Guarded Impredicative Polymorphism

Extended Version

Alejandro Serrano
Jurriaan Hage
A.SerranoMena@uu.nl
J.Hage@uu.nl
Utrecht University
Utrecht, The Netherlands

Dimitrios Vytiniotis
Simon Peyton Jones
dimitris@microsoft.com
simonpj@microsoft.com
Microsoft Research
Cambridge, United Kingdom

Abstract

The design space for type systems that support impredicative instantiation is extremely complicated. One needs to strike a balance between expressiveness, simplicity for both the end programmer and the type system implementor, and how easily the system can be integrated with other advanced type system concepts. In this paper, we propose a new point in the design space, which we call guarded impredicativity. Its key idea is that impredicative instantiation in an application is allowed for type variables that occur under a type constructor. The resulting type system has a clean declarative specification — making it easy for programmers to predict what will type and what will not —, allows for a smooth integration with GHC's OutsideIn(X) constraint solving framework, while giving up very little in terms of expressiveness compared to systems like HMF, HML, FPH and MLF. We give a sound and complete inference algorithm, and prove a principal type property for our system.

CCS Concepts • **Theory of computation → Type structures**;

Keywords Type systems, impredicative polymorphism, constraint-based inference

This is an extended version of Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. *In PLDI '18: ACM SIGPLAN Conference on Programming Language Design and Implementation, June 18 – 22, 2018, Philadelphia, PA, USA.*

1 Introduction

Type inference for impredicative polymorphism is a deep, deep swamp. There is a dense literature of papers presenting type systems for impredicative polymorphism [1, 8, 9, 22, 23]. Alas none of them quite worked well enough to be deployed in a production compiler: either the system was too complicated for users to predict its behaviour, or it was too complicated to implement, or sometimes both.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
2018. ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192389

Yet it is tantalising: what is wrong with the type $[\forall a.\, a \to a]$, a list of polymorphic functions? If we have $xs :: [\forall a.\, a \to a]$, why can't we write $(head\ \ xs)$? Not every programmer wants such types but, when they do, it is very annoying that they are disallowed, apparently for obscure technical reasons. That is why we keep trying.

So what is the problem? The difficulty is that to accept $(head\ \ xs)$ we must instantiate the type variable of *head*'s type with a polymorphic type. More precisely, since $head :: \forall p.\, [p] \to p$, we must instantiate p with $(\forall a.\, a \to a)$. This instantiation seems deceptively simple, but in practice it is extremely hard to combine with type inference. We respond to this challenge by making the following contributions:

- Every attempt to combine type inference with impredicativity involves a design trade-off between complexity, expressiveness, and annotation burden. Our key contribution is a new trade-off, which we call *guarded instantiation* or GI (Section 2).
  GI is simple: simple for the programmer to understand (Section 2.1-2.3), simple in its declarative specification and metatheory (Section 3); and simple in its implementation (Section 4). We do not extend the syntax of (System F) types in order to provide a specification of the type system (unlike previous work [1, 9, 23]), nor do we introduce new forms of annotations [19] or side-conditions that require principal types [8].
- Despite GI's relative simplicity, it accepts without annotation particularly celebrated and practically important examples, such as *runST* $ *e* (Figure 2).
- We give a declarative type system for GI for a small core language, highlighting the key ideas of our system (Section 3). Then we show simple extensions to handle a more full-fledged language, including type annotations (Section 3.4), **let** bindings (Section 3.5), and pattern matching (Appendix A). The system has a notion of *principal type* akin to Hindley-Milner type systems, that is, existence of a monomorphic substitution mediating between types. In particular, impredicativity is never guessed in GI (Section 3.6). The resulting system can express any System F program.

- We provide a sound and complete *inference algorithm* (Section 4) for GI, based on *constraints*. The type inference algorithm is a modest extension of the constraint-based algorithm already used by GHC. Type-correct programs can readily be elaborated into System FC, GHC's intermediate language, without extensions. Our inference algorithm scales readily to handle GADTs, type classes, higher kinds, type-level functions and other type system features.
- We provide a prototype implementation of the whole system, integrated with Haskell's type classes.

Type inference for impredicativity is dense with related work, as we discuss in Section 6. A small but useful contribution to a dense field is Figure 2, which presents key examples from the literature and shows how each major system behaves.

2 The key idea: intuition and examples

We begin with an informal introduction to GI, which we make fully precise in Section 3. In this discussion we make use of functions defined in Figure 1.

2.1 Exploiting the easy case

What is hard about typing ($head\ \ ids$)? Nothing! In $head$'s type the variable p appears under a list type constructor. Given the type of ids, $[\forall a.\, a \rightarrow a]$, it is plain as a pikestaff that we must instantiate p with $\forall a.\, a \rightarrow a$. The difficulty comes when we have a "naked" or "un-guarded" type variable in one of the arguments, such as p in $single :: \forall p.\, p \rightarrow [p]$. Now if we examine ($single\ \ id$), it is not clear whether we should instantiate p with $\forall a.\, a \rightarrow a$, or with $Int \rightarrow Int$, or some other monomorphic type.

In fact, ($single\ \ id$) does not have a most general type. It has both of these two incomparable types $\forall a.\, [a \rightarrow a]$ and $[\forall a.\, a \rightarrow a]$. To make things worse ($single\ id$) is a perfectly typeable Hindley-Milner program (with the former type) so we must allow this type. But to support impredicativity, we must also allow the latter. But under which conditions?

Our approach is to exploit the common case. We focus on n-ary applications ($f\, e_1 \ldots e_n$). It is more conventional to deal with binary applications, but in fact n-ary applications (unencumbered with intervening **let** or **case** constructs) are wildly dominant in practice, and we can get much better typing by treating the application all at once. Then we adopt this rule to type such n-ary applications:

> The **Instantiation Rule**. *Given a n-argument call $f\ e_1 \ldots e_n$ to a function $f :: \forall a_1 \ldots a_p.\, \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \phi$, where $m \leqslant n$, a type variable a_i may be instantiated to:*
>
> 1. *A polymorphic type ϕ, if a_i appears under a type constructor in one of the σ_j (note that we only take in consideration as many types as arguments given). In this case we say that a_i is* guarded *in the type of f.*
> 2. *A top-level monomorphic type μ (see Figure 3), if a_i appears in any of the σ_j at all.*
> 3. *A fully monomorphic type τ, otherwise.*

This rule is carefully crafted. To illustrate, consider these examples (consult Figure 1 for the types):

- ($map\ poly$) is OK because in the type of $map :: \forall a\ b.\, (a \rightarrow b) \rightarrow [a] \rightarrow [b]$, both type variables appear under the type constructor ($\rightarrow$) in the first argument. We can instantiate both to ($\forall a.\, a \rightarrow a$), by case (1) of the Rule, as required to match the type of $poly$.
- ($single\ \ ids$) is OK because we can instantiate $single :: \forall a.\, a \rightarrow [a]$ with the top-level-monomorphic type $\forall b.\, [b]$, using case (2) of the Instantiation Rule.
- ((:) id) is a *partial application* of (:). So although a appears guarded in the second argument of $(:) :: \forall a.\, a \rightarrow [a] \rightarrow [a]$, in this call we can only take advantage of the first argument (see $m \leqslant n$ in the rule.). Hence a can only be instantiated by a top-level monomorphic type, by clause (2) of the rule. If we add a second argument in the call, such as ((:) $id\ \ ids$), instantiation may be polymorphic since the second argument is now taken into consideration.
- ($id\ poly\ (\lambda x.\, x)$) is a tricky one. Here id is applied to two arguments although its type, $\forall a.\, a \rightarrow a$, apparently only has one; moreover the type of id's second argument must be polymorphic, and the ($\lambda x.\, x$) must be generalised. But the Instantiation Rule says that this application is OK: the type of $poly$ is $(\forall a.\, a \rightarrow a) \rightarrow (Int, Bool)$, a top-level monomorphic type. Thus the instantiation of id to that type is allowed by clause (2) of the Instantiation Rule.

The Instantiation Rule is still informal (which we remedy in Section 3) but it is very helpful to have a rule of thumb to explain to a programmer what will and will not work.

2.2 Ignoring the context of a call

Notice that the Instantiation Rule *takes no account of the context of the call.* For example, consider $ids +\!\!+ single\ id$. We know that the result of ($single\ \ id$) must be $[\forall a.\, a \rightarrow a]$, given that ids has the same type, and you might think that would be enough to fix the instantiation of $single$. But not in GI! The swamp beckons, and we stay on dry land.

Moreover, the Instantiation Rule allows the programmer to understand impredicativity in a simple bottom-up way. For example, consider the expression ($map\ \ head\ \ (single\ \ ids)$) (Figure 2). In GI, the types for $head$ and $single\ \ ids$ are instantiated independently, so we never need to consider the *interaction* between the arguments. This modularity pays off in the metatheory too.

There is a price to pay, however. As a degenerate case, a function application without any arguments – that is, a variable – may only instantiate *fully* monomorphically – no polymorphism, even if it appears under a type constructor.

head	$:: \forall p. [p] \to p$	*id*	$:: \forall a. a \to a$	*ids*	$:: [\forall a. a \to a]$
tail	$:: \forall p. [p] \to [p]$	*inc*	$:: Int \to Int$	*map*	$:: \forall p\ q. (p \to q) \to [p] \to [q]$
[ ]	$:: \forall p. [p]$	*choose*	$:: \forall a. a \to a \to a$	*app*	$:: \forall a\ b. (a \to b) \to a \to b$
(:)	$:: \forall p. p \to [p] \to [p]$	*poly*	$:: (\forall a. a \to a) \to (Int, Bool)$	*revapp*	$:: \forall a\ b. a \to (a \to b) \to b$
single	$:: \forall p. p \to [p]$	*auto*	$:: (\forall a. a \to a) \to (\forall a. a \to a)$	*flip*	$:: \forall a\ b\ c. (a \to b \to c) \to b \to a \to c$
(++)	$:: \forall p. [p] \to [p] \to [p]$	*auto′*	$:: (\forall a. a \to a) \to b \to b$	*runST*	$:: \forall v. (\forall s. ST\ s\ v) \to v$
length	$:: \forall p. [p] \to Int$			*argST*	$:: \forall s. ST\ s\ Int$

Figure 1. Type signatures for functions used in the text

Thus, the empty list constructor $[\,] :: \forall a. [a]$ cannot be assigned a type $[\forall a. a \to a]$. We describe how to loosen this restriction in Section 3.3.

2.3 Lambdas

In common with many other approaches to impredicativity, we take a conservative position on lambda-bound variables. Consider $g\ (\lambda f. (f\ \texttt{'x'}, f\ True))$, where $g :: ((\forall a. a \to a) \to (Char, Bool)) \to Int$. Since g can only be applied to a function whose argument is itself polymorphic, you could imagine that information being propagated to f and so the program could be accepted. In common with many other systems, we reject all programs that require a lambda-bound variable to be polymorphic, unless it is explicitly annotated:

> The **Lambda Rule**. *Every lambda abstraction whose argument is polymorphic must be annotated. Otherwise, the bound variable can only have a fully monomorphic type.*

By a "fully monomorphic type" we mean "no foralls anywhere". Nothing about guardedness here! In contrast, MLF requires an annotation only when the argument is used more than once in the body with different polymorphic types.

While the Lambda Rule deals with the arguments to lambdas, it says nothing about the return type. To get a polymorphic return type, an annotation needs to be provided. For example, for $\lambda(x :: \forall a. a \to a). x\ x$, GI infers the type $(\forall a. a \to a) \to b \to b$, and not $(\forall a. a \to a) \to (\forall a. a \to a)$. To get the latter type, we have to write $\lambda(x :: \forall\ a. a \to a). (x\ x :: \forall a. a \to a)$ instead.

2.4 Expressiveness

By treating n-ary applications as a whole, and taking guardedness from both the function type and the argument types, we can infer impredicative instantiations in many practically-useful situations. We summarise a collection of examples culled from the literature in Figure 2. This table also compares our system with others, but we defer discussion of related work to Section 6.

A celebrated example is the function $(\$) :: (a \to b) \to a \to b$. Haskellers use this function all the time to remove parentheses in their code, as in $(runST\ \$\ \textbf{do}\ \{\dots\})$ (the type of $runST$ is given in Figure 2). This call absolutely requires impredicative instantiation of the variable a in the type of ($). It is so annoying to reject this program that GHC implements a special, built-in typing rule for $f\ \$\ x$. Of course, that is horribly non-modular: if the programmer re-defines another version of ($), even with the same type, some programs cease to type check. In GI both type variables appear under the $(\to)$ constructor, so impredicative instantiation is allowed.

The lack of support for impredicative types is painful. For example, consider the following from Haskell's `lens` library:

$$\textbf{type}\ Lens\ s\ t\ a\ b = \forall f. Functor\ f \Rightarrow (a \to f\ b) \to s \to f\ t$$

Programmers think of a lens as a first-class value, and are perplexed when they cannot put a lens into a list or other data structure. With GI, many more lens-manipulating programs become well-typed.

One might worry about the order of quantifiers. Take:

$$f :: (\forall a\ b. a \to b \to b) \to Int \qquad x :: \forall b\ a. a \to b \to b$$
$$g :: [\forall a\ b. a \to b \to b] \to Int \qquad xs :: [\forall b\ a. a \to b \to b]$$

The application $(f\ x)$ is well-typed in GI, despite the differing quantifier ordering, because we compare f's argument type and x's actual type using *subsumption*; effectively we instantiate and re-generalise. In contrast, GI does not accept the application $(g\ xs)$, because under a list constructor we compare the types using *equality*. Happily, while *top-level* quantifiers (such as those for x) are invisibly inferred (with unpredictable ordering), *nested* quantifiers, such as those in g and xs's type, are never inferred but rather declared through a type signature. This makes accidental incompatibility vanishingly rare in practice, as we verify in Section 5.

3 Declarative specification

We first present a systematic description of the declarative specification of GI. We use the term declarative in the sense of not syntax-directed. After we have proven soundness and completeness for the constraint-based variant, the programmer can take this declarative specification – easier to understand but without a direct inference algorithm – as a basis to understand when and why annotations are needed.

3.1 Syntax

The syntax of the language is given in Figure 3. The language has some distinctive features. First, as discussed in Section 2, we deal with n-ary applications instead of binary ones. A lone term variable is treated as nullary application. Because of the

		GI	MLF [1]	HMF [8]	FPH [23]	HML [9]
A	POLYMORPHIC INSTANTIATION					
A1	$const2 = \lambda x\ y.\ y$	✓	✓	✓	✓	✓
	MLF infers $(b \geqslant \forall c.\ c \to c) \Rightarrow a \to b$, GI infers $a \to b \to b$.					
A2	*choose id*	✓	✓	✓	✓	✓
	MLF and HML infer $(a \geqslant \forall b.\ b \to b) \Rightarrow a \to a$, FPH, HMF, and GI infer $(a \to a) \to a \to a$.					
A3	*choose* [ ] *ids*	✓	✓	✓	✓	✓
A4	$\lambda(x :: \forall a.\ a \to a).\ x\ x$	✓	✓	✓	✓	✓
	MLF infers $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$, GI infers $(\forall a.\ a \to a) \to b \to b$.					
A5	*id auto*	✓	✓	✓	✓	✓
A6	*id auto′*	✓	✓	✓	✓	✓
A7	*choose id auto*	✓	✓	No	No	✓
A8	*choose id auto′*	No	✓	No	No	✓
A9	*f* (*choose id*) *ids*	No	✓	No	✓	✓
	where $f :: \forall a.\ (a \to a) \to [a] \to a$					
	GI needs an annotation on $id :: (\forall a.\ a \to a) \to (\forall a.\ a \to a)$ in the previous two examples.					
A10	*poly id*	✓	✓	✓	✓	✓
A11	*poly* $(\lambda x.\ x)$	✓	✓	✓	✓	✓
A12	*id poly* $(\lambda x.\ x)$	✓	✓	✓	✓	✓
B	INFERENCE OF POLYMORPHIC ARGUMENTS					
B1	$\lambda f.(f\ 1, f\ True)$	No	No	No	No	No
	All systems require an annotation on $f :: \forall a.\ a \to a$.					
B2	$\lambda xs.\ poly\ (head\ xs)$	No	✓	No	No	No
	All systems except for MLF require annotated $xs :: [\forall a.\ a \to a]$.					
C	FUNCTIONS ON POLYMORPHIC LISTS					
C1	*length ids*	✓	✓	✓	✓	✓
C2	*tail ids*	✓	✓	✓	✓	✓
C3	*head ids*	✓	✓	✓	✓	✓
C4	*single id*	✓	✓	✓	✓	✓
C5	*id* : *ids*	✓	✓	No	✓	✓
C6	$(\lambda x.\ x) : ids$	✓	✓	No	✓	✓
C7	*single inc* ++ *single id*	✓	✓	✓	✓	✓
C8	*g* (*single id*) *ids*	No	✓	No	✓	✓
	where $g :: \forall a.\ [a] \to [a] \to a$					
C9	*map poly* (*single id*)	No	✓	✓	✓	✓
	GI needs an annotation on $single\ id :: [\forall a.\ a \to a]$ in the previous two examples.					
C10	*map head* (*single ids*)	✓	✓	✓	✓	✓
D	APPLICATION FUNCTIONS					
D1	*app poly id*	✓	✓	✓	✓	✓
D2	*revapp id poly*	✓	✓	✓	✓	✓
D3	*runST argST*	✓	✓	✓	✓	✓
D4	*app runST argST*	✓	✓	✓	✓	✓
D5	*revapp argST runST*	✓	✓	✓	✓	✓
E	η-EXPANSION					
E1	*k h lst*	No	No	No	No	No
E2	$k\ (\lambda x.\ h\ x)\ lst$	✓	✓	No	✓	✓
	where $h :: Int \to \forall a.\ a \to a$, $k :: \forall a.\ a \to [a] \to a$, and $lst :: [\forall a.\ Int \to a \to a]$					
E3	$r\ (\lambda x\ y \to y)$	No	✓	No	No	No
	where $r :: (\forall a.\ a \to \forall b.\ b \to b) \to Int$					

Figure 2. Comparison of type systems

Skolem / rigid variables	$\mathbb{V}$	$\ni$	$a, b, c, \ldots$
Free type variables	$\text{ftv}(\sigma)$		
Type constructors	$\mathbb{T}$	$\ni$	$\to, \mathsf{T}, \mathsf{S}, \ldots$
Fully mono. types	τ	::=	$a \mid \mathsf{T}\ \overline{\tau}$
Top-level mono. types	μ, η	::=	$a \mid \mathsf{T}\ \overline{\sigma}$
Polymorphic types	σ, ϕ	::=	$\forall \overline{a}.\ \mu$
	$\overline{a}$ may be empty, $\overline{a} \subseteq \text{ftv}(\mu)$		
Term variables		$\ni$	x, y
Expressions / terms	e	::=	x
		$\mid$	$e_0\ e_1\ \ldots\ e_n \quad n \geqslant 0$
		$\mid$	$\lambda x.e$
		$\mid$	$\lambda(x :: \sigma).e$
		$\mid$	$(e_0\ e_1\ \ldots\ e_n :: \sigma)$
		$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$
Environments	Γ	::=	$\epsilon \mid \Gamma, x : \sigma$
Substitutions	θ, φ, π	::=	$[\overline{a \mapsto \sigma}]$
Sorts of variables	$\mathfrak{s}$	::=	$\mathfrak{u} \mid \mathfrak{t} \mid \mathfrak{m}$
	Sorts form a lattice, $\mathfrak{m} \sqsubset \mathfrak{t} \sqsubset \mathfrak{u}$		
Sort assignment	Δ	::=	$[\overline{a \mapsto \mathfrak{s}}]$
Bit	ω	::=	$\bullet \mid \star$
Vector of bits	$\overline{\omega}$	::=	$\epsilon \mid \omega, \overline{\omega}$

Figure 3. Syntax of the language

Lambda Rule (Section 2.3), we provide explicitly-annotated lambda abstractions, $\lambda(x :: \sigma).\ e$, to support lambdas whose bound variable must have a polymorphic type. Annotations and **let**s are treated in their own sections, we first focus on the core language with variables, applications, and lambdas.

Types (Figure 3) are classified by three "sorts", $\mathfrak{u}$, $\mathfrak{t}$, and $\mathfrak{m}$. *Polymorphic* types σ, ϕ, of sort $\mathfrak{u}$, have unrestricted polymorphism. *Top-level monomorphic* types, μ, η, of sort $\mathfrak{t}$, have no polymorphism at the top level, but permit arbitrary nested polymorphic types under a type constructor. Finally, *fully-monomorphic types*, τ, of sort $\mathfrak{m}$, have no trace of polymorphism. Fully monomorphic types correspond to monotypes in the Hindley-Milner tradition. These are the only types which can be assigned to un-annotated lambda-bound variables. We extend this notion to substitutions, and sometimes speak of fully monomorphic substitution to mean that the image of the substitution contains only types of that sort.

For a substitution θ, the image of a type variable a is denoted by $\theta(a)$, and similarly for sort assignments.

3.2 Typing rules

The typing judgment $\Gamma \vdash e : \sigma$ is given in Figure 4, along with some auxiliary judgments.

Rules Abs and AnnAbs concern lambda abstractions, and are straightforward. AnnAbs deals with a lambda $(\lambda(x :: \phi).\ e)$ where the user has supplied a type annotation ϕ: we

$$\boxed{\sigma \text{ respects } \mathfrak{s}} \qquad \boxed{\theta \text{ respects } \Delta}$$

$$\frac{}{\sigma \text{ respects } \mathfrak{u}} \qquad \frac{}{\mu \text{ respects } \mathfrak{t}} \qquad \frac{}{\tau \text{ respects } \mathfrak{m}} \qquad \frac{\forall a \in \mathrm{dom}(\theta), \theta(a) \text{ respects } \Delta(a)}{\theta \text{ respects } \Delta}$$

$$\boxed{\sigma \ \rhd^{\mathfrak{g}} \ \Delta}$$

$$\frac{\mu \ \rhd^{\mathfrak{g}} \ \Delta}{\forall \overline{a}.\, \mu \ \rhd^{\mathfrak{g}} \ \Delta \backslash \overline{a}}\ \textsc{ArgPoly} \qquad \frac{}{\mathsf{T}\,\overline{\phi} \ \rhd^{\mathfrak{g}} \ [\mathrm{ftv}(\overline{\phi}) \mapsto \mathfrak{u}]}\ \textsc{ArgGuard} \qquad \frac{}{a \ \rhd^{\mathfrak{g}} \ [a \mapsto \mathfrak{t}]}\ \textsc{ArgTyVar}$$

$$\boxed{\sigma \rhd^{\mathfrak{s}}_{\overline{\omega}} \Delta}$$

$$\frac{}{\mu \ \rhd^{\mathfrak{s}}_{\epsilon} \ [\mathrm{ftv}(\mu) \mapsto \mathfrak{s}]}\ \textsc{ArgsRes} \qquad \frac{}{a \ \rhd^{\mathfrak{s}}_{\omega, \overline{\omega}} \ [a \mapsto \mathfrak{m}]}\ \textsc{ArgsTyVar}$$

$$\frac{\mu \ \rhd^{\mathfrak{s}}_{\overline{\omega}} \ \Delta}{\forall \overline{a}.\, \mu \ \rhd^{\mathfrak{s}}_{\overline{\omega}} \ \Delta \backslash \overline{a}}\ \textsc{ArgsPoly} \qquad \frac{\sigma_1 \ \rhd^{\mathfrak{g}} \ \Delta_1 \qquad \sigma_2 \ \rhd^{\mathfrak{s}}_{\overline{\omega}} \ \Delta_2}{\sigma_1 \to \sigma_2 \ \rhd^{\mathfrak{s}}_{\bullet, \overline{\omega}} \ \Delta_1 \sqcup \Delta_2}\ \textsc{ArgsArrow}$$

$$\boxed{\phi \ \leqslant^{\mathfrak{s}}_{\overline{\omega}} \ \overline{\sigma}; \mu}$$

$$\frac{}{\mu \ \leqslant^{\mathfrak{s}}_{\epsilon} \ \epsilon; \mu}\ \textsc{InstMono} \qquad \frac{\phi_2 \ \leqslant^{\mathfrak{s}}_{\overline{\omega}} \ \overline{\sigma}; \mu}{\phi_1 \to \phi_2 \ \leqslant^{\mathfrak{s}}_{\omega, \overline{\omega}} \ \phi_1, \overline{\sigma}; \mu}\ \textsc{InstArrow} \qquad \frac{\mu \rhd^{\mathfrak{s}}_{\overline{\omega}} \Delta \qquad \theta \text{ respects } \Delta \qquad \theta\mu \ \leqslant^{\mathfrak{s}}_{\overline{\omega}} \ \overline{\sigma}; \mu}{\forall \overline{a}.\, \mu \ \leqslant^{\mathfrak{s}}_{\overline{\omega}} \ \overline{\sigma}; \mu}\ \textsc{InstPoly}$$

$$\boxed{\Gamma \vdash^{\mathsf{fun}} e : \sigma} \qquad \boxed{\Gamma \vdash^{\mathsf{arg}}_{\omega} e : \sigma}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \sigma}\ \textsc{VarHead} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash^{\mathsf{fun}} e : \sigma}\ \textsc{ExprHead} \qquad \frac{\Gamma \vdash e : \forall \overline{a}.\, \mu \qquad \overline{b} \notin \Gamma}{\Gamma \vdash^{\mathsf{arg}}_{\bullet} e : \forall \overline{b}.\, [\overline{a \mapsto \tau}]\mu}\ \textsc{ArgGen}$$

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}\ \textsc{Abs} \qquad \frac{\Gamma, x : \phi \vdash e : \sigma}{\Gamma \vdash \lambda(x :: \phi).\, e : \phi \to \sigma}\ \textsc{AnnAbs} \qquad \frac{\Gamma \vdash e_1 : \phi \qquad \Gamma, x : \phi \vdash e_2 : \sigma}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma}\ \textsc{Let}$$

$$\frac{n \geqslant 0 \qquad \Gamma \vdash^{\mathsf{fun}} e_0 : \phi \qquad \phi \ \leqslant^{\mathfrak{m}}_{\omega_1, \ldots, \omega_n} \ \sigma_1, \ldots, \sigma_n; \mu \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_1} e_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash^{\mathsf{arg}}_{\omega_n} e_n : \sigma_n}{\Gamma \vdash e_0\, e_1\ \ldots\ e_n : \mu}\ \textsc{App}$$

$$\frac{n \geqslant 0 \qquad \Gamma \vdash^{\mathsf{fun}} e_0 : \phi \qquad \phi \ \leqslant^{\mathfrak{u}}_{\omega_1, \ldots, \omega_n} \ \sigma_1, \ldots, \sigma_n; \eta \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_1} e_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash^{\mathsf{arg}}_{\omega_n} e_n : \sigma_n}{\Gamma \vdash (e_0\, e_1\ \ldots\ e_n :: \forall \overline{b}.\, \eta) : \forall \overline{b}.\, \eta}\ \textsc{AnnApp}$$

Figure 4. Declarative type system

simply bring x into scope with type ϕ. As discussed in Section 2.3, where there is no annotation (rule Abs) we insist that x has a fully-monomorphic type τ.

All the action is in rule App for n-ary applications ($e_0\ e_1 \ldots e_n$). First, note that a lone variable is treated as a nullary application. Second, the typing rules allow us to break an n-ary application in different ways, because e_0 could itself be an application. In practice we always choose e_0 not to be an application, so we get as many arguments as possible, and thereby maximise the opportunities for guardedness.

The first step in App is typing the head of the application e_0. The corresponding judgment $\vdash^{\mathsf{fun}}$ either looks for a variable in the environment or uses the normal typing judgment if the head is another kind of term. After typing the head, rule app instantiates the type of the head with the *instantiation* judgement $\leqslant^{\mathfrak{s}}_{\overline{\omega}}$, yielding a list of argument types $\sigma_1 \ldots \sigma_n$. App uses $\vdash^{\mathsf{arg}}$ to check each argument e_i against the corresponding σ_i. This argument-checking judgement $\vdash^{\mathsf{arg}}$ can *generalise* the inferred type of the argument to match the type σ expected by the function, to support higher-rank polymorphism. Without such a rule, we would not be able to type check *poly* $(\lambda x.\, x)$, which requires the argument to be of type $\forall a.\, a \to a$.

For example, an application of a function to three arguments would give rise to the following instantiation:

$$\begin{array}{l} \forall ab.\, a \to (\forall c.\, [a] \to b \to c \to [b]) \\ \quad \leqslant^{\mathfrak{m}}_{\omega_1,\omega_2,\omega_3} \;\; \sigma_a, [\sigma_a], \tau_b; (\tau_c \to [\tau_b]) \end{array}$$

This example illustrates several points. First, the bit-vector $\overline{\omega}$ corresponds 1-1 with the arguments in the application. For now, the bit-values are irrelevant, but we will use them in Section 3.3. Second, the instantiation judgement returns a list of argument types that matches the length of the vector $\overline{\omega}$ (rule InstArrow); in this case, there are three arguments, and the instantiated argument types are σ_a, $[\sigma_a]$, and τ_b. Third, the judgement can instantiate nested foralls.

Fourth, and core to our contribution, each type variable is instantiated, by rule InstPoly, with a type whose sort reflects the way the type variable appears in the function type. This analysis is performed by the judgement $\sigma \rhd^{\mathfrak{s}}_{\overline{\omega}} \Delta$, which returns a classification Δ of the free type variables of σ. This classification directly implements the three cases of the Instantiation Rule (Section 2.1):

1. If a appears under a type constructor (i.e. guarded) in any of the first n arguments of σ, then a may be instantiated by an unrestricted type ϕ (rule ArgGuard).
2. If a appears in one of the first n arguments of σ, a may be instantiated by a top-level monomorphic type μ (rule ArgTyVar).
3. If a appears only in the result type of σ, after stripping off n arguments, then a may be instantiated by a type of sort $\mathfrak{s}$ (rule ArgsRes). In the invocation of $\leqslant$ in rule App the sort $\mathfrak{s}$ is always fully monomorphic $\mathfrak{m}$, but we need the extra generality for annotations (Section 3.4).

Once we have Δ to classify each type variable rule, InstPoly instantiates the function type with a Δ-respecting substitution θ, and recurses. Figure 4 also defines what it means for a substitution θ to "respect" a classification Δ.

Unlike some other systems (see [13] for a comprehensive account), all type constructors in GI are invariant, *including functions*. This means that neither $[\forall a.\, a \to a] \not\leqslant^{\mathfrak{s}}_{\overline{\omega}} [Int \to Int]$ nor $Int \to (\forall a.\, a \to a) \not\leqslant^{\mathfrak{s}}_{\overline{\omega}} Int \to Int \to Int$. However, because $\leqslant$ handles foralls nested to the right of arrows (rule InstPoly), we can often work around the lack of covariance using η-expansion. For example, suppose we have functions $f :: \forall a.\, a \to (\forall b.\, b \to a)$, and $g :: (Int \to Int \to Int) \to Bool$. Then $(g\ f)$ is ill-typed; but $(g\ (\lambda xy.\, f\ x\ y))$ is well typed.

3.3 Single variables

The type system described up to now propagates polymorphism between arguments and from the arguments to the return type of the function – this is the essence of guardedness. Alas, this creates a problem for single variables, which are treated as nullary applications: since there are no arguments, no type variable is considered guarded, and thus

$$\frac{x : \boxed{\forall \overline{p}.\, \tau} \in \Gamma \qquad \overline{b} \notin \Gamma}{\Gamma \vdash^{\mathsf{arg}}_{\star} x : \forall \overline{b}.\, [\overline{a \mapsto \sigma}]\tau}\ \textsc{VarGen}$$

$$\frac{\Delta_1 = [\mathsf{ftv}(\sigma_1) \mapsto \mathfrak{m}] \qquad \sigma_2 \rhd^{\mathfrak{s}}_{\overline{\omega}} \Delta_2}{\sigma_1 \to \sigma_2 \rhd^{\mathfrak{s}}_{\star,\overline{\omega}} \Delta_1 \sqcup \Delta_2}\ \textsc{ArgsStar}$$

Figure 5. Decl. type system with single variables

impredicativity is forbidden. We can see this by expanding the derivation of $\Gamma \vdash^{\mathsf{arg}}_{\bullet} x : \sigma$ for the case of a variable x.

$$\dfrac{\overline{b} \notin \Gamma \qquad \dfrac{\dfrac{x : \phi \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \phi}\ \textsc{VarHead} \qquad \boxed{\phi \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \mu}}{\Gamma \vdash x : \mu}\ \textsc{App}}{\Gamma \vdash^{\mathsf{arg}}_{\bullet} x : \forall \overline{b}.\, \mu}\ \textsc{ArgGen}$$

The highlighted premise, $\phi \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \mu$, forces instantiation to use only fully monomorphic types.

That is embarrassing, because we cannot typecheck, say *choose* [ ] *ids* from Figure 2. From the type of *ids* it is obvious that [ ] should be given a type $[\forall a.\, a \to a]$. But we cannot do so, because such instantiation is not fully monomorphic. We get back into swampy waters.

For the case of single variables with a rank-1 polymorphic type (quantifiers appear only at top-level) we can get out of the swamp. The VarGen rule in Figure 5 formalizes this idea: if a single variable x appears as the argument of an application – hence the use of the judgement $\vdash^{\mathsf{arg}}$ – and it has a rank-1 type $\forall \overline{p}.\, \tau$, we may instantiate impredicatively before generalizing. This is enough to cover the case of *choose* [ ] *ids*, because the type of [ ] :: $\forall a.\, [a]$ has the right shape. Note, however, that it is still the case that the justification for instantiating the type of the first argument [ ] with top-level monomorphic type has to come from the second argument *ids*. If such justification is not forthcoming, as it should be in the case of *choose* [ ] [ ], then both arguments should be instantiated fully monomorphically.

We manage the bookkeeping to distinguish these situations by keeping track of a vector of bits $\overline{\omega}$. Its elements correspond to the arguments in an application: a $\star$ means that the rule VarGen was applied to type the corresponding argument; application of ArgGen is represented by $\bullet$. The rule ArgsStar ensures that whenever the rule VarGen was used to type a given argument, we reset the sorts of the free variables to $\mathfrak{m}$ so that however these type variables were instantiated, this information cannot be used to justify the impredicative instantiation of other arguments.

3.4 Annotations

The VarGen rule makes it possible to accept more programs, but it is restricted to a very special class of expressions. For the general case, we provide *annotations* as part of the syntax. Since annotations fully specify the types, we do not need to impose guardedness restrictions on those variables appearing in the result. Take the expression *single* $(\lambda x.\, x)$. Due to the type of *single* being $\forall p.\, p \rightarrow [p]$, the type of the expression must be $[\tau \rightarrow \tau]$ for a monomorphic τ. If we want instead to obtain $[\forall a.\, a \rightarrow a]$, we can just annotate the result, thus *single* $(\lambda x.\, x) :: [\forall a.\, a \rightarrow a]$.

Rule AnnApp is almost identical to App, except for the choice of parameter to the instantiation judgment, which is $\mathfrak{u}$. This implies that in contrast to non-annotated applications, variables in the result type of the function might be substituted by any type, polymorphic or not. This is sensible, the annotation tells us exactly what the types are that those variables should be instantiated with.

Annotations also free us from having a different judgment for declarations and expressions. For every combination $f :: \sigma; f = e$ in the source code, we just need to pose the problem of checking $f = e :: \sigma$ for well-typedness.

3.5 let bindings

The simplest way to type **let** $x = e_1$ **in** e_2 is to see it as a shorthand for $(\lambda x.\, e_2)\ e_1$. Alas, such a translation imposes an important restriction on the type of x: it must be fully monomorphic even though the type of e_1 might be more general. The reason is that we try to guess the type of e_1 by looking at the way it is used in e_2, instead of looking at e_1 itself. But there is no need to be so restrictive! The rule Let in Figure 4 works in the other direction: the type obtained from typing e_1 is put in the environment as the one for x, allowing the type of x to be fully polymorphic instead.

One difference between **let** bindings in GI and Hindley-Milner is that the latter always generalizes the type of a let-bound identifier before passing it to the body of the let. Vytiniotis et al. [21] argue however that **let**-generalisation is not so important in practice and that in complex type systems how to generalize is not completely clear. If desired, generalisation can be obtained by annotating the bound expression, **let** $x = (e_1 :: \phi)$ **in** e_2.

3.6 Metatheory

Impredicativity is a great tool, but we do not want to lose those programs which only require top-level polymorphism. The following theorem states this fact, except for the different take on **let** generalization we discussed in Section 3.5.

Theorem 3.1 (Compatibility with rank-1 polymorphism)**.** *Let e be an expression in the syntax of the lambda-calculus with predicative rank-1 polymorphism. If $\Gamma \vdash e : \tau$ in that type system, then $\Gamma \vdash e : \tau$ in* GI.

One key property of GI is that all impredicative instantiations are settled by the shape of the expression and the types in the environment, modulo some monomorphic substitution. For that reason, we say that impredicative polymorphism is *not guessed* in GI.

Theorem 3.2 (Impredicative instantiation is not guessed)**.** *Let Γ be an environment and e an expression. For every pair of fully monomorphic substitutions θ_1 and θ_2, if $\theta_1\Gamma \vdash e : \sigma_1$ and $\theta_2\Gamma \vdash e : \sigma_2$, then there exists a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*

Corollary 3.3. *Let Γ be a closed environment (that is, no type in Γ contains a free variable). If $\Gamma \vdash e : \sigma_1$ and $\Gamma \vdash e : \sigma_2$, then there is a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*

This property suggests a notion of *principal type* similar to the one found in Hindley-Milner. A principal type for an expression e is defined as a type σ^* for which any other type assignment ϕ to e is equal to $\theta\sigma^*$ for a fully monomorphic substitution θ. The fact that we only need to consider *fully monomorphic* substitutions here is a direct consequence of Theorem 3.2. The proof of the principal types property, however, is a corollary of other properties of the inference process, which we describe in Section 4.4. Note also that this theorem only promises that if GI accepts an expression, there was no guessing involved. But there are expressions for which only one choice of polymorphism is possible, yet GI cannot find it and an annotation is required.

In the remainder of this section we look at some properties of GI concerning derivations and stability under transformations. We use the notation $e_1[e_2]$ refers to a context in which e_2 appears. Proofs are given in Appendix D.1.

Theorem 3.4 (Substitution)**.** *If $\Gamma \vdash u : \sigma$ and $\Gamma, x : \sigma \vdash e[x] : \phi$, then $\Gamma \vdash e[u] : \phi$.*

The converse result does not holds: we cannot in general abstract over part of an expression. In practice, that means that we cannot always introduce a **let**, as in changing $e_0\ e_1\ e_2$ to **let** $x = e_0\ e_1$ **in** $x\ e_2$. Our App rule is responsible for propagating information between arguments, if we introduce a **let** for part of an expression, this bound is lost.

Theorem 3.5. *Let $app :: \forall a\ b.\, (a \rightarrow b) \rightarrow a \rightarrow b$ and $revapp :: \forall\ a\ \ b.\, a \rightarrow (a \rightarrow b) \rightarrow b$ be the application and reverse application functions, respectively. Given two expressions f and e such that $\Gamma \vdash^{\mathsf{fun}} f : \sigma_0$, and $\sigma_0 \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \sigma_1 \rightarrow \phi$ then:*

$$\Gamma \vdash f\ e : \phi \iff \Gamma \vdash app\ f\ e : \phi \iff \Gamma \vdash revapp\ e\ f : \phi$$

The hypothesis $\sigma_0 \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \sigma_1 \rightarrow \phi$ means that type variables in f may only be instantiated with fully monomorphic variables. Thus, this transformation only respects well-typedness for the predicative fragment, but not in general when impredicative instantiation is allowed. One example of this restriction is the application $f\ \ ids$, where f has type $\forall a.\, [a] \rightarrow [a]$, which cannot be turned into $app\ f\ \ ids$. The reason is that in the original application the type variable a in f has to be instantiated with a polymorphic type $\forall b.\, b \rightarrow b$. However, this is not allowed in the form with app.

One property which does not hold in GI is *full subject reduction.* If $\Gamma \vdash e : \sigma$ and e β-reduces to e', it may not hold that $\Gamma \vdash e' : \sigma$. For example, *app auto* is typeable, but not its reduced form $\lambda x.\, auto\ x$, since it requires an annotation on x. Subject reduction holds in a milder form: if $\Gamma \vdash e : \sigma$, e β-reduces to e' *and* $\Gamma \vdash e' : \phi$, then σ and ϕ coincide.

4 Type inference using constraints

In the previous section we described GI from a declarative perspective and now we turn to describing an efficient type inference algorithm for it.

Following Pottier and Rémy [15], we first walk over the syntax tree of the source program and generate *typing constraints*, a process that typically introduces many *unification variables* that stand for as-yet-unknown types. Next, we solve those constraints producing a *type substitution* for these unification variables. By separating type inference in two simpler problems, the implementation and conceptual overhead with new source language and type system features remains low. For example, earlier work dubbed OutsideIn(X) applies these ideas to a language like Haskell, with type classes, type-level functions, GADTs, and the like [21]. Another advantage is more sophisticated type-error diagnosis [6, 20, 25].

4.1 Constraints

The main challenge of type inference for impredicativity concern instantiation and generalisation of terms with polymorphic type. Consider the call (*head ids True*), when fully elaborated we want to generate this System F term:

head $(\forall a.\, a \to a)$ *ids Bool True*

That is, we instantiate *head* at type $(\forall a.\, a \to a)$, then apply it to *ids*, to produce a result of type $(\forall a.\, a \to a)$. Now we must in turn instantiate that type with *Bool* to get a function of type $(Bool \to Bool)$ which we can apply to *True*. This second instantiation is problematic because, at constraint generation time, we do not yet know what type we are going to instantiate *head* at; all we know is that (*head ids*) has type α for some as-yet-unknown type α. So we want to defer the instantiation decision.

Sometimes we must defer generalisation decisions too. Consider the function application $((:)\ (\lambda x.\, x)\ ids)$. In System F terms, we want to infer the following elaborated program:

$(:)\ (\forall a.\, a \to a)\ (\Lambda a.\, \lambda(x : a).\, x)\ ids$

in which $(:)$ is instantiated at type $(\forall a.\, a \to a)$, and $(:)$'s first argument is generalised to have that polymorphic type. Now consider constraint generation for this expression. We may instantiate the type of $(:)$ with a fresh unification variable, α say. Ultimately the type of *ids* forces α to be $\forall a.\, a \to a$, but we don't know that yet. Moreover, in the final program we will need to generalise the type of $(\lambda x.\, x)$, but again at constraint generation time we don't know that type either.

When we don't know something at constraint generation time, the solution is to *defer the choice, by generating a constraint that represents that choice.* This is the key idea of the

Unif. var. names	$\mathbb{U}$	$\ni$	$\alpha, \beta, \gamma, \delta, \ldots$
Unif. var.	v	$::=$	$\alpha^{\mathfrak{s}}$
Free unification variables	$\mathrm{fuv}(\sigma)$		
Fully mono. types	τ	$::=$	$\alpha^{\mathfrak{m}} \mid a \mid \mathsf{T}\,\overline{\tau}$
Top-level mono. types	μ, η	$::=$	$\alpha^{\mathfrak{t}} \mid a \mid \mathsf{T}\,\overline{\sigma}$
Polymorphic types	σ, ϕ	$::=$	$\alpha^{\mathfrak{u}} \mid \forall \overline{a}.\, \mu$
	$\overline{a}$ may be empty, $\overline{a} \subseteq \mathrm{ftv}(\mu)$		
Types with generalisation	$\mathfrak{g}$	$::=$	$\forall\{\overline{v}\}.\, C \Rightarrow \sigma$
	$\overline{v}$ may be empty		
Constraints	C	$::=$	$\top$
Conjunction		$\mid$	$C_1 \wedge C_2$
Equality		$\mid$	$\sigma \sim \phi$
Instantiation		$\mid$	$\sigma \leqslant^{\mathfrak{s}}_{\omega} \overline{\phi}; \mu$
Generalisation		$\mid$	$\mathfrak{g} \preceq \sigma$
Quantification		$\mid$	$\forall \overline{a}.\, \exists \overline{v}.\, C$

Figure 6. Extended syntax

constraint solving approach. The game is to develop a constraint language that neatly embodies the choices that we want to defer, and a solver that can subsequently make those choices. With that in mind, Figure 6 gives the syntax of our constraint language.

As mentioned earlier, constraint generation produces many *unification variables*, each of which stands for an as-yet unknown type. Looking at Figure 6, a key idea is that unification variables are drawn from three distinct "alphabets": $\alpha^{\mathfrak{s}}$ for each of the threes sorts $\mathfrak{s}$. (Sorts were introduced in Figure 3.) The sort of a unification variable specifies the possible types that the unification variable can stand for; operationally, a unification variable of sort $\mathfrak{s}$ may only be unified with types belonging to that sort.

The syntax presents several kinds of constraints C. These constraints do not form part of the source language; they are internal to the solver. Equality constraints are self explanatory. Instantiation constraints arise from the occurrence of a polymorphic variable, whose type must be instantiated – but that decision must be deferred (embodied in a constraint). Quantification constraints arise from explicit user type signatures, and pattern matching on data types involving existentials and GADTs. Both are fairly conventional. However *generalisation constraints* are new; they precisely embody the deferred decision about generalisation that we mention above.

4.2 Constraint generation

Constraint *generation* is described in Figure 7 as a four-element judgment $\Gamma \vdash e : \sigma \rightsquigarrow C$. The first two elements are inputs: the environment Γ and the expression e for which to generate constraints. The output of the process is a type σ

$$\boxed{\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow C}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \sigma \rightsquigarrow \epsilon}\ \textsc{VarHead} \qquad \frac{e \text{ not var. or app.} \qquad \Gamma \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow C}\ \textsc{ExprHead}$$

$$\boxed{\Gamma\,;\,\overline{v} \vdash^{\mathsf{arg}}_{\omega} e : \sigma \rightsquigarrow C}$$

$$\frac{x : \forall\overline{p}.\,\tau \in \Gamma \qquad \forall\overline{p}.\,\tau \text{ closed} \qquad \overline{\alpha} \text{ fresh}}{\Gamma\,;\,\overline{v} \vdash^{\mathsf{arg}}_{\star} x : \sigma \rightsquigarrow [\overline{p \mapsto \alpha^{\mathfrak{u}}}]\tau \preceq \sigma}\ \textsc{VarGen} \qquad \frac{\Gamma \vdash e : \phi \rightsquigarrow C \qquad \overline{v}' = \mathsf{fuv}(\phi, C) - \overline{v}}{\Gamma\,;\,\overline{v} \vdash^{\mathsf{arg}}_{\bullet} e : \sigma \rightsquigarrow \mathbb{V}\{\overline{v}'\}.\,C \Rightarrow \phi \preceq \sigma}\ \textsc{ArgGen}$$

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow C}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x : \alpha^{\mathfrak{m}} \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash \lambda x.\,e : \alpha^{\mathfrak{m}} \to \sigma \rightsquigarrow C}\ \textsc{Abs} \qquad \frac{\Gamma, x : \phi \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash \lambda(x :: \phi).\,e : \phi \to \sigma \rightsquigarrow C}\ \textsc{AnnAbs}$$

$$\frac{\overline{\alpha}, \beta \text{ fresh} \qquad \Gamma \vdash^{\mathsf{fun}} e_0 : \phi \rightsquigarrow C \qquad \Gamma\,;\,\mathsf{fuv}(\Gamma, \phi, C) \vdash^{\mathsf{arg}}_{\omega_i} e_i : \alpha_i^{\mathfrak{u}} \rightsquigarrow C_i}{\Gamma \vdash e_0\, e_1\, \ldots\, e_n : \beta^{\mathfrak{t}} \rightsquigarrow C \wedge \phi \leqslant^{\mathfrak{m}}_{\omega_1,\ldots,\omega_n} \alpha_1^{\mathfrak{u}}, \ldots, \alpha_n^{\mathfrak{u}}; \beta^{\mathfrak{t}} \wedge C_1 \wedge \cdots \wedge C_n}\ \textsc{App}$$

$$\frac{\overline{\alpha} \text{ fresh} \qquad \Gamma \vdash^{\mathsf{fun}} e_0 : \phi \rightsquigarrow C \qquad \overline{v}' = \mathsf{fuv}(\phi, C) - \mathsf{fuv}(\Gamma) \qquad \Gamma\,;\,\mathsf{fuv}(\Gamma, \phi, C) \vdash^{\mathsf{arg}}_{\omega_i} e_i : \alpha_i^{\mathfrak{u}} \rightsquigarrow C_i}{\Gamma \vdash (e_0\, e_1\, \ldots\, e_n :: \forall\overline{b}.\,\eta) : \forall\overline{b}.\,\eta \rightsquigarrow \forall\overline{b}.\,\exists\overline{\alpha^{\mathfrak{u}}}\,\overline{v}'.\,\big(C \wedge \phi \leqslant^{\mathfrak{u}}_{\omega_1,\ldots,\omega_n} \alpha_1^{\mathfrak{u}}, \ldots, \alpha_n^{\mathfrak{u}}; \eta \wedge C_1 \wedge \cdots \wedge C_n\big)}\ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_1 : \phi \rightsquigarrow C_1 \qquad \Gamma, x : \phi \vdash e_2 : \sigma \rightsquigarrow C_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma \rightsquigarrow C_1 \wedge C_2}\ \textsc{Let}$$

Figure 7. Constraint generation

assigned to the expression, possibly including some unification variables, and the set of extended constraints C that the types must satisfy.

Rules Abs and AbsAnn are not surprising: they just extend the environment with a new unification variable or a given polymorphic type, respectively, and then proceed to generate constraints for the body of the abstraction. The usage of a fully monomorphic variable in Abs mimics the restriction imposed by the declarative specification.

Rule App is where most of the work is done. Just like the declarative specification (Figure 4), the head of the application is typed using an ancillary judgment $\Gamma \vdash^{\mathsf{fun}} e : \phi \rightsquigarrow C$, which either looks up a variable in the environment or threads the information to the normal gathering process.

Another ancillary judgment $\Gamma\,;\,\overline{v} \vdash^{\mathsf{arg}}_{b} e : \sigma \rightsquigarrow C$ is used to generate constraints on the arguments. In this judgement we use introduce a completely new constraint form, $\mathfrak{g} \preceq \sigma$, which we call *generalisation* constraint. These constraints allow us to defer the generalisation decisions to the solver, as sketched in Section 4.1. The constraint $(\mathbb{V}\{\overline{v}\}.\,C \Rightarrow \phi) \preceq \sigma$ should be read *"a term of type ϕ with constraints C and unification variables $\overline{v}$ can be instantiated and/or generalised to have type σ"*. Even looking at the syntax alone, you can see that the fruits of constraint generation for each argument e_i are wrapped up, along with the expected argument type α_i from the function, into a generalisation constraint for the solver to deal with later. The reason for the additional argument $\overline{v}$ to the judgement is that we need to forbid generalisation over variables which are visible in the environment or the whole application.

Rule AnnApp deals with a type-annotated application $(e_0\, e_1\, \ldots\, e_n :: \sigma)$. It is similar to App, but it is the first rule to introduce a *quantification constraint* $\forall\overline{b}.\,\exists\overline{v}.\,\overline{C}$. This binds the Skolem variables $\overline{b}$ from the type signature, and existentially quantifies the unification variables free in the constraint but not used outside it. The other difference with App is the use $\mathfrak{u}$ as a parameter to $\leqslant$, rather than $\mathfrak{m}$, as done in the declarative specification.

4.3 Constraint solving

The solver takes the generated constraint C and its free unification variables $\overline{v} = \mathsf{fuv}(C)$, and repeatedly applies the solver rules in Figure 8, until no rule applies. The result is a *residual constraint*. If the residual constraint is in *solved form*, then the program is well typed; if not, the unsolved constraints (e.g. $\mathsf{Int} \sim \mathsf{Bool}$) represent type errors that can be reported to the user. We concentrate first on the solver rules that incrementally solve the constraint.

Each of the rules in Figure 8 rewrites a configuration $C; \overline{v}$ to another configuration. The unification variables $\overline{v}$ are existentially quantified, so you can think of a configuration as representing $\exists\overline{v}.C$. Rule conj and forall are structural rules: the former allows a rule to be applied to one part of a conjunction, while the latter allows a rule to be applied under a quantification. To avoid clutter we implicitly assume that

$$\boxed{\mathsf{freshen}^{\mathfrak{s}}_{\overline{\omega}}(\sigma) \implies \langle \overline{v}, \mu \rangle}$$

$$\frac{\mu \rhd^{\mathfrak{s}}_{\overline{\omega}} \Delta \qquad \overline{\alpha} \text{ fresh} \qquad v_i = \alpha_i^{\Delta(a_i)}}{\mathsf{freshen}^{\mathfrak{s}}_{\overline{\omega}}(\forall \overline{a}.\, \mu) \implies \langle \overline{v}, [\overline{a \mapsto v}]\mu \rangle}$$

$$\boxed{C \mathbin{;} \overline{v} \implies C' \mathbin{;} \overline{v}'}$$

$$[\textsc{conj}]\ \frac{C_1 \mathbin{;} \overline{v} \implies C_1' \mathbin{;} \overline{v}'}{C_1 \wedge C_2 \mathbin{;} \overline{v} \implies C_1' \wedge C_2 \mathbin{;} \overline{v}'} \quad \begin{array}{l}\text{modulo assoc.}\\ \text{and comm. of } \wedge\end{array} \qquad [\textsc{forall}]\ \frac{C \mathbin{;} \overline{v}_{\mathsf{in}} \implies C' \mathbin{;} \overline{v}'_{\mathsf{in}}}{\forall \overline{a}.\, \exists \overline{v}_{\mathsf{in}}.\, C \mathbin{;} \overline{v} \implies \forall \overline{a}.\, \exists \overline{v}'_{\mathsf{in}}.\, C' \mathbin{;} \overline{v}}$$

$$\begin{array}{llcll}
[\top\textsc{ident}] & \top \wedge C \mathbin{;} \overline{v} & \implies & C \mathbin{;} \overline{v} & \\
[\top\textsc{forall}] & \forall \overline{a}.\, \exists \overline{v}_{\mathsf{in}}.\, \top \mathbin{;} \overline{v} & \implies & \top \mathbin{;} \overline{v} & \\
[\textsc{eqrefl}] & \sigma \sim \sigma \mathbin{;} \overline{v} & \implies & \top \mathbin{;} \overline{v} & \\
[\textsc{eqmono}] & \mathsf{T}\, \sigma_1 \,\ldots\, \sigma_n \sim \mathsf{T}\, \phi_1 \,\ldots\, \phi_n \mathbin{;} \overline{v} & \implies & (\sigma_1 \sim \phi_1) \wedge \cdots \wedge (\sigma_n \sim \phi_n) \mathbin{;} \overline{v} & \\
[\textsc{eqsubst}] & (\alpha^{\mathfrak{s}} \sim \sigma) \wedge C \mathbin{;} \overline{v} & \implies & (\alpha^{\mathfrak{s}} \sim \sigma) \wedge [\alpha^{\mathfrak{s}} \mapsto \sigma]C \mathbin{;} \overline{v} & \text{if } \sigma \text{ respects } \mathfrak{s}, \text{ and } \alpha \notin \mathsf{ftv}(\sigma) \\
[\textsc{eqvar}] & \alpha^{\mathfrak{s}_1} \sim \beta^{\mathfrak{s}_2} \mathbin{;} \overline{v} & \implies & \beta^{\mathfrak{s}_2} \sim \alpha^{\mathfrak{s}_1} \mathbin{;} \overline{v} & \text{if } \mathfrak{s}_1 \sqsubset \mathfrak{s}_2 \\
[\textsc{eqfully}] & \alpha^{\mathfrak{m}} \sim \sigma \mathbin{;} \overline{v} & \implies & \{\beta^{\mathfrak{s}} \sim \gamma^{\mathfrak{m}} \mid \beta^{\mathfrak{s}} \in \mathsf{fuv}(\sigma), \mathfrak{s} \neq \mathfrak{m}\} \mathbin{;} \overline{v}, \overline{\gamma^{\mathfrak{m}}} & \overline{v}, \overline{\gamma^{\mathfrak{m}}} \text{ disjoint} \\
[\textsc{inst}\epsilon] & \mu \leqslant^{\mathfrak{s}}_{\epsilon} \epsilon; \eta \mathbin{;} \overline{v} & \implies & \mu \sim \eta \mathbin{;} \overline{v} & \\
[\textsc{inst}\rightarrow] & \mu \leqslant^{\mathfrak{s}}_{\omega, \overline{\omega}} \sigma, \overline{\phi}; \eta \mathbin{;} \overline{v} & \implies & (\mu \sim \sigma \rightarrow \beta^{\mathfrak{u}}) \wedge (\beta^{\mathfrak{u}} \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta) \mathbin{;} \overline{v} & \\
[\textsc{inst}\forall\textsc{l}] & (\forall \overline{a}.\, \mu) \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta \mathbin{;} \overline{v} & \implies & \mu' \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta \mathbin{;} \overline{v}, \overline{v}' & \text{where } \mathsf{freshen}^{\mathfrak{s}}_{\overline{\omega}}(\forall \overline{a}.\, \mu) \implies \langle \overline{v}', \mu' \rangle \\
[\textsc{inst}\mathbb{V}\textsc{l}] & (\mathbb{V}\{\overline{v}'\}.\, \overline{C} \Rightarrow \sigma) \leq \eta \mathbin{;} \overline{v} & \implies & \overline{C} \wedge \left(\sigma \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \eta\right) \mathbin{;} \overline{v}, \overline{v}' & \overline{v}, \overline{v}' \text{ disjoint} \\
[\textsc{inst}\forall\textsc{r}] & \mathfrak{g} \leq (\forall \overline{a}.\, \mu) \mathbin{;} \overline{v} & \implies & \forall \overline{a}.\, (\mathfrak{g} \leq \mu) \mathbin{;} \overline{v} &
\end{array}$$

Figure 8. Solving rules

the rules are read modulo commutativity and associativity of $\wedge$; that is why CONJ only has to handle the left conjunct. The rules $\top$IDENT and $\top$FORALL remove $\top$ constraints from the set, since they are identities for the conjunction.

4.3.1 Basic rules

Rule EQREFL removes trivial equality constraints $\sigma \sim \sigma$. Rule EQMONO indicates that two types headed by constructors are equal if and only if their heads coincide and all the arguments are equal. EQSUBST is the only rule that involves the interaction of two constraints. It applies the substitution of a unification variable to any other constraints conjoined with it (remember the implicit associativity and commutativity of $\wedge$), provided sorts are respected and the substitution does not lead to an infinite type (hence the occurs check). Notice that the equality constraint is not discarded; it remains in case it is needed again; indeed, these equality constraints remain in a solved constraint.

Given the different behaviour embodied by the different sorts of variables, the solver has to propagate this information. EQVAR ensures that whenever we have two variables with different sorts, the least restrictive one is substituted by the most restrictive. For example, when we have an unrestricted $\alpha^{\mathfrak{u}}$ and a top-level monomorphic $\beta^{\mathfrak{t}}$, then $\alpha^{\mathfrak{u}}$ should be replaced by $\beta^{\mathfrak{t}}$, and not the other way around. Full monomorphism goes deeper: EQFULLY ensures that if a type σ is equated with a fully monomorphic variable $\alpha^{\mathfrak{m}}$, all the variables in σ become fully monomorphic too.

One difference between these rules and other presentations is that we do not rewrite an unsatisfiable constraint, such as *Int* $\sim$ *Bool*, to $\perp$. Instead, that constraint is simply stuck, and we can report it at the end.

Note that two polymorphic types need to be *syntactically equal* (modulo α-equality) to match under the EQREFL rule. This means that $(\forall a\ b.\, a \rightarrow b \rightarrow b) \sim (\forall b\ a.\, a \rightarrow b \rightarrow b)$ does *not* hold in our system. As we discuss in Section 2.4, this is not problematic, since on application type variables are instantiated and regeneralized using the $\leq$ relation.

4.3.2 Instantiation and generalisation constraints

For instantiation constraints $\leqslant^{\mathfrak{s}}_{\overline{\omega}}$, we follow closely the judgement with the same name in the declarative specification. Rule INSTϵ encodes the fact that once all arguments are processed, and thus $\overline{\omega}$ is an empty vector of bits, the remaining types must be equal. This is a consequence of the invariance of type constructors. On the other hand, if we have a top-level monomorphic type μ and the vector of bits is not empty, the only possibility of for μ to be a function type. This is the goal of the INST→ rule.

INST∀L instantiates a polytype σ with fresh unification variables, much as in the usual Damas-Milner algorithm, except that we must use a sort-respecting instantiation. This is done by $\text{freshen}^{\mathfrak{s}}_{\overline{\omega}}$, which in turn uses the already-introduced classification judgment $\rhd^{\mathfrak{s}}_{\overline{\omega}}$ (Figure 4). Finally, the new variables enter the set of existentially quantified variables.

Finally, we come to generalisation constraints, which (recall Section 4.1) express a deferred generalisation decision. Rule INST$\mathbb{V}$L is simple: if the right hand side has no top-level foralls (it is of the form η) then there is no generalisation to be done, so it suffices to release all the captured constraints C and existentials υ' into the current constraint.

INST$\mathbb{V}$R is where actual generalisation takes place. In order to forge some intuition, let us look at the constraint

$$(\mathbb{V}\{\alpha\,\beta\}.\,(\alpha\ \leqslant^{\mathfrak{m}}_{\epsilon}\ \epsilon;\beta) \Rightarrow \alpha \to \beta) \leq (\forall p.\, p \to p)$$

This generalisation constraint says that by performing some solving and possibly abstracting over some of the variables α and β, we should get the polymorphic type $\forall p.\, p \to p$. Following standard practice, we skolemise the type on the right, introducing a fresh skolem or rigid variable p, which should not be unified.

$$(\alpha\ \leqslant^{\mathfrak{m}}_{\epsilon}\ \epsilon;\beta)\ \wedge\ (\alpha \to \beta \sim p \to p)$$

We obtain a solution by making $\alpha \sim \beta \sim p$. In order for this solution to remain valid, we must guarantee that the skolem p *does not escape* to the outer world. We recall this restriction by means of a fresh quantification constraint

$$\forall p.\, \exists \alpha\, \beta.\, (\alpha\ \leqslant^{\mathfrak{m}}_{\epsilon}\ \epsilon;\beta)\ \wedge\ (\alpha \to \beta \sim a \to a)$$

Rule INST$\mathbb{V}$R achieves this rather neatly simply by doing skolemisation and pushing the $\mathfrak{g}$ inside; then FORALL and INST$\mathbb{V}$L will do the rest.

The rules applicable to instantiation and generalisation constraints do not handle every case. In particular, whenever an unrestricted variable appears in one of the sides of the constraint, there are good reasons to wait:

1. If we have $\alpha^{\mathfrak{u}}\ \leqslant^{\mathfrak{s}}_{\epsilon}\ \epsilon;\mu$ we cannot turn it directly into $\alpha^{\mathfrak{u}} \sim \mu$, because $\alpha^{\mathfrak{u}}$ might be unified later to a polymorphic type and we need instantiation.
2. Similarly, if we have $(\mathbb{V}\{\overline{\alpha}\}.\,C \Rightarrow \mu) \leq \alpha^{\mathfrak{u}}$, and $\alpha^{\mathfrak{u}}$ is later substituted by a polytype, we must skolemise.

The guardedness restrictions are carefully crafted to ensure that the solver is *confluent* and that it is never completely *stuck*, unless the constraint set as a whole is inconsistent. A single constraint can be stuck for some time, but if the whole set is consistent, by steps applied to other constraints, it will eventually become unstuck.

Theorem 4.1. *Suppose $\Gamma \vdash e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a new set C' without instantiation and generalisation constraints which fixes the value of all unrestricted and top-level monomorphic variables.*

$$\frac{}{\overline{a}\ ;\ \overline{\alpha}\ ;\ \emptyset \vdash \top\ \text{solved}}\ \textsc{Solved}\top$$

$$\frac{\sigma\ \text{respects}\ \mathfrak{s} \qquad \text{ftv}(\sigma) \subseteq \overline{a} \cup \overline{\alpha}}{\overline{a}\ ;\ \overline{\alpha}\ ;\ \{\beta\} \vdash \beta^{\mathfrak{s}} \sim \sigma\ \text{solved}}\ \textsc{SolvedVar}$$

$$\frac{\overline{a}\ ;\ \overline{\alpha}\ ;\ \overline{\beta}_1 \vdash C_1\ \text{solved} \qquad \overline{a}\ ;\ \overline{\alpha}\ ;\ \overline{\beta}_2 \vdash C_2\ \text{solved}}{\overline{a}\ ;\ \overline{\alpha}\ ;\ \overline{\beta}_1 \uplus \overline{\beta}_2 \vdash C_1 \wedge C_2\ \text{solved}}\ \textsc{SolvedConj}$$

$$\frac{\overline{\upsilon} = \overline{\gamma}_1 \uplus \overline{\gamma}_2 \qquad \overline{a} \cup \overline{b}\ ;\ \overline{\alpha} \cup \overline{\gamma}_1\ ;\ \overline{\gamma}_2 \vdash C\ \text{solved}}{\overline{a}\ ;\ \overline{\alpha}\ ;\ \emptyset \vdash \forall \overline{b}.\, \exists \overline{\upsilon}.C\ \text{solved}}\ \textsc{SolvedQuant}$$

Figure 9. Definition of solved set of constraints

Theorem 4.1 tells us that the process of solving can be divided into two phases. In the first phase all constraints, including instantiation and generalization constraints, are turned into a set of equalities, possibly with different quantification levels. This is an instance of the problem of first-order unification under a mixed prefix [3], for which a complete solving algorithm is described by Pottier and Rémy [15].

4.4 Soundness, principality and completeness

The inference algorithm presented here – gathering the constraints from an expression followed by solving them – satisfies the usual properties of soundness, principality, and completeness with respect to the declarative specification.

In order to state the theorems we need some ancillary notions. A constraint is in *solved form* if it consists only of quantification and equality constraints ($\upsilon \sim \sigma$); and the equalities constitute a well-sorted idempotent substitution of its unification variables. For example

$$\exists \alpha^{\mathfrak{m}}.(\alpha^{\mathfrak{m}} \sim \mathsf{Int}) \wedge (\forall b. \exists \beta^{\mathfrak{u}}.\, \beta^{\mathfrak{u}} \sim (b \to \mathsf{Int}))$$

is in solved form. Being in solved form is more than a simple syntactic property; here are two constraints that are not:

$$\exists \alpha.(\alpha \sim \mathsf{Int}) \wedge (\alpha \sim \mathsf{Bool}) \qquad \exists \alpha. \ldots (\forall b.\alpha \sim [b]) \ldots$$

In the first there are two equalities for α (we should apply EQSUBST to make progress); in the second, there is a skolem-escape problem. However it is OK for a unification variable to have *no* equalities; it is simply unconstrained.

Figure 9 defines solved form precisely. We keep a set of variables $\overline{\beta}$ for which we ensure that there is precisely one equality constraint, and another set $\overline{\alpha}$ (the unconstrained variables) for which there are none. Rule SolvedVar expects precisely one β, checks well-sortedness, and also checks that σ does not mention any variables other than the skolems and unconstrained unification variables – the latter check ensures idempotence. Rule SolvedConj partitions the $\overline{\beta}$ between the two conjunctions. Rule SolvedQuant partitions the local existentials $\overline{\upsilon}$ into the unconstrained sets, γ_1 and γ_2 resp. Rule Solved⊤ simply states that ⊤ is a solved form without consuming any variables.

A second auxiliary notion which we need to state the results is *substitution induced by a solved form.*

$$\frac{C_s = E \wedge R, \text{ where } E \text{ are all the equalities in } C_s}{\widehat{C_s} = [\overline{\alpha \mapsto \sigma} \mid \alpha \sim \sigma \in E]}$$

Theorem 4.2 (Soundness)**.** *Let* Γ *be a closed environment and* e *an expression. If* $\Gamma \vdash e : \sigma \rightsquigarrow C$ *and* C_s *is a solution for* C *with an induced substitution* $\widehat{C_s}$*, then we have* $\Gamma \vdash e : \widehat{C_s}(\sigma)$*.*

Theorem 4.3 (Principality)**.** *Suppose* $\Gamma \vdash e : \sigma$*. Then there exists a type* ϕ *such that* $\Gamma \vdash e : \phi$*, and for every other* $\Gamma \vdash e : \sigma'$*, there is a fully monomorphic substitution* π *such that* $\sigma' = \pi\phi$*.*

Theorem 4.4 (Completeness)**.** *Let* Γ *be a closed environment and* e *an expression. If* $\Gamma \vdash e : \sigma$ *then* $\Gamma \vdash e : \phi \rightsquigarrow C$ *and* C *can reach a solved form.*

Proofs of these results are given in Appendix D.2.

4.5 Alternative solver for equalities

Unfortunately, once we extend the language of types, by introducing type classes and local assumptions, the approach by Pottier and Rémy [15] is no longer applicable. With that in mind, we introduce a *different* approach to solve the problem of unification under a mixed prefix, which *does* scale to handle these extensions. The approach is rather simple – a single rule FLOAT in Figure 10 – and is directly inspired by how GHC handles constraints: by floating constraints out from inside a quantification constraint. When can we do that? Precisely when the constraint does not mention the skolems. But what about the existentials? Consider

$$\exists\alpha.\ldots(\forall a.\exists\beta.(\alpha \sim [\beta]) \wedge C)\ldots$$

We would like to float the constraint $(\alpha \sim [\beta])$ out of the quantification constraint, but then β would be out of scope. We can solve this by "promoting" β: producing a fresh β' that lives in the outer scope, and making β equal to it, thus:

$$\exists\alpha,\beta'.\ldots(\alpha \sim [\beta'])) \wedge (\forall a.\exists\beta.(\beta \sim \beta' \wedge C))\ldots$$

All this is expressed directly by rule FLOAT. If we cannot float, we have a skolem escape error; for example, consider:

$$\exists\alpha.\ldots(\forall a.\exists\beta.(\alpha \sim [a]) \wedge C)\ldots$$

Here we cannot float $(\alpha \sim [a])$ because it mentions the skolem a, so an inner skolem has leaked into an outer scope (α is bound further out). Floating makes manifest that skolem escape has not happened, and brings the constraint nearer to solved form.

Conjecture 4.5. *The solver presented in Figure 8 is complete for unification problems under a mixed prefix.*

5 Practical matters

We have implemented a prototype of the type inference process described in this paper, including support for Haskell's type classes by extending GI as described in Appendix B. The expressions in Figure 2 are accepted or rejected as described by the table, in the GI column.

GI does not support any co- or contra-variance in function types. For example, the constraint $Int \to (\forall a.\, a \to a) \leqslant Int \to Int \to Int$ does not hold. In contrast, GHC with the *RankNTypes* extension supports some amount of variance. Libraries such as Scrap Your Boilerplate use this fact very often in definitions with a $\forall$ to the right of an arrow:

$$f :: \forall a.\ a \to (\forall b.\ b \to b)$$
$$f\ x\ y = y$$

Other uses of variance can be worked around by η-expansion. Consider $(flip\ f)$, where $flip$'s type is in Figure 1. This is ill-typed because $flip$ requires an argument of type $a \to b \to c$, but f's type, after instantiation, looks like $\tau \to \forall b.\, b \to b$. The fix is simple: just η-expand the argument, thus $(flip\ (\lambda x \to f\ x))$. GHC does this automatically at the moment, but in fact this η-expansion is unsound in general, since a change in the laziness behavior can be observed.

One might worry that if GI is integrated in GHC many existing Haskell libraries would need to be modified. To quantify this impact, we modified GHC to impose those restrictions and rebuilt all the packages in Stackage which require the *RankNTypes* extension. In order to minimize the annotation burden, we added a simple special case to support function definition in the style of Scrap Your Boilerplate.

With this done, very few packages required modifications, and modifications were always η-expansions. In particular, of the 2,400 packages in Stackage, 609 use *RankNTypes*; of these, only 75 required manual changes, all of which were simple η-expansions. One (*singletons*) would require larger changes, because it uses Template Haskell to *generate* Haskell code; so it needs to generate η-expanded code. Two more failed for reasons we have yet to investigate. Our conclusion is that the impact of our proposed changes is extremely minor, especially since GHC's current covert η-expansion strategy is unsound in the first place.

6 Related work

Full type inference for System F is undecidable [24] – partial type inference with known generalisation positions but unknown instantiations can be reduced to higher-order unification [14]. System F lacks principal types, making modular type inference and the addition of ML-style let-bindings impossible. Higher-rank type inference with *predicative* instantiation has some successful solutions [4, 13, 17], exploiting a mix of annotation propagation and unification.

On the other hand, no solution for impredicativity with a good benefit-to-weight ratio has been presented to-date.

$$[\textsc{float}]\ \frac{\mathrm{ftv}(F) \cap \overline{a} = \emptyset \qquad \overline{\alpha^{s}} = \mathrm{fuv}(F) \cap v_{\mathsf{in}} \qquad \overline{\gamma^{s}}\ \text{fresh} \qquad \overline{E} = \bigwedge_{\alpha^{s} \in \overline{\alpha}} \alpha^{s} \sim \gamma^{s}}{\forall \overline{a}.\ \exists \overline{v}_{\mathsf{in}}.\ (C \wedge F)\ ;\ \overline{v} \implies [\overline{\alpha^{s} \mapsto \gamma^{s}}]F \wedge \forall \overline{a}.\ \exists \overline{v}_{\mathsf{in}}.\ (C \wedge E)\ ;\ \overline{v}, \overline{\gamma^{s}}}$$

Figure 10. Solving rule for quantification constraints

MLF [1, 2, 18] is an *extension* of System F based on quantification with instance and equality bounds. The resulting system is powerful, but also quite complex to implement; in return we get back principal types. There have been several attempts to simplify the user-facing part of MLF to System F types. FPH [23] exposes a "box" structure around inferred types (that would be hidden under a constraint in MLF). Flexible types [9], also known as HML, avoid quantification over equality constraints. Implementing these systems in a working compiler is a significant undertaking, and so is the integration with features like type classes[10].

For this reason, there are proposals for algorithms simpler than MLF. Boxy Types [22] is an early attempt to push bidirectional inference to allow impredicativity, but resulted in a complex specification. HMF [8] imposes universal conditions on typing derivations to recover principal types. QML [19], inspired by boxed polymorphism [12, 16], introduces an only-explicitly-instantiable ∀. Recent work by Eisenberg et al. [5] also proposes a distinction between an implicitly and an explicitly instantiable ∀; only the latter is impredicative.

We return now to the table in Figure 2, where we present a collection of examples appearing in selected related works on type inference for impredicativity. We have selected those systems because they all strike a good balance between expressivity and requiring very few type annotations. The table shows the flexibility/expressivity price we pay in order to keep the implementation and specification costs low, and avoid the introduction of new type system features (such as types with constraints or boxes) or new forms of annotations (as in QML). We also show how to recover the typeability of a program in cases where a valid type exists.

In the table we see that MLF can type all programs that do not require implicit η-expansion ($k\ h\ lst$) or the use of a polymorphic function argument at two types ($\lambda f.\,(f\ 1, f\ True)$), but uses constrained types and a unification algorithm that is not straightforward to integrate in a type inference engine with elaboration to System F. HML – a simplification of MLF with only "flexible" type bounds in constraints – is only slightly less powerful than MLF. HML cannot type $\lambda xs.\,poly\ (head\ xs)$ because xs would have to be assigned a polymorphic type, even though it's only used at this one type. FPH – which is based on System F types – is equally expressive to these systems for the fragment of System F consisting only of variables and applications. However, the FPH treatment of λ-abstractions requires the returned type to be a fully-resolved top-level monomorphic type. Therefore, FPH fails to type check *choose id auto* because *auto* will be assigned the same type as GI infers ($(\forall a.\ a \to a) \to b \to b$). GI – modulo the quirk about not generalizing the bodies of lambda abstractions that only MLF and HML can tackle – type checks almost the same programs as those systems. To determine why a program fails to type check (and how it should be fixed) it suffices to determine whether some function has been instantiated to a type with top-level polymorphism in its arguments. For example, $f\ (choose\ id)\ ids$ fails to type check because it requires the instantiation of *choose* to $(\forall a.\ a \to a) \to (\forall a.\ a \to a) \to (\forall a.\ a \to a)$ and no argument has a type with a top-level constructor.

HMF is based on local decisions about polymorphic instantiations and – without an extension to n-ary applications – fails to type check programs where the local instantiation has to be delayed to take more arguments into account (e.g. fails to type check $id : ids$), though it does generalize in argument positions (e.g. *snoc id ids* is accepted, for $snoc :: \forall a.\ [a] \to a \to [a]$). Leijen [8] proposes an extension of the basic algorithm to n-ary applications that makes these examples type check: after the function type is instantiated enough to cover all the arguments, we proceed to type check the arguments in *a computed order*, instead of left-to-right. The arguments that must be type checked against a naked type variable coming from the instantiated function type are postponed and checked last, in the hope that the rest of the arguments will by that time determine any impredicative instantiations. The procedure is iterated, possibly uncovering impredicative instantiations in each round. Under that extension $id : ids$ is accepted. Alas, the new algorithm is not accompanied with a declarative specification. Our system achieves a similar effect (and we conjecture is equally expressive), thanks to the delaying we get from the use of constraints. In the end, the main important difference of HMF with GI is that GI provides a declarative specification and an algorithm that easily integrates in a pre-existing constraint-based type inference engine.

7 Further work

GI seeks a sweet spot that balances *simplicity* with *expressiveness*. We are also exploring some nearby variants. For example, extending VARGEN to handle larger expressions would allow us to accept examples A9, C8, C9 in Figure 2; and by improving skolemisation we could accept E3. It remains to be seen whether the extra expressiveness justifies the extra complexity, but GI seems to be an encouragingly robust base camp.

References

[1] Didier Le Botlan and Didier Rémy. 2003. ML^{F}: raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 27–38. https://doi.org/10.1145/944705.944709

[2] Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Inf. Comput.* 207, 6 (2009), 726–785.

[3] Hubert Comon and Pierre Lescanne. 1988. *Equational problems and disunification.* Research Report RR-0904. INRIA. https://hal.inria.fr/inria-00075652

[4] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. https://doi.org/10.1145/2500365.2500582

[5] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag New York, Inc., New York, NY, USA, 229–254. https://doi.org/10.1007/978-3-662-49498-1_10

[6] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the type inference process. *SIGPLAN Notices* 38, 9 (2003), 3–13. https://doi.org/10.1145/944746.944707

[7] James Hook and Peter Thiemann (Eds.). 2008. *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008.* ACM.

[8] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism, See [7], 283–294. https://doi.org/10.1145/1411204.1411245

[9] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 66–77. https://doi.org/10.1145/1480881.1480891

[10] Daan Leijen and Andres Löh. 2005. Qualified types for MLF. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 144–155. https://doi.org/10.1145/1086365.1086385

[11] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282. https://doi.org/10.1145/357162.357169

[12] J. W. O'Toole, Jr. and D. K. Gifford. 1989. Type Reconstruction with First-class Polymorphic Values. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89).* ACM, New York, NY, USA, 207–217. https://doi.org/10.1145/73141.74836

[13] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82.

[14] Frank Pfenning. 1988. Partial Polymorphic Type Inference and Higher-order Unification. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88).* ACM, New York, NY, USA, 153–163. https://doi.org/10.1145/62678.62697

[15] François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

[16] Didier Rémy. 1994. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '94).* Springer-Verlag, London, UK, UK, 321–346. http://dl.acm.org/citation.cfm?id=645868.668492

[17] Didier Rémy. 2005. Simple, Partial Type-inference for System F Based on Type-containment. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05).* ACM, New York, NY, USA, 130–143. https://doi.org/10.1145/1086365.1086383

[18] Didier Rémy and Boris Yakobowski. 2008. From ML to ML^{F}: graphic type constraints with efficient type inference, See [7], 63–74. https://doi.org/10.1145/1411204.1411216

[19] Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-class Polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML (ML '09).* ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1596627.1596630

[20] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[21] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

[22] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 251–262. https://doi.org/10.1145/1159803.1159838

[23] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell, See [7], 295–306. https://doi.org/10.1145/1411204.1411246

[24] J. B. Wells. 1993. *Typability and Type Checking in the Second-Order Lambda-Calculus Are Equivalent and Undecidable.* Technical Report. Boston, MA, USA.

[25] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 12–21. https://doi.org/10.1145/2737924.2738009

$$\begin{array}{llcl} \text{Data constructor} & & \ni & K \\ \text{Expressions / terms} & e & ::= & \dots \mid \textbf{case}\ e_0\ \textbf{of}\{\overline{K\,\overline{x} \to e}\} \end{array}$$

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : \sigma_0 \qquad \sigma_0 \leqslant^{m}_{\epsilon} \epsilon; \mathsf{T}\,\overline{\phi_0} \\ \text{for each branch } K_i\,\overline{x_i} \to e_i \text{ with} \\ \underline{K_i : \forall \overline{a}\,\overline{b}.\,\overline{\sigma_i} \to \mathsf{T}\,\overline{a} \in \Gamma} \\ \Gamma, x_i : [\overline{a \mapsto \phi_0}]\sigma_i \vdash e_i : \phi_\star \end{array}}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\{\overline{K\,\overline{x} \to e}\} : \phi_\star}\ \textsc{Case}$$

Figure 11. Decl. type system with pattern matching

A Pattern matching

Introducing pattern matching in the language gives no surprises, as witnessed by the rule Case in Figure 11. We first need to check that the scrutinized expression e_0 can be given a type compatible with the indicated data constructor. It is important to note that the mere presence of the data constructors is enough to know the type constructor T, there is no inference at that point. Then we introduce new term variables in each branch, whose type is obtained by combining the type of the constructor with the inferred values for the type variables in T, namely $\overline{\phi_0}$. The return type of every branch should be equal, $\phi_\star$ (note that we do *not* allow the types of the branches to be instantiated).

The corresponding Case rule for constraint gathering is given in Figure 12. In this case we need a quantification constraint to introduce new skolem constants, for the existentially quantified type variables $\overline{b}_i$ in the data constructor.

B Integration with other language features

One of the main advantages of organising a type checker around the concept of constraints is that extensions to the type system can be accommodated for quite easily. Indeed, this is one of the main advantages of our system over existing work. In this section we describe how to deal with other forms of constraint beyond the standard equality constraints we now support.

For example, Haskell supports *type classes*, which restrict the scope of a polymorphic abstraction to a subset of types. The archetypal example is *Eq*, which describes the types with support for decidable equality. Such a type class constraint is visible in the type of the equality operator $(\equiv) :: \forall\, a.\, Eq\ a \Rightarrow a \to a \to Bool$. Similarly, languages like OCaml and PureScript feature *row* (or *record*) *types* like $\{\, x :: Point, y :: Point \,\}$ in addition to usual ADTs.

The good news is that a constraint-based formulation of typing makes it easy to cope with new concepts if they can also be described in terms of constraints. Several examples in the literature, like HM(X) [15] and OutsideIn(X) [21], are actually frameworks which can be parametrized by different constraint systems, hence the X in their names.

The modifications needed to accommodate the new kinds of constraints in GI are given in Figure 13. First, we split up the syntax of constraints into so-called simple and extended constraints. The former consists of constraints that can be used by programmers in their programs, while the second category consists of additional constraints that are internal to the solver. Syntax for any new kind of constraints can be added to the syntax of simple constraints Q; we have in fact done so for the specific example of type classes. We also modify the syntax of polymorphic types so that they may contain a number of simple constraints.

Now, suppose that we need to check that $\forall\, a.\, Ord\ a \Rightarrow a \to Bool$ is an instance of $\forall\, a.\, Eq\ a \Rightarrow a \to Bool$, in other words $(\mathbb{V}\{\alpha\}.\, Eq\ \alpha \Rightarrow \alpha \to Bool) \preceq (\forall\, a.\, Ord\ a \Rightarrow a \to Bool)$. During this process we are allowed to *assume* that $Ord\ a$ holds in order to discharge $Eq\ \alpha$. To be able to store this information in our extended constraints, we modify the syntax of quantification constraints to include the assumed information as part of an implication, in this case

$$\forall a.\, (Ord\ a \supset \mathbb{V}\{\alpha\}.\, Eq\ \alpha \Rightarrow \alpha \to Bool \preceq a \to Bool)$$

Implication constraints are also introduced by the updated rules for constraint gathering. An annotated application – rule AnnApp – may also mention constraints, which are assumed while checking the enclosed expression. Rule Case allows some constraints to be locally valid, which means that the system gains support for generalized algebraic data types (GADTs).

Figure 14 presents the updates needed for the solver to cope with this new form of constraints. The solver now relates two sets of constraints: the *assumptions* and the *wanted* constraints. Whenever we go under an implication, we introduce the constraints that are part of the antecedent as additional assumptions. The solver is allowed to rewrite in both sets, and more importantly, to use an assumed constraint to rewrite a wanted one. The modifications to the rules are very much in line with Vytiniotis et al. [21].

The rules that dealt with polymorphic types, namely inst∀l and inst∀r need to be updated. In the former case, a constraint in the type becomes an obligation to prove that it holds. In the latter case the constraints become assumptions that we are allowed to use while solving; these are stored in an implication constraint. Note that the guardedness restrictions do not change by the introduction of constraints.

Floating of constraints and promotion of variables also require changes, as described in rule float. In particular, we are only allowed to float equality constraints, which in the vanilla system were the only kind of simple constraints. The reason is that other kinds of constraints may incorporate information of the assumptions while they are solved, and these assumptions only hold in their respective branches. Equality constraints do not pose this problem.

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow C}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \rightsquigarrow C_0 \qquad \overline{\alpha}, \beta \text{ fresh} \qquad \begin{array}{l}\text{for each branch } K_i\,\overline{x_i} \longrightarrow e_i \\ \quad \underline{K_i : \forall \overline{a}\,\overline{b}_i.\,\overline{\sigma_i} \longrightarrow \mathsf{T}\,\overline{a} \in \Gamma} \\ \quad \Gamma, x_i : \overline{[\overline{a \mapsto \alpha^{\mathrm{u}}}]\sigma_i} \vdash e_i : \phi_i \rightsquigarrow C_i \\ \quad \overline{v}_i = \mathsf{fuv}(\phi_i, C_i) - \mathsf{fuv}(\Gamma) - \overline{\alpha}\end{array}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\,\{\overline{K\,\overline{x} \longrightarrow e}\} : \beta^{\mathrm{u}} \rightsquigarrow C_0 \wedge (\sigma_0 \leqslant^{\mathrm{m}}_{\epsilon} \epsilon; \mathsf{T}\,\overline{\alpha^{\mathrm{u}}}) \wedge \forall \overline{b}_i.\,\exists \overline{v}_i.\,(C_i \wedge \beta^{\mathrm{u}} \sim \phi_i)}\ \textsc{Case}$$

Figure 12. Constraint generation for pattern matching

$$\begin{array}{llcll}
\text{Polymorphic types} & \sigma, \phi & ::= & \alpha^{\mathrm{u}} \mid \forall \overline{a}.\ Q \Rightarrow \mu & \\
\text{Simple constraints} & Q & ::= & \top \mid Q_1 \wedge Q_2 & \\
 & & \mid & \sigma \sim \phi & \text{Equality} \\
 & & \mid & \ldots & \text{Open for extension} \\
 & & \mid & C\ \sigma_1 \ldots \sigma_n & \text{Type classes, for example} \\
\text{Extended constraints} & C & ::= & Q \mid C_1 \wedge Q_2 & \\
\textit{Instantiation} & & \mid & \sigma \leqslant^{\mathrm{s}}_{\overline{\omega}} \overline{\phi}; \mu & \\
\textit{Generalisation} & & \mid & \mathfrak{g} \preceq \sigma & \\
\textit{Quantification and implication} & & \mid & \forall \overline{a}.\,\exists \overline{v}.\,(Q \supset C) &
\end{array}$$

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow C}$$

$$\frac{\vdots \qquad \vdots}{\Gamma \vdash (h\,e_1\ \ldots\ e_n :: \forall \overline{b}.\,Q \Rightarrow \eta) : \forall \overline{b}.\,Q \Rightarrow \eta \rightsquigarrow \forall \overline{b}.\,\exists \overline{v}.\,(Q \supset C \wedge \ldots)}\ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \rightsquigarrow C_0 \qquad \overline{\alpha}, \beta \text{ fresh} \qquad \begin{array}{l}\text{for each branch } K_i\,\overline{x_i} \longrightarrow e_i \\ \quad \underline{K_i : \forall \overline{a}\,\overline{b}_i.\,Q_i \Rightarrow \overline{\sigma_i} \longrightarrow \mathsf{T}\,\overline{a} \in \Gamma} \\ \quad \Gamma, x_i : \overline{[\overline{a \mapsto \alpha^{\mathrm{u}}}]\sigma_i} \vdash e_i : \phi_i \rightsquigarrow C_i \\ \quad \overline{v}_i = \mathsf{fuv}(\phi_i, C_i) - \mathsf{fuv}(\Gamma) - \overline{\alpha}\end{array}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\,\{\overline{K\,\overline{x} \longrightarrow e}\} : \beta^{\mathrm{u}} \rightsquigarrow C_0 \wedge (\sigma_0 \leqslant^{\mathrm{m}}_{\epsilon} \epsilon; \mathsf{T}\,\overline{\alpha^{\mathrm{u}}}) \wedge \forall \overline{b}_i.\,\exists \overline{v}_i.\,(Q_i \supset C_i \wedge \beta^{\mathrm{u}} \sim \phi_i)}\ \textsc{Case}$$

Figure 13. Extensions for integration with other constraints

The $C\ ;\ \overline{v} \implies C'\ ;\ \overline{v}'$ relation only deals with one set of constraints. But as implications enter the game, we need to consider the interaction between two sets: the assumed and the wanted ones. For that matter we introduce a new rewriting judgment $Q_a\ ;\ C\ ;\ \overline{v} \implies Q'_a\ ;\ C'\ ;\ \overline{v}'$, where assumed constraints Q_a are rewritten to Q'_a and wanted constraints C to C'. Note that set of assumed constraints consists only of simple constraints.

Rules ASSUMED and WANTED allow us to apply any rule to any of the two sets involved in solving. Rule INTERACT deals with the interaction of an assumption and a wanted constraint; the result is put on the wanted set. For example, if you have $\alpha \sim \mathit{Int}\ ;\ \beta \sim [\alpha]\ ;\ \alpha, \beta$, the rule moves to $\alpha \sim \mathit{Int}\ ;\ \beta \sim [\mathit{Int}]\ ;\ \alpha, \beta$. Rule DUPL is a simple case of interaction, in which a constraint in the wanted set is "crossed out" if it is already in the assumed set. Note that information may only flow from assumptions to wanted constraints, and never the other way around.

The last rule, an updated version of FORALL, is responsible for dealing with implication constraints. The constraints C inside of the implication are rewritten in an environment where the set of assumed constraints is enlarged with the antecedent Q from the implication.

C From and to System F

Our system is as expressive as System F, the gold standard for a fully-expressive impredicative type system. We show so by describing a translation from every System F expression to GI in Figure 15. We also give the converse translation from GI terms to System F terms in Figure 16 (as usual, ϵ denotes an empty list). In particular, this proves that GI is sound as a type system. For the sake of conciseness we do not include the translations of pattern matching.

$$\boxed{C\,;\overline{v} \implies C'\,;\overline{v}'}$$

$$[\text{INST}\forall\text{L}] \quad (\forall\overline{a}.\ Q \Rightarrow \mu) \leqslant^{\mathfrak{s}}_{\omega} \overline{\phi};\eta\,;\overline{v} \implies \overline{[a \mapsto v']}Q \wedge (\mu' \leqslant^{\mathfrak{s}}_{\omega} \overline{\phi};\eta)\,;\overline{v},\overline{v}'$$
$$\text{where } \mathsf{freshen}^{\mathfrak{s}}_{\omega}(\forall\overline{a}.\,\mu) \implies \langle \overline{v}', \mu' \rangle$$

$$[\text{INST}\forall\text{R}] \quad \mathfrak{g} \leq (\forall\overline{a}.\ Q \Rightarrow \mu)\,;\overline{v} \implies \forall\overline{a}.(\,Q \supset \mathfrak{g} \leq \mu)\,;\overline{v}$$

$$[\text{FLOAT}]\ \frac{F \text{ is an equality} \qquad \mathsf{ftv}(F) \cap \overline{a} = \emptyset \qquad \overline{\alpha^{\mathfrak{s}}} = \mathsf{fuv}(F) \cap v_{\mathsf{in}} \qquad \overline{\gamma^{\mathfrak{s}}} \text{ fresh} \qquad \overline{E} = \bigwedge_{\alpha^{\mathfrak{s}} \in \overline{\alpha}} \alpha^{\mathfrak{s}} \sim \gamma^{\mathfrak{s}}}{\forall\overline{a}.\,\exists\overline{v}_{\mathsf{in}}.\,(Q \supset C \wedge F)\,;\overline{v} \implies [\overline{\alpha^{\mathfrak{s}} \mapsto \gamma^{\mathfrak{s}}}]F \wedge \forall\overline{a}.\,\exists\overline{v}_{\mathsf{in}}.\,(Q \supset C \wedge E)\,;\overline{v},\overline{\gamma^{\mathfrak{s}}}}$$

$$\boxed{Q_a\,;C\,;\overline{v} \implies Q'_a\,;C'\,;\overline{v}}$$

$$[\text{ASSUMED}]\ \frac{Q_a\,;\overline{v} \implies Q'_a\,;\overline{v}_{\mathsf{forget}}}{Q_a\,;C\,;\overline{v} \implies Q'_a\,;C\,;\overline{v}}$$

$$[\text{WANTED}]\ \frac{C\,;\overline{v} \implies C'\,;\overline{v}'}{Q_a\,;C\,;\overline{v} \implies Q_a\,;C'\,;\overline{v}'}$$

$$[\text{INTERACT}]\ \frac{Q \wedge C^*\,;\overline{v} \implies C'\,;\overline{v}'}{Q_a \wedge Q\,;C \wedge C^*\,;\overline{v} \implies Q_a \wedge Q\,;C \wedge C'\,;\overline{v}'}$$

$$[\text{DUPL}]\ \frac{}{Q_a \wedge Q\,;C \wedge Q\,;\overline{v} \implies Q_a \wedge Q\,;C\,;\overline{v}}$$

$$[\text{FORALL}]\ \frac{Q_a \wedge Q\,;C\,;\overline{v}_{\mathsf{in}} \implies Q'\,;C'\,;\overline{v}'_{\mathsf{in}}}{Q_a\,;\forall\overline{a}.\,\exists\overline{v}_{\mathsf{in}}.\,(Q \supset C)\,;\overline{v} \implies Q_a\,;\forall\overline{a}.\,\exists\overline{v}'_{\mathsf{in}}.\,(Q \supset C')\,;\overline{v}}$$

Figure 14. Solving rules for implication constraints

Theorem C.1 (Embedding of System F). *Let e be a System F expression. If $\Gamma \vdash^{\mathsf{F}} e_{\mathsf{F}} : \sigma \rightsquigarrow e'$, as defined in Figure 15, then $\Gamma \vdash e' : \sigma$ in* GI.

The main difference between GI and System F is that in the former impredicative instantiation is restricted by guardedness. The presented translation relies on annotations to work around them.

- Following GI we present n-ary application – which in the case of System F also includes type application. The given application rule only works if the guardedness restrictions are satisfied, otherwise an annotation on e_0 needs to be added before applying the rule.
- In GI the result of a non-annotated application always gets a top-level monomorphic type. This is not the case in System F, and thus an annotation may be required to further generalize.
- Completely unrestricted instantiation is only available via annotations. In order to apply this translation, we need to split applications so that guardedness guarantees are always met. Every time we split, we introduce a new annotation guiding the type checking process.

Proof. By induction on the typing derivation in System F, $\Gamma \vdash^{\mathsf{F}} e : \sigma$.

Variable. We need to derive $\Gamma \vdash x : \sigma$, for a general $\sigma = \forall\overline{a}.\,\mu$. We recall that single variables are treated like 0-ary application, so some amount of instantiation must take place.

$$\frac{\dfrac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \sigma}\ \textsc{VarHead} \qquad \sigma \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon;\mu}{\Gamma \vdash x : \mu}\ \textsc{App}$$

If the set of quantified variables $\overline{a}$ is empty, then this derivation is all we need. If it is not, we need an annotation to re-generalise those.

$$\frac{\dfrac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \sigma}\ \textsc{VarHead} \qquad \sigma \leqslant^{\mathfrak{u}}_{\epsilon} \epsilon;\mu}{\Gamma \vdash (x :: \forall\overline{a}.\,\mu) : \forall\overline{a}.\,\mu}\ \textsc{AnnApp}$$

We could get a smaller translation in some cases by using the VarGen rule, but the simpler type system with annotations is enough for our goals.

Abstraction. By induction hypothesis we are able to type check the body of the abstraction. The distinction between fully monomorphic and unrestricted types in the translation ensures that we can use the right Abs or AnnAbs rule from the declarative specification.

$$\text{System F terms} \quad e_\mathsf{F} ::= x \mid \lambda(x :: \sigma).\, e_\mathsf{F} \mid \Lambda a.\, e_\mathsf{F} \mid e_\mathsf{F}\, e_\mathsf{F} \mid e_\mathsf{F}\, \sigma$$

$$\boxed{\Gamma \vdash^\mathsf{F} e_\mathsf{F} : \sigma \rightsquigarrow e'}$$

$$\frac{x : \mu \in \Gamma}{\Gamma \vdash^\mathsf{F} x : \mu \rightsquigarrow x} \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash^\mathsf{F} x : \sigma \rightsquigarrow (x :: \sigma)}$$

$$\frac{\Gamma, x : \tau \vdash^\mathsf{F} e_\mathsf{F} : \phi \rightsquigarrow e'}{\Gamma \vdash^\mathsf{F} \lambda(x :: \tau).\, e_\mathsf{F} : \tau \to \phi \rightsquigarrow \lambda x.\, e'} \qquad \frac{\Gamma, x : \sigma \vdash^\mathsf{F} e_\mathsf{F} : \phi \rightsquigarrow e'}{\Gamma \vdash^\mathsf{F} \lambda(x :: \sigma).\, e_\mathsf{F} : \sigma \to \phi \rightsquigarrow \lambda(x :: \sigma).\, e'}$$

$$\frac{\Gamma \vdash^\mathsf{F} e_\mathsf{F} : \phi[a_1, \dots, a_n] \rightsquigarrow e'}{\Gamma \vdash^\mathsf{F} \Lambda a_1. \dots \Lambda a_n.\, e_\mathsf{F} : \forall a_1. \dots \forall a_n.\, \phi \rightsquigarrow (\lfloor e' \rfloor :: \forall a_1 \dots a_n.\, \phi)}$$

$$\frac{\begin{array}{c} \Gamma \vdash^\mathsf{F} e_0 : \sigma \rightsquigarrow e'_0 \qquad \sigma_0 = \forall \overline{a}_1.\, \sigma_1 \to \forall \overline{a}_2.\, \sigma_2 \to \dots \forall \overline{a}_n.\, \sigma_n \to \forall \overline{a}_r\, \overline{a}_p.\, \mu_r \\ \sigma_0 \leqslant^{\mathrm{u}}_{\underbrace{\bullet, \dots, \bullet}_{n \text{ times}}} [\overline{a_1 \mapsto \phi_1}]\sigma_1, [\overline{a_1 \mapsto \phi_1}, \overline{a_2 \mapsto \phi_2}]\sigma_2, \dots, [\overline{a_1 \mapsto \phi_1}, \dots, \overline{a_n \mapsto \phi_n}; \overline{a_r \mapsto \phi_r}]\mu_r \\ \Gamma \vdash^\mathsf{F} e_1 : [\overline{a_1 \mapsto \phi_1}]\sigma_1 \rightsquigarrow e'_1 \quad \dots \quad \Gamma \vdash^\mathsf{F} e_n : [\overline{a_1 \mapsto \phi_1}, \dots, \overline{a_n \mapsto \phi_n}]\sigma_n \rightsquigarrow e'_n \end{array}}{\Gamma \vdash^\mathsf{F} e_0\, \overline{\phi}_1\, e_1\, \overline{\phi}_2\, e_2\, \dots\, \overline{\phi}_n\, e_n\, \overline{\phi}_r : \forall \overline{a}_r.\, \mu_r \rightsquigarrow \begin{cases} e'_0\, e'_1 \dots e'_n & \text{if } \overline{a}_p \text{ is empty} \\ & \text{and } \overline{\phi}_r \text{ fully mono.} \\ (e'_0\, e'_1 \dots e'_n :: \forall \overline{a}_r.\, \mu_r) & \text{otherwise} \end{cases}}$$

$$\text{where} \begin{cases} \lfloor e :: \sigma \rfloor = \lfloor e \rfloor \\ \lfloor e \rfloor = e & \text{otherwise} \end{cases}$$

Figure 15. Translation from System F

Type abstraction. There are two cases to consider:

- $\lfloor e' \rfloor$ *is not an application.* In this case the annotation is treated as an annotated application with zero arguments. By induction hypothesis, we know that $\Gamma \vdash e' : \phi[\overline{a}]$. We assume without loss of generality that $\phi = \forall \overline{b}.\, \mu$, and thus $\Gamma \vdash e' : \forall \overline{b}.\, \mu[\overline{a}]$. Now we can build the following derivation:

$$\frac{\dfrac{\Gamma \vdash e' : \forall \overline{b}.\, \mu[\overline{a}]}{\Gamma \vdash^{\mathsf{fun}} e' : \forall \overline{b}.\, \mu[\overline{a}]} \qquad \forall \overline{b}.\, \mu[\overline{a}] \leqslant^{\mathrm{u}}_{\epsilon} \epsilon; \mu[\overline{a}]}{\Gamma \vdash (e' :: \forall \overline{a}\, \overline{b}.\, \mu) : \forall \overline{a}\, \overline{b}.\, \mu} \textsc{AnnApp}$$

- $\lfloor e' \rfloor$ *is an application.* In this case e' results from the application of either App or AnnApp in the declarative specification. By inspection of the rule AnnApp, we can see that we can always choose to quantify over more variables via an annotation.

(Type) application. The premises about guardedness ensure that we can apply the rule App or AnnApp in the declarative specification. As discussed in the main text, if at a certain application we cannot apply this rule, we can always split the application, annotate the head and then use the application rule again. □

D Proofs

D.1 Declarative specification

Lemma D.1. *Let Γ be an environment and e an expression. For every pair of fully monomorphic substitutions θ_1 and θ_2, if $\theta_1\Gamma \vdash^{\mathsf{fun}} e : \sigma_1$ and $\theta_2\Gamma \vdash^{\mathsf{fun}} e : \sigma_2$, then there exists a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*

Theorem 3.2 (Impredicative instantiation is not guessed)**.** *Let Γ be an environment and e an expression. For every pair of fully monomorphic substitutions θ_1 and θ_2, if $\theta_1\Gamma \vdash e : \sigma_1$ and $\theta_2\Gamma \vdash e : \sigma_2$, then there exists a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*

Proof. We prove Lemma D.1 and Theorem 3.2 by mutual induction over the typing derivation of e.

Proof of Lemma D.1. We distinguish two cases:

- *Case* VarHead. The two derivations to consider are:

$$\frac{x : \theta_1\sigma \in \theta_1\Gamma}{\theta_1\Gamma \vdash^{\mathsf{fun}} x : \theta_1\sigma} \qquad \frac{x : \theta_2\sigma \in \theta_2\Gamma}{\theta_2\Gamma \vdash^{\mathsf{fun}} x : \theta_2\sigma}$$

  Thus we can take $\sigma^* = \sigma$ and as substitutions those applied to Γ, which are fully monomorphic by our assumptions.
- *Case* ExprHead. Follows by induction over the premise.

$$\boxed{\sigma \ \leqslant^{\mathsf{s}}_{\overline{\omega}} \ \overline{\phi}; \mu \rightsquigarrow \overline{\psi_1}, \ldots, \overline{\psi_n}, \overline{\psi_r}}$$

$$\frac{}{\mu \ \leqslant^{\mathsf{s}}_{\epsilon} \ \epsilon; \mu \rightsquigarrow \epsilon} \ \textsc{InstMono}$$

$$\frac{\phi_2 \ \leqslant^{\mathsf{s}}_{\overline{\omega}} \ \sigma_2, \ldots, \sigma_n; \mu \rightsquigarrow \overline{\psi_2}, \ldots, \overline{\psi_n}, \overline{\psi_r}}{\phi_1 \rightarrow \phi_2 \ \leqslant^{\mathsf{s}}_{\omega, \overline{\omega}} \ \phi_1, \sigma_2, \ldots, \sigma_n; \mu \rightsquigarrow \epsilon, \overline{\psi_2}, \ldots, \overline{\psi_n}, \overline{\psi_r}} \ \textsc{InstArrow}$$

$$\frac{\forall \overline{a}.\, \mu \rhd^{\mathsf{s}}_{\overline{\omega}} \Delta \qquad \theta \text{ respects } \Delta \qquad \theta\mu \ \leqslant^{\mathsf{s}}_{\overline{\omega}} \ \sigma_1, \ldots, \sigma_n; \eta \rightsquigarrow \epsilon, \overline{\psi_2}, \ldots, \overline{\psi_n}, \overline{\psi_r}}{\forall \overline{a}.\, \mu \ \leqslant^{\mathsf{s}}_{\overline{\omega}} \ \sigma_1, \ldots, \sigma_n; \eta \rightsquigarrow \overline{\theta(a)}, \overline{\psi_2}, \ldots, \overline{\psi_n}, \overline{\psi_r}} \ \textsc{InstPoly}$$

$$\boxed{\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow e_{\mathsf{F}}}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{fun}} x : \sigma \rightsquigarrow x} \ \textsc{VarHead}$$

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow e_{\mathsf{F}}}{\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow e_{\mathsf{F}}} \ \textsc{ExprHead}$$

$$\boxed{\Gamma \vdash^{\mathsf{arg}}_{\omega} e : \sigma \rightsquigarrow e_{\mathsf{F}}}$$

$$\frac{\Gamma \vdash e : \forall \overline{a}.\, \mu \rightsquigarrow e_{\mathsf{F}} \qquad \overline{b} \notin \Gamma}{\Gamma \vdash^{\mathsf{arg}}_{\bullet} e : \forall \overline{b}.\, [\overline{a \mapsto \tau}]\mu \rightsquigarrow \Lambda \overline{b}.\, e_{\mathsf{F}}\, \overline{\tau}} \ \textsc{ArgGen}$$

$$\frac{x : \forall \overline{p}.\, \tau \in \Gamma \qquad \overline{b} \notin \Gamma}{\Gamma \vdash^{\mathsf{arg}}_{\star} x : \forall \overline{b}.\, [\overline{a \mapsto \sigma}]\tau \rightsquigarrow \Lambda \overline{b}.\, x\, \overline{\sigma}} \ \textsc{VarGen}$$

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow e_{\mathsf{F}}}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma \rightsquigarrow e_{\mathsf{F}}}{\Gamma \vdash \lambda x.\, e : \tau \rightarrow \sigma \rightsquigarrow \lambda(x :: \tau).\, e_{\mathsf{F}}} \ \textsc{Abs}$$

$$\frac{\Gamma, x : \phi \vdash e : \sigma \rightsquigarrow e_{\mathsf{F}}}{\Gamma \vdash \lambda(x :: \phi).\, e : \phi \rightarrow \sigma \rightsquigarrow \lambda(x :: \sigma).\, e_{\mathsf{F}}} \ \textsc{AnnAbs}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{fun}} e_0 : \phi \rightsquigarrow e_{0,\mathsf{F}} \qquad \phi \ \leqslant^{\mathfrak{m}}_{\omega_1, \ldots, \omega_n} \ \sigma_1, \ldots, \sigma_n; \mu \rightsquigarrow \overline{\psi_1}, \ldots, \overline{\psi_n}, \overline{\psi_r} \\ \Gamma \vdash^{\mathsf{arg}}_{\omega_1} e_1 : \sigma_1 \rightsquigarrow e_{\mathsf{F},1} \qquad \ldots \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_n} e_n : \sigma_n \rightsquigarrow e_{\mathsf{F},n}\end{array}}{\Gamma \vdash e_0\, e_1\, \ldots\, e_n : \mu \rightsquigarrow e_{0,\mathsf{F}}\, \overline{\psi_1}\, e_{\mathsf{F},1}\, \ldots\, \overline{\psi_n}\, e_{\mathsf{F},n}\, \overline{\psi_r}} \ \textsc{App}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{fun}} e_0 : \phi \rightsquigarrow e_{0,\mathsf{F}} \qquad \phi \ \leqslant^{\mathfrak{u}}_{\omega_1, \ldots, \omega_n} \ \sigma_1, \ldots, \sigma_n; \eta \rightsquigarrow \overline{\psi_1}, \ldots, \overline{\psi_n}, \overline{\psi_r} \\ \Gamma \vdash^{\mathsf{arg}}_{\omega_1} e_1 : \sigma_1 \rightsquigarrow e_{\mathsf{F},1} \qquad \ldots \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_n} e_n : \sigma_n \rightsquigarrow e_{\mathsf{F},n}\end{array}}{\Gamma \vdash (e_0\, e_1\, \ldots\, e_n :: \forall \overline{b}.\, \eta) : \forall \overline{b}.\, \eta \rightsquigarrow \Lambda \overline{b}.\, h_{\mathsf{F}}\, \overline{\psi_1}\, e_{\mathsf{F},1}\, \ldots\, \overline{\psi_n}\, e_{\mathsf{F},n}\, \overline{\psi_r}} \ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_1 : \phi \rightsquigarrow e_{\mathsf{F},1} \qquad \Gamma, x : \phi \vdash e_2 : \sigma \rightsquigarrow e_{\mathsf{F},2}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma \rightsquigarrow (\lambda(x :: \phi).\, e_{\mathsf{F},2})\, e_{\mathsf{F},1}} \ \textsc{Let}$$

Figure 16. Translation to System F

Proof of Theorem 3.2. We distinguish seven cases:

- *Case* Abs. The derivations look like:

$$\frac{\theta_1\Gamma, x : \tau_1 \vdash e : \sigma_1}{\theta_1\Gamma \vdash \lambda x.e : \tau_1 \to \sigma_1} \qquad \frac{\theta_2\Gamma, x : \tau_2 \vdash e : \sigma_2}{\theta_2\Gamma \vdash \lambda x.e : \tau_2 \to \sigma_2}$$

  We cannot apply the induction hypothesis yet, since the environments in the premises are not of the right shape. Consider instead the environment $\Gamma' = \Gamma, x : \alpha$ for a fresh α, and the substitutions

$$\theta_1' = [\alpha \mapsto \tau_1] \circ \theta_1 \qquad \theta_2' = [\alpha \mapsto \tau_2] \circ \theta_2$$

  These substitutions are fully monomorphic, since all of θ_i and τ_i ($1 \leqslant i \leqslant 2$) are fully monomorphic by hypothesis. We have then the following equalities over the environments

$$\theta_1\Gamma, x : \tau_1 = \theta_1'\Gamma' \qquad \theta_2\Gamma, x : \tau_2 = \theta_2'\Gamma'$$

  and thus we can apply the induction hypothesis to e to obtain σ^*, φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$. Consider now the extended substitutions:

$$\varphi_1' = [\alpha \mapsto \tau_1] \circ \varphi \qquad \varphi_2' = [\alpha \mapsto \tau_2] \circ \varphi_2$$

  Since τ_1 and τ_2 are fully monomorphic types, φ_i' are fully monomorphic substitutions. Take $\sigma' = \alpha \to \sigma^*$. We have, for each derivation, that

$$\varphi_i'\sigma' = \varphi_i'(\alpha \to \sigma^*) = \varphi_i'\alpha \to \varphi_i'\sigma^* = \tau_i \to \sigma_i$$

  And we are done: σ', φ_1' and φ_2' are the desired outputs of the theorem.
- *Case* AnnAbs. Similar to Abs.
- *Case* App. In this case the derivations have the following shape for each $i \in \{1, 2\}$.

$$\frac{\theta_i\Gamma \vdash^{\mathsf{fun}} e_0 : \theta_i\phi \qquad \phi \leqslant^{\mathrm{m}}_{\overline{\omega}} \sigma_{i,1}, \ldots, \sigma_{i,n}; \mu_i \qquad \overline{\theta_i\Gamma \vdash^{\mathsf{arg}}_{\omega_j} e_j : \sigma_{i,j}}}{\theta_i\Gamma \vdash e_0\, e_1\, \ldots\, e_n : \mu_i}$$

  We have two choices: either ϕ is a top-level monomorphic type η, or $\theta_i\phi = \forall\overline{a}.\, \theta_i\eta$. This shall remain true during the derivation of $\leqslant^{\mathrm{m}}_{\overline{\omega}}$. Intuitively, the substitutions do not alter the "polymorphism structure" of the type. In particular, as result type of the instantiation, we get μ_1 and μ_2 for which we know that there exists a common μ^* such that:

$$\mu_1 = \varphi_1^m\varphi_1^r\theta_1\mu^* \qquad \mu_2 = \varphi_2^m\varphi_2^r\theta_2\mu^*$$

  where φ_i^m correspond to the fully monomorphic substitutions applied to those type variables which do not appear in any of the arguments. But we should be careful here: φ_i^r are unrestricted, and for those we do not directly obtain the conclusion of the theorem.

  Let us first consider the case in which $\omega_j = \bullet$ for all j. In other words, all arguments are typed using the ArgGen rule for $\vdash^{\mathsf{arg}}$. By the definition of the respects relation for substitutions, for each quantified type variable a in ϕ we have *at least one* $\sigma_{i,j}$ for which $\varphi_i^r\theta_i\sigma_{i,j}$ is of the form $\forall\overline{a}.\,\mathsf{T}\,\overline{\psi}$ for both derivations. We can see this by distinguishing between unrestricted variables , for which we already have the type constructor in ϕ; and top-level monomorphic variables for which the corresponding substitution must map to top-level monomorphic types.

  Take the expression corresponding to this type variable a – let us call it e_a – and the derivations $\theta_1\Gamma \vdash e_a : \varphi_1^r\theta_1\sigma_{1,a}$ and $\theta_2\Gamma \vdash e_a : \varphi_2^r\theta_2\sigma_{2,a}$. By the induction hypothesis there exists a σ_j^* and fully monomorphic substitutions $\xi_{i,a}$ such that $\varphi_i^r\theta_i\sigma_{i,a} = \xi_{i,a}\sigma_a^*$.

  We know that for each type variable a appearing in some argument of the application we have at least one ξ_a so that a is in its domain. It does not matter which one we choose: they all have to agree or otherwise the derivation is ill-formed. Take now the following substitutions:

$$\xi_i = \bigcirc_{a \in \mathsf{args}^{\mathsf{n}}(\phi)}[a \mapsto \xi_{i,a}(a)]\,,$$

  where $\bigcirc$ denotes substitution composition. By construction, we know that $\varphi_i^r = \xi_i$: they have the same domain and they map to the same outputs. But now we are sure that ξ_i are fully monomorphic, since arise from a composition of fully monomorphic substitutions. In conclusion, we have that

$$\mu_1 = \varphi_1^m\xi_1\theta_1\mu^* \qquad \mu_2 = \varphi_2^m\xi_2\theta_2\mu^*$$

  and all of φ_i^m, ξ_i and θ_i are fully monomorphic.

  When we use VarGen we need to work a bit more. If for each type variable there is one e_a where ArgGen has been applied, the proof still works. On the other hand, if for a type variable a all expressions where it appears make use of VarGen, we cannot guarantee the existence of the σ_j^* and fully monomorphic substitutions $\xi_{i,a}$ by induction hypothesis. However, in that case the definition of $\rhd^{\mathrm{s}}_{\overline{\omega}}$ ensures that the substitution for the type variable a is fully monomorphic, giving us the desired result.
- *Case* AnnApp. This case is trivial, since the annotation ensures that both derivations yield the same type.
- *Case* Let. In this case the derivations are:

$$\frac{\theta_i\Gamma \vdash e_1 : \phi_i \qquad \theta_i\Gamma, x : \phi_i \vdash e_2 : \sigma_i}{\theta_i\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_i} \text{ for } i \in \{1, 2\}$$

  We cannot readily apply the induction hypothesis to the second premise – which would give us the desired conclusion –, since the environment may not be in the right shape. By the induction hypothesis on the first premise, there exists a type ϕ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\phi_i = \varphi_i\phi^*$. We can build a new version of ϕ^*, ϕ', where all the free variables are fresh – and thus disjoint from those in Γ – and corresponding φ_i' where the domain is replaced by these free variables. Consider $\theta'_i = \theta_i \circ \varphi'_i$:

$$\begin{array}{rcl}\theta'_i(\Gamma, x : \phi') & = & \theta'_i(\Gamma), x : \theta'_i\phi' \\ & = & \text{(substitutions have disjoint domains)} \\ & & \theta_i\Gamma, x : \varphi'_i\phi' \\ & = & \text{(by definition of } \varphi'_i) \\ & & \theta_i\Gamma, x : \phi_i\end{array}$$

This means that we have found a fully monomorphic substitution θ'_i for the environment, which allows us to apply induction on the second premise and reach the desired conclusion.

- *Case* Case. By induction on the first premise of Case, we get σ_0^* and two substitutions φ_1 and φ_2 such that $\sigma_{0,i} = \varphi_i\sigma_0^*$ for each of the two derivations. Without loss of generality, we can assume that $\sigma_0^* = \forall\overline{a}.\,\mu_0^*$. Then $\sigma_{0,i} = \forall\overline{a}.\,\varphi_i\mu_0^*$.
  The next step is to notice that the instantiation judgment is nullary. As a result, all the type variables in $\overline{a}$ *must* be substituted by fully monomorphic types. Let π_i be the substitution for each of the derivations. Then we know that $\mathsf{T}\,\overline{\phi_{0,i}} = \pi_i\varphi_i\mu_0^* = \mathsf{T}\,\overline{\pi_i\varphi_i\phi_0^*}$. Playing the same game we did for Let, we can build substitutions θ'_i such that,

$$\theta'_i(\Gamma, \overline{x_i : [\overline{a \mapsto \phi_{0,i}}]\sigma_j}) = \theta_i\Gamma, \overline{x_i : [\overline{a \mapsto \pi_i\varphi_i\phi_0^*}]\sigma_j}$$

  The last step is applying induction hypothesis to one of the branches and obtain the desired common polymorphic type and fully monomorphic instantiations. The branch we choose does not matter, since they all give the same type as result.

□

Theorem 3.4 (Substitution). *If* $\Gamma \vdash u : \sigma$ *and* $\Gamma, x : \sigma \vdash e[x] : \phi$*, then* $\Gamma \vdash e[u] : \phi$*.*

Proof. By induction over the expression $e[x]$. The only interesting case is when x is the head of an application, that is, $e = x\,e_1\,\ldots\,e_n$ and u is also an application, $u = u_0\,u_1\,\ldots\,u_n$. Note that in that case the assigned types are always top-level monomorphic, so $\sigma = \mu$ and $\phi = \eta$.

The derivations involved look like:

$$\frac{\Gamma \vdash^{\mathsf{fun}} u_0 : \sigma_u \qquad \sigma_u \leqslant^{\mathfrak{m}}_{\omega_1,\ldots,\omega_m} \sigma_1,\ldots,\sigma_m;\mu \qquad \overline{\Gamma \vdash^{\mathsf{arg}}_{\omega_i} u_i : \sigma_i}}{\Gamma \vdash u_0\,u_1\,\ldots\,u_m : \mu}$$

$$\frac{\Gamma \vdash^{\mathsf{fun}} x : \mu \qquad \mu \leqslant^{\mathfrak{m}}_{\omega'_1,\ldots,\omega'_n} \phi_1,\ldots,\phi_n;\eta \qquad \overline{\Gamma \vdash^{\mathsf{arg}}_{\omega'_j} e_j : \phi_j}}{\Gamma \vdash x\,e_1\,\ldots\,e_n : \eta}$$

and the question is whether we can derive:

$$\frac{\Gamma \vdash^{\mathsf{fun}} u_0 : \sigma_u \qquad \sigma_u \leqslant^{\mathfrak{m}}_{\omega_1,\ldots,\omega_m,\omega'_1,\ldots,\omega'_n} \sigma_1,\ldots,\sigma_m,\phi_1,\ldots,\phi_n;\eta \qquad \overline{\Gamma \vdash^{\mathsf{arg}}_{\omega_i} u_i : \sigma_i} \qquad \overline{\Gamma \vdash^{\mathsf{arg}}_{\omega'_j} e_j : \phi_j}}{\Gamma \vdash u_0\,u_1\,\ldots\,u_m\,e_1\,\ldots\,e_n : \eta}$$

All the premises on this last rule come directly from those in the hypotheses, except for the instantiation

$$\sigma_u \leqslant^{\mathfrak{m}}_{\omega} \sigma_1,\ldots,\sigma_m,\phi_1,\ldots,\phi_n;\eta$$

For the n last components we can reuse the derivation in the second hypothesis. For the first m components, inspection on the rules of guardedness show that any instantiation with m arguments is admissible when $m + n$ are considered. □

Theorem 3.5. *Let* $app :: \forall a\ b.\,(a \to b) \to a \to b$ *and* $revapp :: \forall a\ b.\,a \to (a \to b) \to b$ *be the application and reverse application functions, respectively. Given two expressions* f *and* e *such that* $\Gamma \vdash^{\mathsf{fun}} f : \sigma_0$*, and* $\sigma_0 \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon;\sigma_1 \to \phi$ *then:*

$$\Gamma \vdash f\ e : \phi \iff \Gamma \vdash app\ f\ e : \phi \iff \Gamma \vdash revapp\ e\ f : \phi$$

Proof. Let us compare the derivations of the first two.

$$\frac{\Gamma \vdash^{\mathsf{fun}} f : \sigma_0 \qquad \sigma_0 \leqslant^{\mathfrak{m}}_{\omega_e} \sigma_1;\mu \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_e} e : \sigma_1}{\Gamma \vdash f\ e : \mu}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{fun}} app : \forall a\ b.\,(a \to b) \to a \to b \\ \Gamma \vdash^{\mathsf{arg}}_{\omega_f} f : \sigma_1 \to \mu \qquad \Gamma \vdash^{\mathsf{arg}}_{\omega_e} e : \sigma_1 \\ \forall a\ b.\,(a \to b) \to a \to b \leqslant^{\mathfrak{m}}_{\omega_f,\omega_e} (\sigma_1 \to \mu),\sigma_1;\mu\end{array}}{\Gamma \vdash f\ e : \mu}$$

The application of *app* can be instantiated with any type, given that both variables a and b are guarded. We need to consider the two different ways in which $\Gamma \vdash^{\mathsf{arg}}_{\omega_f} f : \sigma_1 \to \mu$ can be derived. If it is derived using ArgGen, then we have:

$$\frac{\Gamma \vdash f : \sigma_0 \qquad \sigma_0 \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon;\sigma_1 \to \phi}{\Gamma \vdash^{\mathsf{arg}}_{\bullet} f : \sigma_1 \to \phi}$$

Then the premises for the whole derivation are the same, except for the highlighted one. We know that if $\sigma_0 \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon;\sigma_1 \to \phi$, then $\sigma_0 \leqslant^{\mathfrak{m}}_{\bullet} \sigma_1;\phi$, by inspection of the rule relating instantiation and function types.

The other possibility is that f is a variable and the rule for single variables, VarGen, applies:

$$\frac{f : \forall\overline{p}.\,\tau \in \Gamma \qquad \forall\overline{p}.\,\tau \leqslant^{\mathfrak{u}}_{\epsilon} \epsilon;\sigma_1 \to \phi}{\Gamma \vdash^{\mathsf{arg}}_{\star} f : \sigma_1 \to \phi}$$

In this case VarGen does not rule our polymorphic instantiation. However, the $\leqslant$ judgment applied to *app* will ignore this argument for guardedness purposed. That means that we can only instantiate impredicative those type variables coming for e, exactly the same as we could with $f\ e$.

The proof for *revapp* is similar to the one for *app*. □

D.2 Constraint-based formulation

Theorem 4.1. *Suppose $\Gamma \vdash e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a new set C' without instantiation and generalisation constraints which fixes the value of all unrestricted and top-level monomorphic variables.*

Lemma D.2. *Suppose $\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a new set C' without instantiation or generalisation constraints which fixes the value of all unrestricted and top-level monomorphic variables.*

Proof. We prove both theorems by mutual induction. Note that Lemma D.2 follows simply from Theorem 4.1, as the head typing judgment either produces no constraints with the rule VarHead or refers to the normal typing judgment as a premise with the rule ExprHead.

For Theorem 4.1, we proceed by induction on the constraint generation judgment $\Gamma \vdash e : \sigma \rightsquigarrow C$.

Case Abs ***and*** AnnAbs**.** Follows directly from induction hypothesis, since the constraints from the body are copied as the output.

Case App**.** We need to consider three sets of constraints: the set C coming from generating constraints for the head, the set of instantiation constraints $C_{\leqslant}$ which extract the expected types of arguments from the function types, and the set of generalisation constraints $C_{\leq}$ which check that the actual arguments fit into those expected types. The third set is obtained from the ancillary judgement $\vdash^{\mathsf{arg}}$.

The desired properties for the first set of constraints, C, follow by induction. Since C fixes all the unrestricted and top-level monomorphic variables, this implies that ϕ is fixed up to fully monomorphic variables. As a result, we can rewrite all the constraints in $C_{\leqslant}$ into equalities – but note that instantiations may introduce new variables to be fixed.

First consider the case in which all applications of $\vdash^{\mathsf{arg}}$ use the ArgGen rule. In that case, the newly-introduced variables can be divided between unrestricted $\overline{u}$, top-level monomorphic $\overline{t}$, and fully monomorphic $\overline{m}$. Only the first two interact with arguments, by definition. For the first set we know that for each variable α there is at least one argument of the form $\forall \overline{a}. \mathsf{T}\,\overline{\phi}$, where $\alpha \in \mathsf{ftv}(\overline{\phi})$. For that argument there is a corresponding expression e_j and a generalisation constraint,

$$(\forall \{\overline{\gamma_j}\}. C_j \Rightarrow \sigma_j) \leq (\forall \overline{a}. \mathsf{T}\,\overline{\phi})$$

Rule inst∀r and inst∀l apply, rewriting this constraint to $C_j \wedge \sigma_j \leqslant^{\mathsf{m}}_{\epsilon} \epsilon; \mathsf{T}\,\overline{\phi}$, possibly under an universal quantifier $\forall \overline{b}$. Now we can apply the induction hypothesis to C_j, which means that we can rewrite these constraints to a set of type equalities which fixes the unrestricted and top-level monomorphic variables.

We still have a remaining constraint $\sigma_j \leqslant^{\mathsf{m}}_{\epsilon} \epsilon; \mathsf{T}\,\overline{\phi}$ and we have not proven yet that the variable $\alpha \in \mathsf{ftv}(\overline{\phi})$ is fixed. But since σ_j is fixed up to fully monomorphic components, we can decide which instantiation rule to apply; in either case a type equality is produced. In turn, this equality fixes the value for α. As a result, the generalisation constraint can be completely turned into equalities, as desired. Furthermore, this value of α is floated out of the universal quantification – if it was ever introduced – up to the point where the variable was introduced.

Once the value of α is fixed — because of its appearance under a type constructor —, we can deal with those cases in which the variable appears alone as an argument,

$$(\forall \{\overline{\gamma_k}\}. C_k \Rightarrow \sigma_k) \leq \alpha$$

But now we already know the type for α up to its fully monomorphic components! Thus, we are able to decide which rule in the solver to apply, and then apply the induction hypothesis. The end result is again a set of equalities, maybe under a universal quantifier.

Arguments which feature a top-level monomorphic variable $\beta \in \overline{t}$ are dealt with in the same way; the fact that the variable is top-level monomorphic is enough to unwrap the generalisation constraint. Then, the solving proceeds as with α unrestricted.

Finally, no constraints are generated over the set of fully monomorphic types $\overline{m}$. This is OK, since these variables are already fixed up to fully monomorphic components, by definition. This case also applies when all the expressions featuring a variable α result from the application of VarGen.

Case AnnApp**.** This case is almost identical to App. The only difference is that now there is a set of variables $\overline{u}'$ which would have been classified as either top-level or fully monomorphic which are now classified as unrestricted.

As in the case of App, we apply the induction hypothesis over the head to guarantee that the set of constraints C can be turned into a set of equalities under a mixed prefix. Since these equalities fix σ up to fully monomorphic components, we can still apply the same reasoning to the set of instantiation constraints $C_{\leqslant}$. In particular, the constraint $\beta_n \leqslant^{\mathsf{u}}_{\epsilon} \epsilon; \eta$ is turned into an equality, too, and fixes the value of all those $\overline{u}'$ variables – those are the ones appearing in the assignment to β_n and η is completely known since it is explicitly given by the programmer.

For the remaining variables we apply the same reasoning as before, distinguishing between unrestricted and top-level monomorphic variables and going through each of the arguments. □

D.2.1 Solved form and solutions

In Section 4.4 we introduced the notion of a *solved form*. In the following results we also need the notiong of when a solved form C_s is a *solution* for a set of constraints C. We give the corresponding judgment $C_s \models C$ in Figure 17.

$$\boxed{C_\mathrm{s} \models C} \; \boxed{C_\mathrm{s} \models' C}$$

$$\frac{C_\mathrm{s} \models' \widehat{C_\mathrm{s}}(C)}{C_\mathrm{s} \models C}$$

$$\frac{}{C_\mathrm{s} \models' \sigma \sim \sigma} \qquad \frac{C_\mathrm{s} \models' C_1 \qquad C_\mathrm{s} \models' C_2}{C_\mathrm{s} \models' C_1 \wedge C_2} \qquad \frac{\forall \overline{b}. C'_\mathrm{s} \in C_\mathrm{s} \qquad C'_\mathrm{s} \models C}{C_\mathrm{s} \models' \forall \overline{b}. C}$$

$$\frac{C_\mathrm{s} \models \mu \sim \eta}{C_\mathrm{s} \models' \leqslant^{\mathfrak{s}}_{\epsilon} \epsilon; \eta} \qquad \frac{C_\mathrm{s} \models \mu \sim \sigma \to \sigma' \qquad C_\mathrm{s} \models \sigma' \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta}{C_\mathrm{s} \models' \mu \leqslant^{\mathfrak{s}}_{\omega, \overline{\omega}} \sigma, \overline{\phi}; \eta} \qquad \frac{\overline{a} \text{ were instantiated to } \overline{\alpha} \qquad C_\mathrm{s} \models [\overline{a \mapsto \alpha}]\mu \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta}{C_\mathrm{s} \models' \forall \overline{a}. \mu \leqslant^{\mathfrak{s}}_{\overline{\omega}} \overline{\phi}; \eta}$$

$$\frac{C_\mathrm{s} \models C \qquad C_\mathrm{s} \models \sigma \leqslant^{\mathfrak{m}}_{\epsilon} \epsilon; \eta}{C_\mathrm{s} \models' (\forall \{\overline{\alpha}\}. C \Rightarrow \sigma) \preceq \eta} \qquad \frac{C_\mathrm{s} \models' \forall \overline{b}. (\forall \{\overline{\alpha}\}. C \Rightarrow \sigma \preceq \eta)}{C_\mathrm{s} \models' (\forall \{\overline{\alpha}\}. C \Rightarrow \sigma) \preceq (\forall \overline{b}. \eta)}$$

Figure 17. Definition of solution

Given two solutions C_s and D_s which range *over the same set of variables*, we say that C_s is *more general* than D_s if there exists a fully monomorphic substitution π such that for every pair of assignments $\alpha \sim \sigma$ in C_s, and $\alpha \sim \phi$ in D_s, $\phi = \pi\sigma$.

The solver described in [15] returns most general solutions for a given set of equalities under a mixed prefix. This is a consequence of the fact that they reuse the first-order solver in [11], which always returns most general substitutions.

D.2.2 Soundness and principality

In the proofs of the following results we focus on the core language without **let**, **do** and pattern matching. Nevertheless, the results still hold when the language is extended: no surprises there.

Lemma D.3. *Suppose that C_s is a solution for the constraint:*

$$\sigma \leqslant^{\mathfrak{s}}_{\overline{\omega}} \alpha_1^{\mathfrak{u}}, \ldots, \alpha_n^{\mathfrak{u}}; \delta^{\mathfrak{t}}$$

Then we have a derivation for:

$$\widehat{C_\mathrm{s}}(\sigma) \leqslant^{\mathfrak{s}}_{\overline{\omega}} \widehat{C_\mathrm{s}}(\alpha_1^{\mathfrak{u}}), \ldots, \widehat{C_\mathrm{s}}(\alpha_n^{\mathfrak{u}}); \widehat{C_\mathrm{s}}(\delta)$$

(as defined in the declarative specification)

Proof. The definition for $\models$ correspond one to one to the rules for $\leqslant$ in the declarative specification. □

Theorem D.4 (Soundness, solution version)**.** *Let Γ be an environment and e an expression. Then, for every set of constraints C',*

1. *If $\Gamma \vdash^{\mathsf{fun}} e : \sigma \rightsquigarrow C$, and $C_\mathrm{s} \models C \wedge C'$, then we can build a derivation for $\widehat{C_\mathrm{s}}(\Gamma) \vdash^{\mathsf{fun}} e : \widehat{C_\mathrm{s}}(\sigma)$.*
2. *If $\Gamma \vdash e : \sigma \rightsquigarrow C$ and $C_\mathrm{s} \models C \wedge C'$, then we can build a derivation for $\widehat{C_\mathrm{s}}(\Gamma) \vdash e : \widehat{C_\mathrm{s}}(\sigma)$.*
3. *If $\Gamma\, ; \overline{v} \vdash^{\mathsf{arg}}_{\omega} e : \sigma \rightsquigarrow C$ and $C_\mathrm{s} \models C \wedge C'$, then we have a derivation for $\widehat{C_\mathrm{s}}(\Gamma) \vdash^{\mathsf{arg}}_{\omega} e : \widehat{C_\mathrm{s}}(\phi)$.*

Proof. We prove this theorem by mutual induction over the expression e.

Proof of (1). We distinguish two cases by inversion:

- *Case* VarHead. We have in this case that $\Gamma \vdash^{\mathsf{fun}} x : \sigma \rightsquigarrow \epsilon$ if $x : \sigma \in \Gamma$. In this case C_s is a solution of C'. By applying the substitution $\widehat{C_\mathrm{s}}$ to the environment we get that $x : \widehat{C_\mathrm{s}}(\sigma) \in \widehat{C_\mathrm{s}}(\Gamma)$, which allows us to conclude that $\widehat{C_\mathrm{s}}(\Gamma) \vdash^{\mathsf{fun}} x : \widehat{C_\mathrm{s}}(\sigma)$.
- *Case* ExprHead. Follows directly from the induction hypothesis and (2).

Proof of (2). We distinguish four cases by inversion:

- *Case* Abs. In this case the constraint gathering is:

$$\frac{\Gamma, x : \alpha^{\mathfrak{m}} \vdash e : \mu \rightsquigarrow C}{\Gamma \vdash \lambda x.e : \alpha^{\mathfrak{m}} \to \mu \rightsquigarrow C}$$

  By induction hypothesis we know that for every C', if $C \wedge C'$ is solved to C_s, then we have $\widehat{C_\mathrm{s}}(\Gamma), x : \widehat{C_\mathrm{s}}(\alpha^{\mathfrak{m}}) \vdash e : \widehat{C_\mathrm{s}}(\sigma)$. From this we can apply the corresponding rule from the declarative specification.

$$\frac{\widehat{C_\mathrm{s}}(\Gamma), x : \widehat{C_\mathrm{s}}(\alpha) \vdash e : \widehat{C_\mathrm{s}}(\sigma)}{\widehat{C_\mathrm{s}}(\Gamma) \vdash \lambda x.e : \widehat{C_\mathrm{s}}(\alpha) \to \widehat{C_\mathrm{s}}(\sigma)} \;\textsc{App}$$

  Now we just need to notice that $\widehat{C_\mathrm{s}}(\alpha) \to \widehat{C_\mathrm{s}}(\sigma) \sim \widehat{C_\mathrm{s}}(\alpha \to \sigma)$ and we are done.
- *Case* AnnAbs. Similar to the Abs case.
- *Case* App. Suppose that the shape of the expression is $e_0\, e_1\, \ldots\, e_n$. By applying (1) on e_0 we know that a solution C_s determines a derivation $\widehat{C_\mathrm{s}}(\Gamma) \vdash^{\mathsf{fun}} e_0 : \widehat{C_\mathrm{s}}(\sigma)$. By Lemma D.3, a solution for the $\leqslant$ constraint determines a derivation for $\widehat{C_\mathrm{s}}(\sigma) \leqslant^{\mathfrak{m}}_{\ldots} \widehat{C_\mathrm{s}}(\alpha_1^{\mathfrak{u}}), \ldots, \widehat{C_\mathrm{s}}(\alpha_n^{\mathfrak{u}}); \widehat{C_\mathrm{s}}(\delta^{\mathfrak{t}})$. Finally, by applying (3) to each argument, we get the corresponding derivations for $\widehat{C_\mathrm{s}}(\Gamma) \vdash^{\mathsf{arg}}_{b_i} e_i : \widehat{C_\mathrm{s}}(\alpha_i^{\mathfrak{u}})$. As a result, we can apply the App rule from the declarative specification.
- *Case* AnnApp. Similar to the App case.

Proof of (3). We distinguish two cases by inversion:

- *Case* ArgGen. We can assume, without loss of generality, that the generalisation constraint has the form $(\forall\{\overline{v}\}. C \Rightarrow \sigma) \le (\forall\overline{b}. \eta)$ for a possibly empty set of variables $\overline{b}$.
  Let us first consider the simple case in which the set of quantified variables $\overline{b}$ is empty. By the definition of solution, this implies that we have
  $$C_s \models C \quad \text{and} \quad C_s \models \sigma \leqslant^{\mathrm{m}}_{\epsilon} \epsilon; \eta$$
  Now we can apply (2) to $C_s \models C$ and derive that $\widehat{C_s}(\Gamma) \vdash \widehat{C_s}(\sigma)$. By inspection of the rules, we know that a solution of the second constraint, $\sigma \leqslant^{\mathrm{m}}_{\epsilon} \epsilon; \eta$, defines a fully monomorphic substitution in the declarative specification. As a result, we can apply rule ArgGen and obtain the desired result. The case for a not empty set of constraints is similar, one just has to realize that the rule for $\models'$ for generalization constraints introduce the Skolem variables required by ArgGen.
- *Case* VarGen. In this case the generated constraint is $[\overline{p \mapsto \alpha^{\mathrm{u}}}]\tau \le (\overline{b}. \eta)$. As in the last case of ArgGen, the definition of solution tells us that there is a $\forall\overline{b}. C'_s \in C_s$ such that
  $$C'_s \models C_s(C) \quad \text{and} \quad C'_s \models C_s([\overline{p \mapsto \alpha^{\mathrm{u}}}]\tau) \leqslant^{\mathrm{u}}_{\epsilon} \epsilon; C_s(\eta)$$
  The second application of $\models'$ is trivially equal to $C'_s \models C_s([\overline{p \mapsto \alpha^{\mathrm{u}}}]\tau) \sim C_s(\eta)$, since the type in the left-hand side is by construction a top-level monomorphic one. This is enough to apply rule VarGen from the declarative specification.

□

Theorem 4.2 (Soundness)**.** *Let Γ be a closed environment and e an expression. If $\Gamma \vdash e : \sigma \rightsquigarrow C$ and C_s is a solution for C with an induced substitution $\widehat{C_s}$, then we have $\Gamma \vdash e : \widehat{C_s}(\sigma)$.*

Proof. By Theorem D.4 we know that the existence of such a solution C_s implies that $\widehat{C_s}(\Gamma) \vdash e : \widehat{C_s}(\sigma)$. Since Γ is closed, $\widehat{C_s}(\Gamma) = \Gamma$, as desired. □

Theorem D.5 (Derivations provide solutions)**.** *Let Γ be an environment and e an expression.*

1. *If $\theta\Gamma \vdash^{\mathsf{fun}} e : \sigma$ and $\Gamma \vdash^{\mathsf{fun}} e : \phi \rightsquigarrow C$, then there exists a solution C_s for C such that $\widehat{C_s}$ is fully monomorphic and $\widehat{C_s}(\theta\phi) \sim \sigma$.*
2. *If $\theta\Gamma \vdash e : \sigma$ and $\Gamma \vdash e : \phi \rightsquigarrow C$, then there exists a solution C_s for C such that $\widehat{C_s}$ is fully monomorphic and $\widehat{C_s}(\theta\phi) \sim \sigma$.*

Proof. We prove this theorem by mutual induction over the expression e.

Proof of (1). We distinguish two cases by inversion:

- *Case* VarHead. In this case the generated set of constraints C it empty, so we can just take an empty solved form as solution C_s. The corresponding substitution $\widehat{C_s}$ is the identity function, which is trivially fully monomorphic.
  Now, we just need to prove that $\widehat{C_s}(\theta\phi) \sim \theta\phi \sim \sigma$. By inversion of the rule VarHead in the constraint-based formulation we know that ϕ must come from an element $x : \phi \in \Gamma$. On the other hand, we know that in the declarative specification σ must come from $x : \sigma \in \theta\Gamma$. Since Γ is the same in both derivations, it must be the case that $\sigma \sim \theta\phi$.
- *Case* ExprHead. Follows by induction on the premise.

Proof of (2). We distinguish four cases:

- *Case* Abs. The derivation in the declarative specification looks like:
  $$\frac{\theta\Gamma, x : \tau \vdash e : \sigma}{\theta\Gamma \vdash \lambda x.e : \tau \to \sigma}$$
  Let us first rewrite the premise as $\theta'(\Gamma, x : \alpha)$, where $\theta' = [a \mapsto \tau] \circ \theta$. Now we can apply the induction hypothesis to obtain a solution C'_s such that $\widehat{C_s}'(\theta'\phi) \sim \sigma$, where ϕ is the type assigned to the abstraction body during constraint gathering.
  Now consider the problem $C'_s \wedge \alpha \sim \tau$. We know that a solution C_s exists, or otherwise the expression would be ill-typed. Furthermore, $\widehat{C_s}' \circ \theta' = \widehat{C_s} \circ \theta$ – we are just moving the α assignment from one place to the other. In conclusion, we have that:
  $$C_s(\theta(\alpha \to \phi)) \sim C'_s(\theta'(\alpha \to \phi)) \sim \tau \to \sigma$$
- *Case* AnnAbs. Similar to the Abs case.
- *Case* App. The derivation in the specification is:
  $$\frac{\begin{array}{c}\theta\Gamma \vdash^{\mathsf{fun}} e_0 : \sigma_0 \\ \sigma_0 \leqslant^{\mathrm{m}}_{\overline{\omega}} \sigma_1, \dots, \sigma_n; \mu \quad \theta\Gamma \vdash^{\mathsf{arg}}_{\omega_i} e_i : \sigma_i\end{array}}{\theta\Gamma \vdash e_0\, e_1\, \dots\, e_n : \phi}\ \textsc{App}$$
  The constraints generated by this sequence of App rules are:
  $$C_0 \wedge (\phi_0 \leqslant^{\mathrm{m}}_{\overline{\omega}} \alpha^{\mathrm{u}}_1, \dots, \alpha^{\mathrm{u}}_n; \delta^{\mathrm{t}}) \wedge C_1 \wedge \dots \wedge C_n$$
  By (1) applied to e_0 there exists a solution C^0_s such that $\widehat{C_s}^0\theta\phi_0 \sim \sigma_0$. From the derivation of $\sigma_0 \leqslant^{\mathrm{m}}_{\overline{\omega}} \sigma_1, \dots, \sigma_n; \mu$ in the declarative specification we obtain a list of equalities C^{inst}_s, in particular $\alpha^{\mathrm{u}}_i \sim \sigma_i$ and $\delta^{\mathrm{t}} \sim \mu$.
  Let us call $C'_s = C^{\mathrm{h}}_s \wedge C^{\mathsf{inst}}_s$ and consider for each argument the derivation $\widehat{C_s}'(\theta\Gamma) \vdash^{\mathsf{arg}}_{\omega_i} e_i : \widehat{C_s}'(\sigma_i)$; which holds by substitutivity in the typing judgment. In the case σ_i is polymorphic, the rules for $\vdash^{\mathsf{arg}}$ perform the job required to obtain a solution, namely introducing rigid variables. We are thus left with $\widehat{C_s}'(\theta\Gamma) \vdash e_i : \widehat{C_s}'(\eta_i)$, where we can apply induction hypothesis to

obtain $C_s^{\text{arg}_i}$. Define C_s^i to be exactly $C_s^{\text{arg}_i}$ if no rigid variables where introduced and $\forall\overline{b}.\, C_s^{\text{arg}_i}$ otherwise. The final solution is the conjunction:

$$C_s^{\text{h}} \wedge C_s^{\text{inst}} \wedge C_s^1 \wedge \ldots C_s^n$$

- *Case* AnnApp. Similar to the App case.

□

Corollary D.6. *Suppose $\Gamma \vdash e : \sigma$ and let $\Gamma \vdash e : \phi \rightsquigarrow C$. Then there exists a solution C_s for C such that $\widehat{C_s}$ is monomorphic and $\widehat{C_s}(\phi) \sim \sigma$.*

Corollary D.7. *Suppose $\Gamma \vdash e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a solved form.*

Proof. By Theorem 4.1 every consistent set of constraints can be turned into an instance of the problem of first-order unification under a mixed prefix. The solver in Pottier and Rémy [15] describes a complete algorithm for this problem, from which we obtain the desired solved form. □

Corollary D.8 (Principality, solution version)**.** *Let $\Gamma \vdash e : \sigma$ and let $\Gamma \vdash e : \phi \rightsquigarrow C$. If C is solved to C_s, then $\Gamma \vdash e : \widehat{C_s}(\phi)$ and there exists a monomorphic substitution π such that $\pi\widehat{C_s}(\phi) \sim \sigma$.*

Proof. By soundness we know that if C is solved to C_s, then $\Gamma \vdash e : C_s(\phi)$. From the derivation of $\Gamma \vdash e : \sigma$ we obtain another solution C_s' such that $C_s'(\phi) \sim \sigma$. Since the solver produces most general solutions we know in particular that there exists a fully monomorphic substitution π such that $\pi \circ \widehat{C_s} = \widehat{C_s}'$. This gives the desired result. □

Theorem 4.3 (Principality)**.** *Suppose $\Gamma \vdash e : \sigma$. Then there exists a type ϕ such that $\Gamma \vdash e : \phi$, and for every other $\Gamma \vdash e : \sigma'$, there is a fully monomorphic substitution π such that $\sigma' = \pi\phi$.*

Proof. Take $\Gamma \vdash e : \phi \rightsquigarrow C$. Given that the derivation $\Gamma \vdash e : \sigma$ exists, by Corollary D.7 we are guaranteed to obtain a solution C_s for C. Then by Corollary D.8 we know that $\Gamma \vdash e : \widehat{C_s}(\phi)$ and $\sigma \sim \pi\widehat{C_s}(\phi)$ for where π is a fully monomorphic substitution. Take $\sigma^\star = \widehat{C_s}(\phi)$ and we are done. □

Theorem 4.4 (Completeness)**.** *Let Γ be a closed environment and e an expression. If $\Gamma \vdash e : \sigma$ then $\Gamma \vdash e : \phi \rightsquigarrow C$ and C can reach a solved form.*

Proof. Consider a closed environment Γ and an expression e. By Corollary D.6 we know that if we generate constraints $\Gamma \vdash e : \phi \rightsquigarrow C$, then there exists a solution for C. In particular, since there exists a solution the constraint set C is *not* inconsistent.

Corollary D.7 states that every constraint set is either inconsistent or can be rewritten to a solved form. Given that our constraint set C is not inconsistent, this implies that it can be rewritten to a solved form, as desired. □