fn*, 131, 182
- erasure semantics, 83, 224
 - without erasing the types, **81**
- eta-expansion, 187
- evaluation
 - under abstractions, 82, 131
- evaluation context, 26, 29, 33, 257, 276, 284, **310**
- evaluation order, 33, 34, 48, 48*fn*
- evaluator, *see* abstract machine
- example
 - of demand/use typing, 271
 - of exact use and demand, 261
- executable language, **29**
 - actual, 31
- execution time, *see* run time
- existential constructor, **187**
- existential quantifier, *see* hiding operator
- explicit term form, 81
- explicit typing, 27, 44, 141
- export, 53, 56, 65, 92, 113
- export list, 184
- exported binding, 173, 204, 205, 213–215
- expressivity, 164
- extended type system, 310
- extensions, 229, 251
 - required, 213

- factorial, 271
- false assumption, 182
- fft2*, 190, 197–199, 204, 214
- final pass
 - deferring, 215
- Fix, 158
- fixed point, 104
- FL*₀, 129, 168
- flags, 189
- flip*, 78
- FLIX*₀, **131**
- FLIX*₁, 269
- floating, 199, 215, 254
- floating in, **11**
- floating out, *see* full laziness
- floating transformation, 8, 83, 173, 188
- flow analysis, 13, 14, 117, 123, 272
- foldr*, 210
- foldr/build*, 186, 188
- folklore, 10, 128
- forbidden variable, **90**, 94*fn*, 99–101, 103, 104*fn*, 112
- forcing, 104, 105, 112, 121
- free occurrence in abstraction, *see* occurrence, free in abstraction
- free usage variable, 82
- free variables, 48
- fresh name supply, 172
- fresh usage variable, 87, 174
- fresh variable, 53, 56, 61*fn*, 108*fn*, 124, 163*fn*, 183, 294*fn*, 307*fn*
- FreshLUB*, 161
- Full (analysis parameter setting), 195
- full language, 83, 127, 225
- full laziness, 7, **12**, 13, 115
- full source language, 129
- full-scale language, 212

- function argument, 44, 67, 70, 266
- function result, 44, 67, 266
- function type, 44
- functional language, *see* lazy functional language
- fusage-heavy**, 173, 189
- fusagesp**, 189
- fusagesp-dconmaxcount_n**, 183, 189
- fusagesp-dconmoden**, 183, 189
- fusagesp-moden**, 173, 189
- fusagesp-oneshotmoden**, 189
- fusagespec**, 189
- fusion, 158, *see also* deforestation
- future use, 67*fn*, 99, 109
- future work, **228**
- game tree, 5
- gamteb, 191
- garbage collection, 12, 13, 15, 16, 39, 118, 149, 191, 193, 200
- generalisation, 89, 92, **92–106**, 109, 111, 112, 114, 117, 121, 128, 174, 224, *see* usage abstraction
 - Damas–Milner, 89
 - effect of, 85
 - restriction of, 85, 173, 204, 213, 214
- generalisation step, 89
- generalised type, 135
- generalised variables, 92
- GHC, *see* Glasgow Haskell Compiler
- Girard–Reynolds, *see* polymorphic lambda calculus
- Glasgow Haskell Compiler, **6**, 13, 18, 19, 81*fn*, 83, 114, 115, 168, 184, 191, 213, 225, 226, 229, 252, 257, 273
 - specific constructs, 184–187
- global analysis, 19, 21
- global constraint set, 66
- global property, 113
- global well-formedness constraint, 141
- “go wrong”, 4, 17, 31, 36, 60, 81, 107, 128, 130, 131, 287
- “good programs have small types”, 110
- goodness ordering, 18, **52**, 53, 57, 58, 61, 66, 68, 69, 71, 87, 89, 92, 96, 101, 114, 118, 119, 159
 - covariant of, 52, 71
- graph reduction, 8
- GRIN, 13
- ground annotation, 56, 95
- ground variable, 100, 103
- guard \triangleright , 257, 269, 272
- Hack (analysis parameter setting), 204
- Hack+Anal (analysis parameter setting), 204
- Halting Problem, 17
- Haskell, 3, 6, 65, 77, 128, 145, 157, 159, 190
- Hasse diagram, 96, 105
- heap, 9, **33**, 34, 39, 82, 180, 254, 275
- Heavy (analysis parameter setting), 197, 206
- heuristic, 66, 104, **111**, 228
- hiding operator, 69, 124
- HM(X), 123
- hole, 177, 276
- Holy Grail, 114
- Hope, 128
- HOT, 4, 16
- hyperfunctions, 158
- ideal, 95*fn*
- identifier, 168
- identity function, 65, 74, 77, 79
- idInt*, 93, 95
- if0, **27**, 34, 47, 48
- I.H., 275
- implementable, 19
- implementation, 115, 167–215, 224, 226, 229, 257
 - chronology, 168–171
 - necessity of, 224
- implementation experience, 212
- implicit parameters, 168
- implicit polymorphism, 89
- imported types, 169
- impredicativity, 89, 114
- improvement
 - of implementation, 214
- in a, *see* passages
- incomparable, 74, 86
- indirection, 192
- induction, 144, 145
- inference, 69, 93, 99*fn*, 117–119, 128, 150, 159, 163, 172, 173, 211, 225, 227, 309
 - extra passes of, 173
 - passes, 115
 - sequencing of, 173
 - type, 27
- inference algorithm, 18, 53–58, 61, 71, 74, **86–92**, 103, 224, 272

- inference rule, 67
- infinite proof tree, 144
- infinite regress, 180
- information, 75, 85
 - loss, *see* approximation
- information flow, *see* usage propagation
- initial algebra semantics, 128
- initial configuration, **34**
- inlining, **11**, 12, 66, 115, 173, 188, 198, 206, 215, 223, 272
 - cross-module, 215
- instances, 75, 95, 96, 181
- instantiation, 66, 77, 79, 83*fn*, 87, 89, 91, 92, 95, 121, 174, 176, 182, *see* usage application
- implicit, 128
- instantiation constraints, 120
- instrumentation, **31**, 81, 106, 131
- instrumented executable language, 131
- integrate, 190, 197
- interesting, *see* usage variable, boring/interesting
- interface file, 184, 186
- interleaving
 - constraint building, 89
- intermediate code, 227
- intermediate data structure, 138, 149, 151, 165
- intermediate language, 4, 7, 177, 180
 - typed, 16
- intermediate variable, 94*fn*
- internal variable, **90**, 94*fn*, 99, 100, 104, 105
- interval annotation, 39, 69
- intuitionistic, *see* linear
- isomorphism, 145, 155, 156
- ISWIM, 3
- IT_0 , **31**, 36, 161
- IT_1 , 53, 94
- IT_2 , 86, **159**
- join point, 188
- kernel, 118
- kind, *see* sort
- kitchen sink, 277
- kiwi, 62*fn*
- Kleene–Mycroft iteration, *see* polymorphic recursion
- L_0 , **27**, 36
- lambda abstraction, 141
- lambda calculus
 - Church’s, 3
- lambda cube, 168
- lambda usage, *see* usage, of function
- languages, **231**
- lattice, 91, 95, 96, 100, 123
- laziness, 184
 - necessary, 210, 213, 214
- lazy constructor, 130, 135*fn*
- lazy evaluation, **4**, 8, 14, 39, 71
- lazy functional language, 1, 3, 6, 70, 71, 223
- “lazy” lambda calculus, 5*fn*
- lazy list, 157
- lazy state threads, 114
- letrec, 7, 269
- libraries, 193*fn*, 204, 204*fn*
- library function, 77
- lift, 191, 198
- lifted type, **180**
- lifting, 85, 266
- linear, 68, 72, 253, 267, *see also* affine
- linear logic, 13, 48, 70, 270
- linearity, 14, 273
- LISP, 3
- List, 154
- list (algebraic data type), 137, 139, 141, 210, 213
- literal, 29, 33, 47, 141, 179
- LIX_0 , **29**
- LIX_1 , **42**, 83
- LIX_2 , **78**
- $LIXC_0$, **33**, 37
- local analysis, 19
- local property, 113
- lowlighted text, 78
- lowlighted text, 31
- LX , **31**
- main*, 177
- mandel, 191, 198
- manifest function, 188
- many times, 10, 43, 47, *see also* thunk, updatable
- map*, 171
- maximal applicability, 21, 63, 67, 74, 77, 92, 93, 95, 96, 113, 150, 182
- maximisation, 52, 53, 56, 68
- maximum, 48
- measurement, 9, 19–20, 69, 115, 176, 189–193, 213, 223, 224
- misleading, 10, 211

- metric, *see* measurement
- ML, 3, 16, 65, 85, 87, 89, 99, 110, 114, 117, 120, 128, 157, 159
 - type inference, 110
- module, *see* separate compilation
- monad, 172
- Mono (analysis parameter setting), 197
- monomorphic, 41–72, 81, 94, 128, 264
- monomorphic analysis, 14, 169, 197, 214, 226
- monomorphic language, 89
- monomorphic type, 74, 89
- monomorphic usage analysis, 74
- monomorphism, 99, 113
- monotonicity, 272
- Moore family, 52
- most general, *see* principal
- multiple application, *see* adjacent lambdas
- multiplicative, 267*fn*
- multiplier, 190, 205
- multiset, 55*fn*
- mutator time, 200

- needed, 1
- negative, *see* polarity
- negative constraint, 272
- negative occurrence, 79
- nested data type, *see* non-regular data type
- nesting, 64, 110, 164
- newtype, 168, 179, 185
- nfib*, 225
- Nil*, 210
- NoFib*, 10, 19, 115, 190, 193, 213
 - too fast, 197
- non-regular data type, 145, 159, 165
- None* (analysis parameter setting), 197, 204
- nonrestrictivity, 60, 107, 163
- nonvariant, 182
- not used, *see* zero usage
- notation, 231–236
- Note*, 168, 178
- nullary constructor, 178, 179, 182

- O'Haskell, 115, 120, 165
- object-oriented languages, 121
- observability, 71
- observational equivalence, 119
- observational subtyping, 118
- occurrence, 44, 50, 51, 68
 - free in abstraction, 47, 48, 68, 72, 83, 91, 135, 140, 188
 - in scope, 47, 48, 48, 55, 72, 83, 135, 138, 140, 142, 174, 178
 - positive (negative), *see* polarity, of occurrence
 - syntactic approach insufficient, 253, 254, 267
- occurrence analyser, 188
- odd and even lists, 156
- one-shot lambda, 11, 11, 29, 115, 173, 187, 192, 198–200, 215, 229, 254
- open source, 168
- operational semantics, 7, 17, 20, 31, 36, 39, 42, 59, 79, 81–83, 106, 131, 161, 162, 178, 224, 257–261
- operational significance, 131, 135
- opportunity, 190, 192, 193, 200, 211, 213
 - static, 214
- optimisation, 6, 7, 10, 11, 83*fn*, 113, 153, 173, 184, 187–189, 192, 200, 225, 253
 - misguided, *see* simplifier, making bad choices
- optimisation pass
 - sequencing of, 173, 176
- optimising compiler, 168
- order of evaluation, 15
- outermost annotation, *see* topmost annotation
- outlier, 195
- overall usage, 139, 139, 140, 146, 148, 155, 183, *see also* topmost annotation

- pair (algebraic data type), 138, 143
- parameterisation
 - of algebraic data type, 138, 139, 143
- parametric polymorphism, 131
- parametricity, 134
- parsers, 157
- partial application, *see* curried function
- passages
 - twisty little
 - maze of, *see* all alike
- passes
 - multiple, 197
- pathological, 254*fn*
- performance, 19, 64, 169, 183, 184, 190–215, 224, *see also*

- measurement
- permanent indirection, 192
- permits \leq , 256
- pessimisation, **56**, 63–65, 67, 87, 92, 161, 179, 184, 215
- pessimistic annotation, 171, 181
- placeholder, 171
- plus3*, 78, 96
- pointer, 180
- poisoning, 14, 15, **50**, 65, 74, 112–114, 158, 173, 174, 224, 225
- polarity, **44**, 79, 95, 99, 99*fn*, 103, 105, 112, 118, 182
 - in environment, 101*fn*
 - of annotation position, 67, 75, 92, 93, 179
 - of occurrence, 139, 146, 158
 - of position, 148, 150, 181
- polymorphic case, 178
- polymorphic lambda calculus, 7, 116, 128, 168
- polymorphic recursion, 91, 123, 159
- polymorphic usage analysis, 65, 169
- polymorphism, 16, 93, 106, 224
 - vs. subtyping, 77
 - beyond ML, 114
 - constrained, 14, 15
 - extra, 75
 - rank-1, 109, 114
 - rank-2, 114
 - redundant, 75
 - related work, 116
 - type, 14, 42, 128
- polyvariance, 16, 117, 123, 176
- positive, *see* polarity
- positive occurrence, **79**
- possibly used many times, *see* many times
- power, 224, 225, 229
- powerset, 255
- powertype, 95*fn*
- pragma, 184
- pragmatic subtyping, 120
- precision, 149, 152, 153, 165
- predicativity, 89
- predictability, 66, 77
- primitive operation, *see* primop
- primitive ordering, 42, **43**, 50, 68, 70, 257
- primitive recursion, 210, 213
- primitive reduction, 83
- primitive transition relation, 259, *see* transition relation, primitive
- primitive type, 179
- primop, **27**, 29, 33, 34, 40, 47, 180–182
 - partially saturated, 29, 37
- principal, 93, 95, 96, 113, 120
- principal type, 18, 66, 68–70, 74, 75, 77, 149
- principal type scheme, 128
- principal typing, 18, 61
- problem, **223**
 - monomorphic analysis, 64, 74
- problems, 115
- product \cdot , 256, 267, 272
- production compiler, 70, 167, 212, 224, 227
- profiling, *see* ticky-ticky profiling
- program analysis, **15**
- programs
 - all reasonable, 190
 - chosen, 190
- progress, **20**, 161, 285
- Progress Lemma, **37**, 59, 106
- projection, *see* usage projection types
- proof, 18, 20, 59–64, 81, 82, 106, 161–164, 275–311
- pruning, 284
- pseudo-completeness
 - of inference, 309
 - theorem, 63
- puzzle, 191, 198
- Qoheleth, 230
- quadratic behaviour, 91*fn*
- quasi-linear type, 15, 48*fn*, 270
- queens, 149, 190, 191, 209–210, 213, 225
- rank-1 usage polymorphism, 89, 171
- reachability, 57*fn*, 120
- real functional language, 224
- recursion, 150, 151, 154–159, 184, 255, 272
 - indirect, 157
 - mutual, 27, 50, 156, *see also* you
 - negative, 157, 158
 - non-regular, 159, 270
 - non-uniform, 158
 - self-, 154
 - type, 139, 141, 143, 144
 - non-regular, 145
- recursive binding group, *see* binding group
- recursive data type, 149, 170

- recursive occurrence, 99*fn*
- reduction, *see* transition relation
 - deterministic, 288
- redundant, 95
- reference count, 69
- regular data type, 145, 165
- regular tree, 156, 158
- regular tree grammar, 119
- repetition, 198
- representative variable, 104
- reptile, 191, 198
- restriction
 - of generalisation, *see* generalisation, restriction of
 - of number of parameters, 183, 195, 197
 - of use of rule, 55
 - rule location in derivation, 87
- results, 169, **193–214**, 224, 225
- Reynolds–Girard, *see* polymorphic lambda calculus
- ρ -type, **266**
- Rice’s Theorem, 17*fn*
- root-normal proof tree, 276
- rose tree, 147, 151, 152, 157
- RTS, 187, 191, 192
- rules, 184, **186**, **237–250**
- run time, 19, 197–199, 214
- runtime system, 227
- satisfiable, **291**
- Scheme, 118
- scoping, 79, 81, 85, 91
- scrutinee, **130**, 131, 142, 178, 270
 - annotation, 142
- SECD, 39
- selector thunk, 192
- semantic subtyping, 118
- semantics, 17
- separate compilation, 19, 21, **21**, 53, 56, 63, 64, 67, 74, 87, 177, 184, 200, 215
- separation between creation and use, 137–139
- seq, 36, 253, 255, 257, 269
- set-based analysis, 118
- sharing, 12–14, 48, 78, 137, 139–142, 198, 212, 224
 - of constraints, 120
- sharing constraint, 140, 270
- short-circuit binding, 254
- σ -type, 43, 79, 85, 106, 134, 135, 169, **264**
- sign negation, **81**
- signature, 21, 63, 67
- simple, 190, 198, 199, 206
- simple polymorphism, 73, **75–77**, 89, 96, 224, 226, 228, 264, 272
 - vs. O’Haskell, 120
 - applications, 111
- simple-polymorphic language, 89
- simple-polymorphic type, 121*fn*
- simplification, 118
- simplifier, 8, *see also* optimiser
 - making bad choices, 197–199, 214, 215, 229
- Simula-67, 131
- single component data type, 153
- single pass, 91*fn*
- size
 - of code, 206
 - of module, 200
- small types, 110
- small-step semantics, 20
- soft typing, **20**, 60, 93, 107, 150
- solution, 52, 55, 57, 58, 61, 71, 87, 92, 109, 117, 124, **224**, **291**
- solver, 103, 121
- sorcerer’s apprentice, 155
- sorts
 - two, 43, 79
- soundness, 13–15, 18, 20, 35, 36, 49, 50, 59, 61, 69, 75, 81, 85, 91, 99, 106, 107, 119, 141, 142, 162, 164, 173, 179, 181, 182, 224, 271, 275, 287
 - lemma, 37
 - of inference, 293, 307, 309
 - theorem, 38, 59, 63
- source language, 17, **27–28**, 227
- source polymorphism, *see* type polymorphism
- source type, 169, 171
- space-safety, 12, 39, 272
- specialisation, 7, 114, 121, 169, 171, 174–176, 184, 195, 204, 206, 213, 214, 226, 229
 - cascading, 176
 - versus generalisation, 205
- splitting, *see* polyvariance
- ST hack, 188, 200
- stable pointer, **180**, 181
- stack, 9, **33**, 39

- standard libraries, 115
- state, 181
- state threads, 189
- STG, 8, 9, 115, 169, 177, 178, 187, 188, 227
- strict let, 36, 178, 253
- strictness, 4, 6, 7, 14, 48, 70, 119, 131, 177, 180, 184, 229, 253–254, 257
- strictness analysis, 251, 272, 273
- strictness/absence analyser, 257
- stripping, 29, 37, 39, 44, 59, 131
- strong typing, 3
- strongly-connected component, 89*fn*, 177
- structure
 - of typing, 89
- stuck, 40
- subject reduction, 20, *see* type preservation
- substitution, 142, 171, 174–176, 185, 275, 281, 284, 290
- subsumption, 15, 45, 47, 48, 70, 75, 83, 86, 94, 99, 100, 116, 117, 225, 269, *see also* subtyping
- subtract \ominus , 256
- subtype, 78, 117, 118, 179
- subtype ordering, 105
 - contravariance of, 52, 71
- subtyping, 14, 15, 49, 50, 64, 66, 68, 70, 86, 89, 119, 135, 180, 181, 269
 - vs. polymorphism, 77
 - algebraic data type, 143–147, 165
 - related work, 116
 - semantic vs. syntactic, 95
 - structural, 86, 95, 117
 - tree model of, 165
 - type constructor, 297
- sum (algebraic data type), 143
- sum of uses, 252, 270
- sumdown*, 100
- supercombinators, 13
- suspension, 9
- syntactic occurrence, *see* occurrence, in scope
- syntactic proof, 20
- syntax-directed, 56, 87, 161
 - not, 89
- System F , *see* polymorphic lambda calculus
- System F_ω , 168
- \mathcal{T}_0 , 38
- tag, 128–130, 180
- tagged sum, *see* tag
- tail recursion, 340
- target language, 18, 42, 81
- τ -type, 43, 79, 85, 106, 134, 135, 139, 169, 264
- term form, 85
- terminal configuration, 26
- termination, 34, 39, 59, 106
- test suite, *see* NoFib
- theorem prover, 210
- theorems for free, 131
- thesis
 - how to read, 22
- thunk, 9, 44, 50, 81, 114, 115, 169, 173–176, 178, 180, 191, 200, 211, 225, 253, 255
 - consumer-updated, 50
 - dynamic, 211, 214
 - dynamically-allocated, 209
 - in datum, 135*fn*
 - manual marking of, 211
 - re-entrant, 187
 - removal of, 197
 - self-updating, 50
 - single-entry, 11, 29, 95, 187, 191, 192, 209–211
 - top-level, 211
 - updatable, 115, 187, 192
 - used at most once, 209
 - used-once, 19, 193, 215
- ticky-ticky profiling, 191
- tiered types, *see* sorts
- TIM, 39
- top level, 187
- top-level binder, 173, 204, 205
- topmost annotation, 43, 47, 48, 75, 79, 83, 85, 113, 138–141, 148, 150, 175, 178, 179, 182, 186, 270
- tortoise, *see* turtle
- toy language, 225
- transition relation, 33, 259
 - primitive, 33, 34
- transitive closure, 103
- TransitiveClosure*, 105, 292
- translation of data type, *see* annotation scheme
- trivial bounds, 125
- trivial translation, 31, 35, 39, 47
- tuning, 189
- turtle, 180, 180*fn*
- twice*, 96, 121

- twist-list, 158
- TyNote, 169
- type, 3, 27, 43, 47
 - erasure, *see* erasure
 - recursive, *see* recursion, type
 - uninhabited, 141
- type abstraction, 128, 129, 134–135, 161
 - erasure of, 131
- type annotation, 27, 120
- type application, 128, 129, 135, 172, 185
 - erasure of, 131
- type argument, 134, 141, 146
- type class, 168
- type constructor, 129, 145, 161, 165, 171, 181, 185, 266
- type parameter, 129
- type polymorphism, 77, 83, 123, **128**, 129, 134–135, 161, 224
- type preservation, 224, 269
- type scheme, 87, 89, 117
- type signature, 114
- type system, 264–271
- type variable, 129, 134–135, 171
- type-based analysis, 16, 18, **18**, 41, 111, 123, 226
- type-directed, 7
- type-erasure semantics, *see* erasure semantics
- typeability, 150, 163, 182
- typecase, 131
- typechecker, 169, 227
- typed intermediate language, 77, 128, 168
- typings
 - sample, 77
- u in $(T \dots)^u$, *see* overall usage
- unary constructor, 153
- unboxed data type, 180
- unboxing, 251
- undecidability, 17, 114
- unfolding, 184
 - of datatype, 155
- unification, 91, 99–101, 104, 105, 112, 117, 119–121, 151, 164, 181, 186
 - to resolve constraint, 78
- union-find, 57, 63, 110, 164
- uniqueness, 70
- uniqueness annotation, 69
- uniqueness propagation, 140
- uniqueness type, **69**
- uniqueness typing, 123, 152
- unlifted type, **180**, 181
- unsatisfiable, 57, 103
- update, **9**, 10, 34, 211, 253
 - avoidance, 9, 13, 19, 187, 197, 199, 200, 223, 252
 - in place, 13, 15, 69
 - measurement of, 9
 - wasted, 1, 9
- update flag, 26, 29, **29**, 35, 39, 43, **45**, 47, 50, 52, 56, 81, 83, 87, 107, 131, 141, 161, 224, 275
- update frame, **33**, 34, 36, 39, 187, 191, 192, 197, 257
- update marker, 68
- update marker check avoidance, 15, 39, 214
- update marker check intervals, 273
- update-in-place, 192
- usage, 10, 36, 40, 43, 44, 48, 50, 60, 83, 107, 179, 251, 272
 - vs. uniqueness, 70
 - inside datum, 135
 - of function, 178, 185, 187
- usage abstraction, 78, 81, **81**, **85**, 87, 89, 106, 123
- usage analysis, 1, 6, **11**, 26, 39, 41, 113, 137, 176, 224, 273
 - related work, 12–15
 - simple polymorphic, 73
- usage annotation, 43, **45**, 72, 81, 89, 134, 138, 140, 169, 178, 180, 185, 187, 224, 255, **264**
 - irrelevant, 180–182
 - static not dynamic, 211
- usage application, 78, 81, **81**, 85, 91, 106, 123
- usage argument, 141, 153, 170, 174, 176, 182, 183, 270
- usage generalisation, 87, 134
- usage information, 214, 229
 - accuracy of, 199
 - exploiting, 197
- usage interval analysis, 272
- usage kind, 170
- usage parameter, 139, **139**, 148, 149, 151, 159
- usage polymorphism, 77, 78, 121, 131, 181
 - arbitrary, 109
 - explicit, 81

- usage projection types, 171–172, 174, 185, 212
- usage propagation, 53, 70, 75, 137, 138, 140, 141, 179, 181
- usage quantification, 78, 79, 86
 - vs. subtyping, 95
- usage scheme, 171
- usage signature, 185
- usage site, 149
- usage specialisation, 185, *see* specialisation
- usage type, 79, 212
 - three-tier, 264
- usage variable, 53, 55, 57, 74, 75, **79**, 82, 83, 85, 86, 95
 - boring/interesting, 171, **175**
 - in scope, 82
- usage-interval analysis, 13
- usage-monotype, 87
- usage-variable-follows-type-variable, 114
- use, 69, 178, **254**, 261, 272, 273
- use site, *see* usage propagation
- use-once-don't-drag, 12, **39**, 40, 59, 70*fn*, 272
- used at most once, *see* at most once
- used many times, *see* many times
- user-defined algebraic data types, *see* algebraic data types

- Value*, 59, 106, 259*fn*, 287
- value, 33, 40, 68, 82, 83, 254
 - inside datum, 135
 - usage of, 47
- value annotation, 255
- value argument, 141
- (Value) rule, 141
- variability, 193, 206, 213
 - of effectiveness, 229
- variable
 - usage of, 47
- variance, **44**, 181, 215

- weak head normal form, 178, 267, 273
- weakening, 270, 277
- weaker typing judgement, 275
- well-annotation, **35**, 69
- well-founded induction, 308
- well-typed, 78, 87
- “well-typed programs do not. . .”, *see* “go wrong”
- well-typing, 37, 114, 123, 131

- well-typing rules, 16, 18, 28, 47, 52, 53, 83, 129, 135, 138, 141–143, 161, 162, 267–269
- WHNF, *see* weak head normal form
- work-safety, 12, 39, 272
- worker/wrapper transformation, 7, 252
- Wrong*, **35**, 59, 106, 259*fn*, 287

- Y2K, 16
- you, *see* are

- Zed, 51*fn*
- zero annotation, 252
- zero usage, 11, 15, 36, 213
- zero use, 257
- zig-zag, 96, 112

This thesis was set in Charter, 11pt on 13.6pt, using \TeX 2_ε with a customised version of the book class. Chapter headings are in Narkisim. Key packages used were amsmath, amssym, footmisc (thanks to Donald Arsenau for modifying this package to include footnote continuation marks), geometry, graphicx, Peter Sewell's \mrefs, pifont, Paul Taylor's Proof Trees, PSfrag, stmaryrd, and X_Y-pic. Building the thesis from sources also involves Bi \TeX , dvips, MakeIndex, MetaPost, Perl, pnm2ps, Tgif, and Xfig. Some graphics and data processing used Microsoft[®] Excel 2000 and CorelDRAW!, occasionally via rdesktop. Hugs and Graphviz's dotty were used for some figures in Chapter 4. Simon Marlow's nofibanalyse tool was used to process some of the data from test runs of the compiler. Editing was mostly done using Emacs, with some vi and cat, along with Xdvi and gv. CVS was used for version control. TrueType[®] font conversion by TTF2PT1. Development was done under Solaris x86 and Linux.