Practical aspects of evidence-based compilation in System FC

Dimitrios Vytiniotis Simon Peyton Jones
Microsoft Research, Cambridge
{dimitris,simonpj}@microsoft.com

Abstract

System FC is an explicitly typed language that serves as the target language for Haskell source programs. System FC is based on System F with the addition of erasable but explicit type equality proof witnesses. This paper improves FC in two directions: The first contribution is extending term-level functions with the ability to return equality proof witnesses, which allows the smooth integration of equality superclasses and indexed constraint synonyms, features currently absent from Haskell. We show how to ensure soundness and satisfy the zero-cost requirement for equality witnesses using a familiar mechanism, already present in GHC: that of unlifted types. Our second contribution is an equality proof simplification algorithm, which greatly reduces the size of the target System FC terms.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Abstract data types; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Design,Languages

Keywords Haskell, Type functions, System FC

1. Introduction

A typed intermediate language provides a firm place to stand, free from the design trade-offs of a complex source language. Moreover, type-checking the intermediate program provides a powerful consistency check on the earlier stages of elaboration, desugaring, and optimization. The Glasgow Haskell Compiler (GHC) has just such an intermediate language, which has evolved recently from System F to System FC (Sulzmann et al. 2007; Weirich et al. 2011) to accommodate the source-language features of GADTs (Cheney and Hinze 2003; Peyton Jones et al. 2006; Sheard and Pasalic 2004) and type families (Chakravarty et al. 2005; Kiselyov et al. 2010) In this paper we distill some lessons gained from our practical experience of implementing FC. Specifically, we offer two main contributions:

- Our earlier work treated equality evidence as a *type-level* phenomenon, completely erased before code generation begins. In this paper we show that it is both simpler and more expressive to treat equality evidence as a *term-level* value – without compromising the zero runtime cost property. (Section 3).
  - The approach allows us to maintain an erasable coercion language, where proofs may nevertheless be produced by runtime, potentially divergent, terms. We discuss several more traditional approaches to erasure in Section 5.
  - Moreover, the approach has greatly simplified the type inference and core typechecking implementation in GHC by treating type class evidence and coercion evidence uniformly, as ordinary expressions. A particularly important consequence is that it becomes easy to implement "equality super-classes", which were previously so awkward that GHC did not support them.
  - By treating coercions as ordinary expressions that have ordinary types we are able for the first time to support (type-indexed) equality constraint synonyms, subsuming the class constraint synonyms proposal of Orchard and Schrijvers (2010), a feature sought by Haskell programmers. [1]
- We present a novel coercion simplification algorithm which allows the compiler to replace a coercion with an equivalent but much smaller one (Section 4).
  - Coercion simplification is of great practical importance. We encountered programs whose un-simplified coercion terms grow to many times the size of the actual executable terms, to the point where GHC choked and ran out of heap. When the simplifier is enabled, coercions simplify to a small fraction of their size (Section 4.2).
  - To get these benefits, coercion simplification must take user-declared equality axioms into account, but the simplifier *must never loop* while optimizing a coercion – no matter which axioms are declared by users. We prove that this is the case in Section 4.3.

  Despite its great practical importance, coercion simplification did not appear to be well-studied in the coercion literature, but we give some connections to related work in Section 5.

2. Coercions as values

We begin by reviewing the role of an intermediate language. A rich, complex source language (Haskell) is desugared into a small, simple intermediate language. The source language is implicitly typed, and a type inference engine figures out the type of every binder and sub-expression. To make type inference feasible, Haskell embodies many somewhat ad-hoc design compromises; for example, λ-bound variables are assigned monomorphic types. By contrast, the intermediate language is simple, uniform, and explicitly typed. It can be typechecked by a simple, linear time algorithm.

To make this concrete, Figure 1 gives the syntax of System FC, the calculus implemented by GHC's intermediate language. The term language is mostly conventional, consisting of System F, together with let bindings, data constructors and case expressions. The syntax of a term encodes its typing derivation: every binder carries its type, and type abstractions $\Lambda a{:}\eta\,.\,e$ and type applications $e\,\varphi$ are explicit. The typing rules are given in Figure 2.

[1] `http://www.haskell.org/pipermail/haskell-cafe/2010-November/086701.html`

Kinds

η	$::=$	$\star \mid \kappa \to \kappa$	
κ	$::=$	$\# \mid \eta$	

Type constants

H	$::=$	T	Datatypes
	$\mid$	$(\to)$	Arrow
	$\mid$	$(\sim)$	Equality type

Types

$\varphi, \sigma, \tau, \upsilon$	$::=$	a	Variables
	$\mid$	H	Constants
	$\mid$	F	Type functions
	$\mid$	$\varphi_1\ \varphi_2$	Application
	$\mid$	$\forall a{:}\eta.\varphi$	Polymorphic types

Coercion values

γ, δ	$::=$	x	Variables
	$\mid$	$C\ \overline{\gamma}$	Axiom application
	$\mid$	$\gamma_1\ \gamma_2$	Application
	$\mid$	$\langle\varphi\rangle$	Reflexivity
	$\mid$	$\gamma_1\ ;\ \gamma_2$	Transitivity
	$\mid$	$sym\ \gamma$	Symmetry
	$\mid$	$nth\ k\ \gamma$	Injectivity
	$\mid$	$\forall a{:}\eta.\gamma$	Polymorphic coercion
	$\mid$	$\gamma @ \varphi$	Instantiation

Expressions

e, u	$::=$	$x \mid \lambda x{:}\sigma.\, e \mid e\ u$	
	$\mid$	$\Lambda a{:}\eta.\, e \mid e\ \varphi$	
	$\mid$	$K \mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	
	$\mid$	$\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ u$	
	$\mid$	$e \triangleright \gamma$	Casts
	$\mid$	$\lfloor\gamma\rfloor$	Coercions
p	$::=$	$K\ \overline{c{:}\eta}\ \overline{x{:}\tau}$	Patterns

Types of data constructors

$$K : \forall \overline{a{:}\eta_a}\ \overline{c{:}\eta_c}.\, \overline{\tau} \to T\ \overline{a}$$

Types of axioms

$$C : \forall \overline{a{:}\eta}.\, \varphi_1 \sim \varphi_2$$

Environments

Γ	$::=$	$\cdot \mid \Gamma, bnd$
bnd	$::=$	$a{:}\eta \mid x{:}\sigma$
	$\mid$	$K{:}\sigma \mid T{:}\kappa \mid F{:}\kappa \mid C{:}\sigma$

Notation

$T\ \overline{\tau}$	$\equiv$	$T\ \tau_1 \ldots \tau_n$
$\overline{\tau} \to \tau$	$\equiv$	$\tau_1 \to \ldots \to \tau_n \to \tau$
Γ_0	$\equiv$	initial (closed) environment

Figure 1: Syntax of System FC

$$\boxed{\Gamma \vdash^{\mathsf{tm}} e : \tau}$$

$$\frac{(x{:}\tau) \in \Gamma \quad \tau \neq \sigma_1 \sim \sigma_2}{\Gamma \vdash^{\mathsf{tm}} x : \tau}\ \textsc{EVar}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{ty}} \sigma : \kappa}{\Gamma \vdash^{\mathsf{tm}} \lambda x{:}\sigma.\, e : \sigma \to \tau}\ \textsc{EAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \sigma \to \tau \quad \Gamma \vdash^{\mathsf{tm}} u : \sigma}{\Gamma \vdash^{\mathsf{tm}} e\ u : \tau}\ \textsc{EApp}$$

$$\frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \Lambda a{:}\eta.\, e : \forall a{:}\eta.\, \tau}\ \textsc{ETAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \forall a{:}\eta.\, \tau \quad \Gamma \vdash^{\mathsf{ty}} \varphi : \eta}{\Gamma \vdash^{\mathsf{tm}} e\ \varphi : \tau[\varphi/a]}\ \textsc{ETApp}$$

$$\frac{\Gamma \vdash^{\mathsf{ty}} \sigma : \kappa \quad \Gamma \vdash^{\mathsf{tm}} u : \sigma \quad \Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \mathbf{let}\ x{:}\sigma = u\ \mathbf{in}\ e : \tau}\ \textsc{ELet}$$

$$\frac{\Gamma \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{co}} \gamma : \tau \sim \varphi}{\Gamma \vdash^{\mathsf{tm}} e \triangleright \gamma : \varphi}\ \textsc{ECast}$$

$$\frac{\Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash^{\mathsf{tm}} \lfloor\gamma\rfloor : \sigma_1 \sim \sigma_2}\ \textsc{ECoercion} \qquad \frac{(K{:}\sigma) \in \Gamma_0}{\Gamma \vdash^{\mathsf{tm}} K : \sigma}\ \textsc{ECon}$$

$$\frac{\begin{array}{l}(K{:}\forall \overline{a{:}\eta_a}\ \overline{c{:}\eta_c}.\, \overline{\varphi} \to T\ \overline{a}) \in \Gamma_0 \quad \Gamma \vdash e : T\ \overline{\sigma} \\ \text{for each branch } K\ \overline{c{:}\eta_c}\ \overline{x{:}\tau} \to u \\ \quad \tau_i = \varphi_i[\overline{\sigma}/\overline{a}] \\ \quad \Gamma, \overline{c{:}\eta_c}, \overline{x{:}\tau} \vdash u : \sigma\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \overline{K\ \overline{c{:}\eta_c}\ \overline{x{:}\tau} \to u} : \sigma}\ \textsc{ECase}$$

Figure 2: Well-formed terms

2.1 Coercions

The unusual feature of FC is the use of coercions, which we review briefly in this section. We urge the reader to consult (Sulzmann et al. 2007; Weirich et al. 2011) for more examples and intuition.

The term $e \triangleright \gamma$ is a cast, that converts a term e of type τ to one of type φ (rule ECAST in Figure 2). The coercion γ is a witness, providing evidence that τ and φ are equal types – that is, γ has type $\tau \sim \varphi$. The syntax of coercions γ is given in Figure 1, and their typing rules in Figure 4. To see casts in action, consider this Haskell program which uses GADTs:

```
data T a where
   T1 :: Int -> T Int
   T2 :: a -> T a
f :: T a -> [a]
f (T1 x) = [x+1]
f (T2 v) = [v]
main = f (T1 4)
```

We regard the GADT data constructor `T1` as having the type

$$\texttt{T1} : \forall a.(a \sim \texttt{Int}) \to \texttt{Int} \to \texttt{T}\ a$$

So in FC, `T1` takes three arguments: a type argument to instantiate a, a coercion witnessing the equivalence of a and `Int`, and a value of type `Int`. Here is the FC elaboration of `main`:

```
main = f Int (T1 Int <Int> 4)
```

The coercion argument has kind (`Int` $\sim$ `Int`), for which the evidence is just $\langle$`Int`$\rangle$ (reflexivity). Similarly, pattern-matching on `T1` binds two variables: a coercion variable, and a term variable. Here is the FC elaboration of `f`:

```
f = /\(a:*). \(x:T a).
    case x of
      T1 c n -> (Cons (n+1) Nil) |> sym [c]
      T2 v   -> Cons v Nil
main = f Int (T1 Int <Int> 4)
```

The cast converts the type of the result from `[Int]` to `[a]`. The coercion $sym[c]$ is evidence for (or a proof of) the equality of these types, using `c`, of type (`a` $\sim$ `Int`). Figure 4 gives the rules for coercions.

2.2 Equality superclasses

In our earlier work, coercions (like types) may be passed as an argument to a function, but may not be returned as a result. However, it is sometimes very convenient for a function to return a coercion. Haskell allows a type class to have a *superclass*; for example

```
class Eq a => Ord a where
   (>=) :: a -> a -> a
```

This defines class `Ord` to have a method `(>=)` and a superclass `Eq`. Every instance of `Ord` must be an instance of `Eq`. A class has a

runtime representation as a record, or "dictionary", of its methods and superclasses so, for example, from an `Ord` dictionary one can select the `(>=)` method or the `Eq` dictionary. In the FC elaboration, the `class` declaration is translated to a data type declaration and a collection of *selectors*:

```
data Ord a where
  MkOrd :: Eq a -> (a -> a -> a) -> Ord a

geq :: forall a. Ord a -> a -> a -> a
geq (MkOrd _ geq) = geq

eqOrd :: forall a. Ord a -> Eq a
eqOrd (MkOrd eq _) = eq
```

It is very desirable to extend the superclass idea to include equality constraints. For example, consider this Haskell class:

```
class (b ~ F a) => C a b where
  op :: a -> b
```

The equality superclass specifies that any types σ, τ that instantiate $\mathtt{C}\,\sigma\,\tau$ must satisfy $\tau \sim \mathtt{F}\,\sigma$. But what does the selector look like, the equivalent of `eqOrd`? The simple, uniform thing would be:

```
scC :: forall a b. C a b -> (b ~ F a)
scC (MkC eq _) = eq
```

Here, for the first time, we have a function that *returns* a coercion.

2.3 Elaboration simplifications

Having functions that return coercions is not just an aesthetic improvement, but rather a real implementation simplification of the source-to-target elaboration process. Assume we are given:

```
type family F
type instance F Bool = Char
instance C Bool Char where ...
```

GHC supports *type-level functions*, such as `F` above, that are described by *top-level axioms*, such as $\mathtt{Fax} : F\ \mathtt{Bool} \sim \mathtt{Char}$ above. Assume the variable $\mathtt{f} :: \forall ab.\, C\ a\ b \Rightarrow a \to b \to b$ is bound in the context. Elaboration of the term `(f True 'c') :: F Bool` will introduce fresh variables `d` and `co` for yet-unknown evidence:

```
f Bool Char d True 'c' |> co
```

The type inference constraint solver must fill in evidence for $\mathtt{d} : C\ \mathtt{Bool}\ \mathtt{Char}$ and $\mathtt{co} : \mathtt{Char} \sim F\ \mathtt{Bool}$. For instance it may use the class instance for `d` and the type family axiom for `co`. The final elaborated term is:

```
let d = dCBoolChar -- Dict. from instance decl.
in f Bool Char d True 'c' |> sym Fax
```

But what happens if instead of `Fax`, a constraint solver picks up the equality superclass from `d`? If coercions cannot be returned, the only option is to elaborate the original term using pattern matching:

```
let d = dCBoolChar
in case d of
     MkC co _ -> f Bool Char d True 'c' |> co
```

This means that, after type inference is finished, we must "desugar" the term adding `let`-bindings in some cases, or adding pattern matching wrappers in other cases. More complex evidence terms that may depend on other evidence add even more complexity to this desugaring phase.

Instead, we'd like to be able to simply return and bind coercions, just as we do for dictionaries:

```
let d  = dCBoolChar
    co = scC d
in f Bool Char d True 'c' |> co
```

$$\boxed{\Gamma \vdash^{\mathsf{ty}} \tau : \kappa}$$

$$\frac{(a{:}\eta) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} a : \eta}\ \textsc{TVar} \qquad \frac{(T{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} T : \kappa}\ \textsc{TData} \qquad \frac{(F{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} F : \kappa}\ \textsc{TFun}$$

$$\frac{\kappa_1, \kappa_2 \in \{\#, \star\}}{\Gamma \vdash^{\mathsf{ty}} (\to) : \kappa_1 \to \kappa_2 \to \star}\ \textsc{TArr} \qquad \frac{}{\Gamma \vdash^{\mathsf{ty}} (\sim) : \kappa \to \kappa \to \#}\ \textsc{TEqPred}$$

$$\frac{\Gamma \vdash^{\mathsf{ty}} \varphi_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash^{\mathsf{ty}} \varphi_2 : \kappa_1}{\Gamma \vdash^{\mathsf{ty}} \varphi_1\ \varphi_2 : \kappa_2}\ \textsc{TApp} \qquad \frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{ty}} \tau : \star}{\Gamma \vdash^{\mathsf{ty}} \forall a{:}\eta.\, \tau : \star}\ \textsc{TAll}$$

Figure 3: Well-formed types

No special treatment is required then during desugaring, and the mechanism for handling (potentially recursive) term-level bindings is readily applicable to coercions.

2.4 Coercion erasure

An elaborated program can contain many casts and coercions. But we do not want them to make the program run slower, because coercions in FC have no computational content. In the end, at runtime they all boil down to the identity function, so a cast can be implemented by a no-op. In this respect coercions are very like types: perhaps we can simply erase them? That was the approach we took in our earlier work (Sulzmann et al. 2007), and in our implementation in GHC. It works fine for the original version of FC, but we discovered that it was not quite expressive enough (Section 2.2), and that in turn has led us to re-think erasure.

Even in the original FC there was a whiff of a problem. In the source language, equality constraints are treated uniformly with type-class constraints and implicit parameters: anywhere a class constraint can appear, an equality constraint can appear, and vice versa. But in the original FC, types and coercions are erased, while class and implicit-parameter constraints (which have computational content) are not. This gave rise to many annoying (but manageable) special cases in the compiler.

The question of equality superclasses highlights the non-uniformity. There is no problem with a function that returns evidence for a class constraint (such as `eqOrd` above), so why should there be a problem with one that returns evidence for an equality constraint?

2.5 Outline of the solution

The solution should be evident by now: instead of treating coercions like types, treat them like values, uniformly to other forms of evidence. So a coercion, like a dictionary, becomes a first-class value that can be passed to a function, returned as a result, stored in data structure and so on. We describe the modified language in Section 3. What becomes of our claim that casts and coercion-passing should cost nothing? After all, coercions still have no computational content. Happily, it turns out that we can use a form of value erasure to achieve this, and one that is already present in GHC for a completely different reason, as we shall see in Section 3.4.

3. The new FC language

We now give the full story for the new coercions-as-values version of FC. The syntax is in Figure 1, while the typing rules are in Figures 2, 3, and 4. The main new feature is that coercion values $\lfloor\gamma\rfloor$

$$\boxed{\Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2}$$

$$\frac{(x{:}\sigma_1 \sim \sigma_2) \in \Gamma}{\Gamma \vdash^{\mathsf{co}} x : \sigma_1 \sim \sigma_2}\ \textsc{CVar} \qquad \frac{(C{:}\forall \overline{a{:}\eta}.\, \tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \vdash^{\mathsf{co}} \gamma_i : \sigma_i \sim \varphi_i}{\Gamma \vdash^{\mathsf{co}} C\ \overline{\gamma} : \tau_1[\overline{\sigma}/\overline{a}] \sim \tau_2[\overline{\varphi}/\overline{a}]}\ \textsc{CAx}$$

$$\frac{\Gamma \vdash^{\mathsf{co}} \gamma_1 : \sigma_1 \sim \sigma_2 \quad \Gamma \vdash^{\mathsf{co}} \gamma_2 : \sigma_2 \sim \sigma_3}{\Gamma \vdash^{\mathsf{co}} \gamma_1\,;\gamma_2 : \sigma_1 \sim \sigma_3}\ \textsc{CTrans} \qquad \frac{\Gamma \vdash^{\mathsf{ty}} \varphi : \kappa}{\Gamma \vdash^{\mathsf{co}} \langle \varphi \rangle : \sigma \sim \sigma}\ \textsc{CRefl}$$

$$\frac{\Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash^{\mathsf{co}} sym\,\gamma : \sigma_2 \sim \sigma_1}\ \textsc{CSym} \qquad \frac{\Gamma \vdash^{\mathsf{co}} \gamma : H\ \overline{\sigma} \sim H\ \overline{\tau}}{\Gamma \vdash^{\mathsf{co}} nth\ k\ \gamma : \sigma_k \sim \tau_k}\ \textsc{CNth} \qquad \frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash^{\mathsf{co}} \forall a{:}\eta.\gamma : (\forall a{:}\eta.\sigma_1) \sim (\forall a{:}\eta.\sigma_2)}\ \textsc{CAll}$$

$$\frac{\Gamma \vdash^{\mathsf{co}} \gamma_1 : \sigma_1 \sim \sigma_2 \quad \Gamma \vdash^{\mathsf{co}} \gamma_2 : \varphi_1 \sim \varphi_2 \quad \Gamma \vdash^{\mathsf{ty}} : \sigma_1\ \varphi_1 : \kappa}{\Gamma \vdash^{\mathsf{co}} \gamma_1\ \gamma_2 : \sigma_1\ \varphi_1 \sim \sigma_2\ \varphi_2}\ \textsc{CApp} \qquad \frac{\Gamma \vdash^{\mathsf{ty}} \varphi : \eta \quad \Gamma \vdash^{\mathsf{co}} \gamma : (\forall a{:}\eta.\sigma_1) \sim (\forall a{:}\eta.\sigma_2)}{\Gamma \vdash^{\mathsf{co}} \gamma@\varphi : \sigma_1[\varphi/a] \sim \sigma_2[\varphi/a]}\ \textsc{CInst}$$

Figure 4: Well-formed coercions

$$\boxed{[x{\mapsto}u]^{\mathsf{tm}}(e) = e'}$$

$$\begin{array}{rcl} [x{\mapsto}u]^{\mathsf{tm}}(e\ \triangleright\ \gamma) & = & [x{\mapsto}u]^{\mathsf{tm}}(e)\ \triangleright\ [x{\mapsto}u]^{\mathsf{co}}(\gamma) \\ [x{\mapsto}u]^{\mathsf{tm}}(\lfloor \gamma \rfloor) & = & \lfloor [x{\mapsto}u]^{\mathsf{co}}(\gamma) \rfloor \\ & etc & \end{array}$$

$$\boxed{[x{\mapsto}u]^{\mathsf{co}}(\gamma) = \gamma'}$$

$$\begin{array}{rcl} [x{\mapsto}\lfloor \gamma_1 \rfloor]^{\mathsf{co}}(\gamma) & = & \gamma[\gamma_1/x] \\ [x{\mapsto}\lfloor \gamma_1 \rfloor\ \triangleright\ \gamma_2]^{\mathsf{co}}(\gamma) & = & \gamma[sym(nth\ 1\ \gamma_2);\gamma_1;(nth\ 2\ \gamma_2)/x] \\ [x{\mapsto}u]^{\mathsf{co}}(\gamma) & = & \gamma \quad \text{otherwise} \end{array}$$

Figure 5: Term and coercion substitutions

$$\boxed{[a \mapsto \gamma]\!\uparrow\!(\tau) = \gamma'}$$

$$\begin{array}{rcl} [a \mapsto \gamma]\!\uparrow\!(a) & = & \gamma \\ [a \mapsto \gamma]\!\uparrow\!(b) & = & \langle b \rangle \\ [a \mapsto \gamma]\!\uparrow\!(H) & = & \langle H \rangle \\ [a \mapsto \gamma]\!\uparrow\!(F) & = & \langle F \rangle \\ [a \mapsto \gamma]\!\uparrow\!(\tau_1\ \tau_2) & = & \begin{cases} \langle \varphi_1\ \varphi_2 \rangle \text{ when } [a \mapsto \gamma]\!\uparrow\!(\tau_i) = \langle \varphi_i \rangle \\ ([a \mapsto \gamma]\!\uparrow\!(\tau_1))\ ([a \mapsto \gamma]\!\uparrow\!(\tau_2)) \\ \quad \text{otherwise} \end{cases} \\ [a \mapsto \gamma]\!\uparrow\!(\forall a{:}\eta.\tau) & = & \begin{cases} \langle \forall a{:}\eta.\varphi \rangle \text{ when } [a \mapsto \gamma]\!\uparrow\!(\tau) = \langle \varphi \rangle \\ \forall a{:}\eta.([a \mapsto \gamma]\!\uparrow\!(\tau)) \text{ otherwise} \end{cases} \end{array}$$

Figure 6: Lifting types to coercions

are now a form of term e, rather than appearing only in applications (as do types). There is a naturally-corresponding typing rule (ECoerce in Figure 2), which in turn means that $\sigma \sim \tau$ is a type. Since Haskell and FC have a higher-order type system, we define $(\sim)$ to be a type constructor, and construct $\sigma \sim \tau$ using ordinary type application (Figure 1).

Type-level functions are denoted with F, and we use C for top-level equality axioms (Figure 1). Rather than give concrete syntax for declarations of data types, type functions, and axioms, we simply populate the initial top-level type environment Γ_0 with their kinds, as the syntax of Γ in Figure 1 shows. To take an example, the Haskell declarations

```
type family F a :: *
type instance F [a] = [F a]
type instance F Int = Bool
```

give rise to the following bindings in Γ_0:

$$\begin{array}{ll} F & : \star \to \star \\ C_1 & : \forall a{:}\star \to \star.\, F\,[a] \sim [F\ a] \\ C_2 & : F\ \texttt{Int} \sim \texttt{Bool} \end{array}$$

Another notable feature is that the syntax and rule CAx in Figure 4 apply axioms to *coercions* ($C\ \overline{\gamma}$) and not simply *types* ($C\ \overline{\tau}$), contrary to previous FC presentations. This generalization does not improve expressivity[2] but is of great importance for coercion simplification as we shall see towards the end of Section 4.1.

3.1 Coercions are unlifted

Here is an unsettling expression:

```
letrec (loop:() -> (Int~Bool)) = \x. loop x
in let (g:Int~Bool) = loop ()
in (4 |> g) && True
```

(Our formalism lacks `letrec` for brevity, but our implementation supports it of course.) We bind a coercion `g` to the divergent term `loop ()`, and then use `g` to cast 4 to `Bool`. If we were to actually take the `&&` of 4 with `True` we would get a segmentation fault (or something worse). Yet the program is well-typed, using the coercion `g` whose kind is `(Int~Bool)`, which in turn is returned by the diverging function `loop`. So what has become of type soundness?

In any system that allows equality evidence to be computed and returned by a possibly-divergent function we must *ensure that evidence is evaluated before it is acted upon*. In a call-by-value language this would be automatic, but Haskell and FC are call-by-need. So the key to type soundness is to ensure that equality evidence is always evaluated in a call-by-value fashion. (Exactly the same applies to type-class dictionaries, but since they have computational content, using them necessarily forces evaluation.)

Fortunately, although FC is call-by-need, its realization in GHC already has a type-directed call-by-value mechanism, for a completely different reason: unlifted types (Peyton Jones and Launchbury 1991). Here is GHC's implementation of addition on `Int`:

```
data Int = I# Int#
plusInt :: Int -> Int -> Int
plusInt (I# x) (I# y)
  = let z = x +# y
    in I# z
```

An `Int` is an ordinary algebraic data type with a single constructor `I#` (the '`#`' is not special; it is just part of the constructor name).

[2] Exercise: encode the coercion $C\ \overline{\gamma}$ as the transitive composition of coercions that mention $C\ \langle \tau_1 \rangle \ldots \langle \tau_n \rangle$.

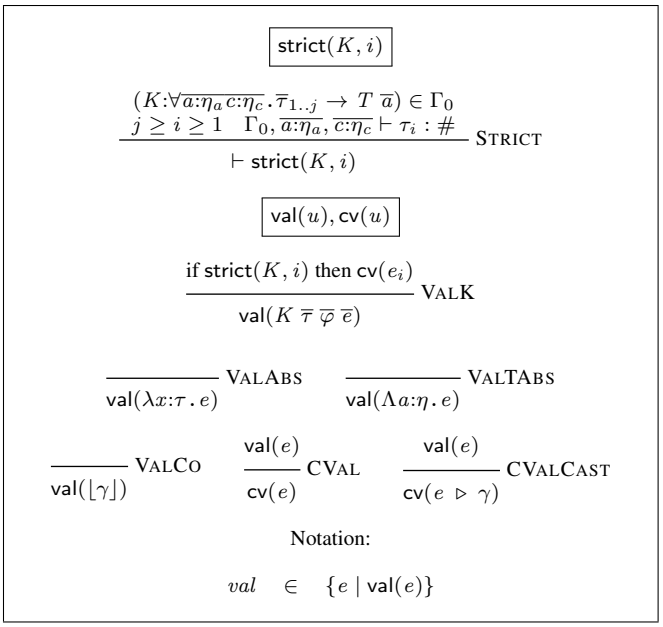

Figure 7: Value forms

This constructor has a single argument, of type `Int#`, which is the type of unboxed integers, a honest-to-goodness 32-bit integer value like C's `int`. A value of type `Int` is always heap-allocated and can be a thunk; a value of type `Int#` is never heap-allocated, and cannot be a thunk – it is always a value. So there is no bottom value of type `Int#`, and we say that it is an unlifted type.

Unlifted types are always treated using call-by-value. For example the `let` binding for `z` does not heap-allocate a thunk, as `let` does for lifted types. Rather, the right-hand side is evaluated immediately and the result is bound to `z`.

This is just what we want for coercions: all we need do is to declare that the type of a coercion, $\sigma \sim \tau$, is an unlifted type, and the existing mechanisms will do the rest. In the troubling example above, the program will loop (trying to evaluate `loop`) but it will not seg-fault. Of course, a good compiler will try to detect unsatisfiable equality constraints, such as `Int~Bool`, and flag a warning that the program is probably erroneous, but in general unsatisfiability is undecidable, so we need a solid base that does not depend on detecting unsatisfiablity.

Formally, we use kinds to distinguish lifted and unlifted types: A lifted type has kind $\star$ while an unlifted one has kind $\#$ (Figure 1). We use η for a non-$\#$ kind and make sure in Figure 3 that all quantified type variables have η-kinds. Importantly, type constructor $(\sim)$ returns $\#$ (rule TEQPRED) to ensure that $\sigma \sim \tau$ is unlifted.

3.2 Operational semantics

The kind distinctions we have introduced above are used in the operational semantics to guide the evaluation order, in Figure 8. Its value forms are given in Figure 7. We follow (Sulzmann et al. 2007) and define *values* $\mathsf{val}(\cdot)$ and *coerced values* $\mathsf{cv}(\cdot)$. Values include constructor applications, λ-abstractions and type abstractions, and the new form of coercion values $\lfloor\gamma\rfloor$ (absent in (Sulzmann et al. 2007)). Data constructor applications are considered values only when the arguments in which the constructor is strict are *coerced values*. Coerced values are either just values, or values cast by a coercion. For instance if we have an axiom $C{:}F\ Int \sim [Int]$, then $(Nil\ [Int]) \triangleright sym\ C$ is such a coerced value. Therefore the design goal of the operational semantics will be:

> A closed well-typed expression is either a coerced value or can step to a well-typed expression.

Turning to the operational semantics in Figure 8, valid evaluation contexts $\mathcal{E}$ are classified with the judgement $\vDash^{\mathsf{E}} \mathcal{E}$. For instance, not all `let`-expressions are valid evaluation contexts, but only those that bind an argument whose type is unlifted (rule ELETOK). Similarly, not all constructor applications are valid evaluation contexts, but only those where the constructor is strict in the argument we apply (rule ECAPPOK). The rest of the rules are straightforward. Rule EVAL decomposes a term into an evaluation context and a redex and uses the relation $\rightsquigarrow$ to perform a one-step reduction.

The rules for $\rightsquigarrow$ are almost identical to those appearing in our previous work, but for the sake of self-containment we give a brief description. Rules PUSH and TPUSH push coercions when they stand in the way of a term or type application. Rule APPLET translates applications of λ-expressions to `let`-expressions. Rule TMBETA performs a substitution when the bound expression is either a coerced value or has lifted kind – in all other cases we have to keep evaluating as rule ELETOK suggests. Rule COMB combines two casts and rule CASE reduces a pattern match. Rule KPUSH finally pushes a coercion that stands in the way of pattern matching, by introducing new coercions $[\overline{a} \mapsto \overline{\delta}]\uparrow(\sigma_i[\overline{\varphi}/c])$.

The notation $[a \mapsto \gamma]\uparrow(\tau) = \gamma'$ describes how a type can be *lifted* to become a coercion, by substituting coercions for its free variables, and is defined in Figure 6. Lifting satisfies the following:

Lemma 3.1 (Lifting)**.** *If* $\Gamma, (a{:}\eta) \vdash^{\mathsf{ty}} \tau : \kappa$ *and* $\Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$ *for* $\Gamma \vdash^{\mathsf{ty}} \sigma_i : \eta$ *then* $\Gamma \vdash^{\mathsf{co}} [a \mapsto \gamma]\uparrow(\tau) : \tau[\sigma_1/a] \sim \tau[\sigma_2/a]$.

With this property of lifting in mind, the reader can verify that rule KPUSH produces a well-typed term. As an aside, observe that lifting produces coercions in which reflexivity has been pushed as high as possible in the result coercion tree structure – this is not strictly necessary for soundness but will be handy for the coercion simplifier presentation in Section 4.

Interesting details hide in the definition of the substitution function $[x{\mapsto}u]^{\mathsf{tm}}(e)$, in Figure 5. This function is standard for terms, but appeals to a different function $[x{\mapsto}u]^{\mathsf{co}}(\gamma)$ when it crosses a coercion boundary. A variable that we substitute for, x, may either be of type $\tau_1 \sim \tau_2$ or not. If it is of type $\tau_1 \sim \tau_2$ then its kind is $\#$, which means that u must be a coerced value and by inversion on the value judgement must have one of the following two forms:

- $u = \lfloor\gamma_1\rfloor$ in which case we may simply substitute γ_1 in γ, or
- $u = \lfloor\gamma_1\rfloor \triangleright \gamma_2$, in which case $\vdash^{\mathsf{co}} \gamma_1 : \sigma_1 \sim \varphi_1$, $\vdash^{\mathsf{co}} \gamma_2 : (\sigma_1 \sim \varphi_1) \sim (\sigma_2 \sim \varphi_2)$, and $\vdash^{\mathsf{tm}} u : \sigma_2 \sim \varphi_2$. Here γ_2 is a coercion between coercions, a *higher-dimensional* construction in the terminology of recent re-formulations of dependent type theory (Licata and Harper 2011). Luckily in our case we can immediately collapse the term $\lfloor\gamma_1\rfloor \triangleright \gamma_2$ to a coercion of type $\sigma_2 \sim \varphi_2$, taking advantage of injectivity of $(\sim)$ and symmetry (features that may or may not be present in more general settings). The collapsed term is just $sym(nth\ 1\ \gamma_2)\ ;\ \gamma_1\ ;\ (nth\ 2\ \gamma_2)$.

Otherwise u will not be of type $\tau_1 \sim \tau_2$ and hence there is no substitution to perform since all variables in γ will be of type $\tau_1 \sim \tau_2$ by rule CVAR in Figure 4. This is reflected in the third line of the definition of $[x{\mapsto}u]^{\mathsf{co}}(\gamma)$.

Formally, subject reduction is proved with the next three results.

Lemma 3.2 (Substitution)**.** *The following are true:*

- *If* $\Delta, (x{:}\tau) \vdash^{\mathsf{tm}} e : \sigma$ *and* $\Delta \vdash^{\mathsf{ty}} \tau : \#$ *and* $\Delta \vdash u : \tau$ *and* $\Delta \vdash \mathsf{cv}(u)$ *then* $\Delta \vdash^{\mathsf{tm}} [x{\mapsto}u]^{\mathsf{tm}}(e) : \sigma$.
- *If* $\Delta, (x{:}\tau) \vdash^{\mathsf{tm}} e : \sigma$ *and* $\Delta \vdash^{\mathsf{ty}} \tau : \eta$ *and* $\Delta \vdash u : \tau$ *then* $\Delta \vdash^{\mathsf{tm}} [x{\mapsto}u]^{\mathsf{tm}}(e) : \sigma$.

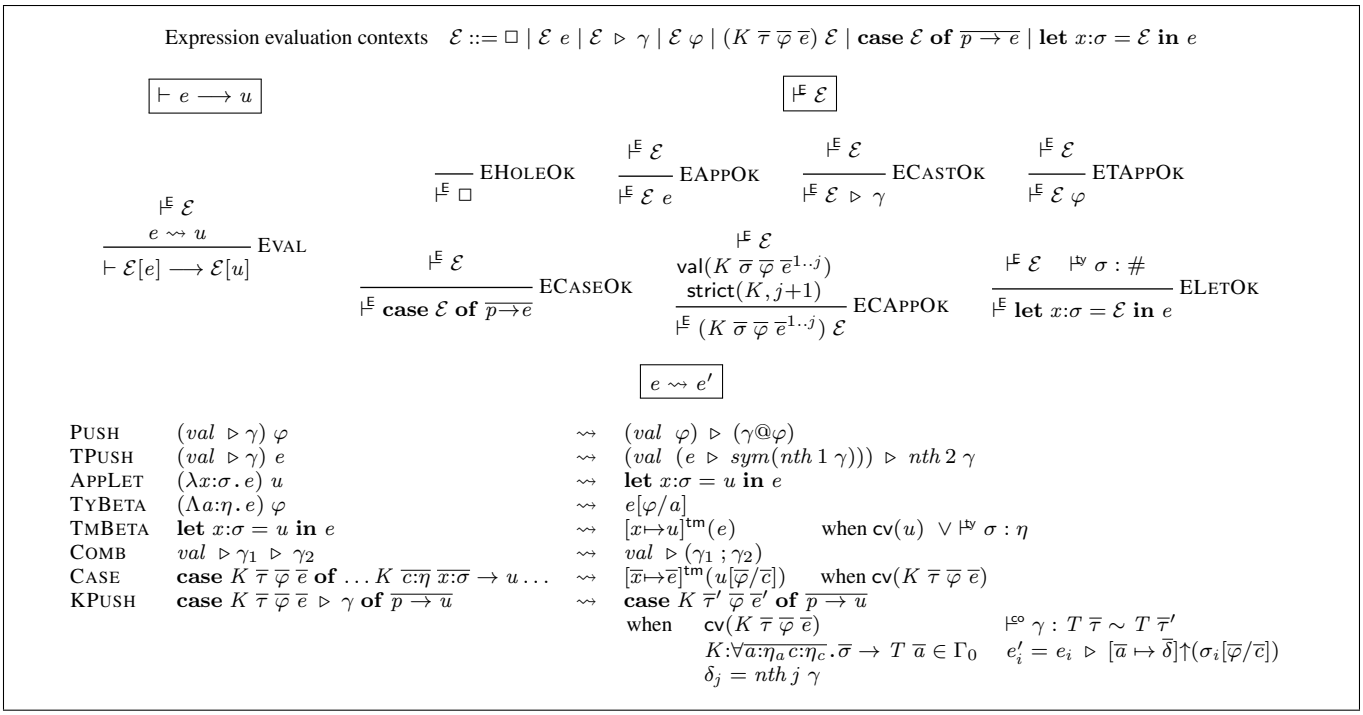

Figure 8: Typed operational semantics

Lemma 3.3 (Type-in-coercion substitution)**.** *If* $\Gamma, (a{:}\eta) \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$ *and* $\Gamma \vdash^{\mathsf{ty}} \tau : \eta$ *then* $\Gamma \vdash^{\mathsf{co}} \gamma[\tau/a] : \sigma_1[\tau/a] \sim \sigma_2[\tau/a]$.

Theorem 3.4 (Subject reduction)**.** *The following are true:*

- *If* $\vdash^{\mathsf{tm}} e : \tau$ *and* $e \rightsquigarrow u$ *then* $\vdash^{\mathsf{tm}} u : \tau$.
- *If* $\vdash^{\mathsf{tm}} e : \tau$ *and* $e \longrightarrow u$ *then* $\vdash^{\mathsf{tm}} u : \tau$.

Finally, progress relies on the assumption of *consistency*, transferred directly from our previous work. Consistency requires that for every two closed types τ_1 and τ_2 that are inhabited by values (including applications of $(\sim)$), if we can derive $\vdash^{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ then τ_1 and τ_2 must have the same head constructor.

Theorem 3.5 (Progress)**.** *Under the assumption of consistency, if* $\vdash^{\mathsf{tm}} e : \tau$ *then either* $e \longrightarrow e'$ *or* $\mathsf{cv}(e)$.

3.3 Coercions as zero-width values

If coercions are runtime values, one might worry about the runtime cost of passing them around. They cannot be erased altogether because, as we have seen, we must evaluate evidence before using it. If we erased all traces of coercions and their computations, the system would become unsound, as we saw in Section 3.1.

Happily, a solution lies readily to hand. Although we cannot erase coercions altogether, once a coercion is evaluated we do not need to look at it. Its structure is irrelevant to the computation, so it can be represented by a very narrow bit-field indeed. We have already seen that `Int#` is an unboxed integer, represented by a 32-bit value, and typically passed in a register. Similarly `Double#` is an unboxed double-precision float, represented by a 64-bit value, typically passed in a floating point register. So we are free to decide that $\sigma \sim \tau$ is an unboxed type, represented by a *zero-bit value*, and passed in, well, a zero-bit register, in a manner reminiscent of the `void` type in C:

- When a function takes a coercion as an argument, we can pass it in a zero-bit register (using call-by-value to evaluate the coercion first, of course).
- When a function returns a coercion, we can return it in a zero-bit register.
- When allocating a data constructor with a coercion argument (such as `T1` in Section 2.1), we can allocate zero bits to hold the coercion.

We have an infinite number of zero-bit registers available, and it takes no instructions to load and store zero-bit values... in short, coercions have no runtime overhead at all. Better still, the mechanism for supporting zero-width values is already present in GHC for another unrelated reason: the I/O monad. In GHC, the I/O monad is implemented like this:

```
type RW = State# RealWorld
newtype IO a = IO (RW -> (# RW, a #)
```

A value of type `RW` is a kind of token representing the state of the world. An I/O computation may transform the world, and hence returns a new `RW` token. However, the state token is passed around merely to maintain the dependency order of I/O operations. Like coercions, it has no computational content, because I/O operations are performed by side effect on the (actual) real world. So values of type `(State# a)` are represented by a zero-width value, and this mechanism is precisely what we need to support coercions.

3.4 Erasure

To be concrete, we formalize the zero-width register idea in a low-level untyped language with facilities for strict evaluation and a "zero-width" value. This language is given in Figure 9, and is a usual λ-calculus with strict λ-abstractions ($\lambda!x.\mathtt{e}$), strict `let` ($\mathbf{let}\ !x = \mathtt{e}\ \mathbf{in}\ \mathtt{u}$), and a zero-width value $\bullet$ (pronounced "spot").

The erasure of a System FC term e is given by the function $\{x\}_\Delta$ which also accepts a type environment Δ. The first interesting case is erasure for λ-abstractions, which – depending on the kind of the type of the argument – erases to strict or non-strict λ-terms. Hence it needs to consult the environment Δ to determine the kind of the

Erasure language

$$
\begin{array}{rcl}
\mathtt{e} & ::= & x \mid \lambda x\,.\,\mathtt{e} \mid \lambda!x\,.\,\mathtt{e} \mid \mathtt{e}\ \mathtt{u} \mid K \mid \mathbf{case}\ \mathtt{e}\ \mathbf{of}\ \overline{\mathtt{p} \to \mathtt{u}} \\
 & \mid & \mathbf{let}\ x = \mathtt{e}\ \mathbf{in}\ \mathtt{u} \mid \mathbf{let}\ !x = \mathtt{e}\ \mathbf{in}\ \mathtt{u} \mid \bullet \\
\mathtt{p} & ::= & K\ \overline{x}
\end{array}
$$

$$\boxed{\{e\}_\Delta = \mathtt{e}}$$

$$
\begin{array}{lcl}
\{x\}_\Delta & = & x \\
\{K\}_\Delta & = & K \\
\{\lambda x{:}\sigma\,.\,e\}_\Delta & = & \lambda x\,.\,\{e\}_\Delta \quad \text{when } \Delta \vdash^{\mathsf{ty}} \sigma : \eta \\
\{\lambda x{:}\sigma\,.\,e\}_\Delta & = & \lambda!x\,.\,\{e\}_\Delta \quad \text{when } \Delta \vdash^{\mathsf{ty}} \sigma : \# \\
\{e\ u\}_\Delta & = & \{e\}_\Delta\ \{u\}_\Delta \\
\{\Lambda a{:}\eta\,.\,e\}_\Delta & = & \lambda!x\,.\,\{e\}_{\Delta,(a:\kappa)} \\
\{e\ \varphi\}_\Delta & = & \{e\}_\Delta\ \bullet \\
\{\mathbf{case}\ e\ \mathbf{of} & & \\
\quad \overline{K\ \overline{c{:}\eta_c}\ \overline{x{:}\tau}{\to}u}\}_\Delta & = & \mathbf{case}\ \{e\}_\Delta\ \mathbf{of}\ \overline{K\,\overline{x}_c\,\overline{x}{\to}\{u\}_{\Delta,\overline{c:\eta_c}}} \\
\{\mathbf{let}\ x{:}\sigma = e\ \mathbf{in}\ u\}_\Delta & = & \mathbf{let}\ !x = \{e\}_\Delta\ \mathbf{in}\ \{u\}_\Delta \\
 & & \text{when } \Delta \vdash^{\mathsf{ty}} \sigma : \# \\
\{\mathbf{let}\ x{:}\sigma = e\ \mathbf{in}\ u\}_\Delta & = & \mathbf{let}\ x = \{e\}_\Delta\ \mathbf{in}\ \{u\}_\Delta \\
 & & \text{when } \Delta \vdash^{\mathsf{ty}} \sigma : \eta \\
\{e \triangleright \gamma\}_\Delta & = & \{e\}_\Delta \\
\{\lfloor\gamma\rfloor\}_\Delta & = & \bullet
\end{array}
$$

Figure 9: Erasure

argument. For instance, if the argument type is $\sigma_1 \sim \sigma_2$, its kind will be $\#$, and the resulting λ-term will be strict. The rules for `let` translate to either strict or ordinary `let` expressions depending on the type of the argument. Casts are completely erased and lifted coercions $\lfloor\gamma\rfloor$ are erased to $\bullet$. Finally, we convert type abstractions $\Lambda a{:}\eta\,.\,e$ to strict λ-terms, and correspondingly type applications to applications to $\bullet$. This is a divergence from Haskell's operational semantics that simply discards type abstractions altogether but is arguably more satisfactory and may also shed some light on erasure if *types*, in addition to coercions, are returned by terms.

We can define a straightforward operational semantics of this language and prove a bisimulation theorem between System FC terms and their erasure – the details are not particularly interesting.

3.5 Indexed constraint synonyms

Another advantage of coercions-as-values mentioned in the introduction is that they naturally allow *equality constraint synonyms*, subsuming previous proposals for class constraint synonyms (Orchard and Schrijvers 2010). For instance, consider first a map class declaration à la *associated type synonyms* (Chakravarty et al. 2005):

```
class Map a where
  type Elem a, Key a
  find :: Key a -> a -> Maybe (Elem a)
  ...
```

It defines the class of a map whose keys are given by the type family Key and elements by $Elem$. The modifications to System FC allow us now to define a constraint for *maps with integer keys* as:

```
type family IMap a
type instance IMap a = (Key a ~ Int, Map a)
```

Simply pairing-up an equality constraint with a class constraint is now entirely possible. Incidentally, considering coercions as values allows us to even define *indexed* equality synonyms, such as:

```
F [e]           = Key [e]       ~ Int
F (Trie e)      = Key (Trie e) ~ BitString
```

Though we have not yet identified very compelling cases for such indexed equality constraint synonyms, our formalism readily supports them.

4. Coercion simplification

System FC terms arise as the result of elaboration of source language terms through type inference or checking. Type inference typically relies on a constraint solver which produces System FC witnesses of equality, that decorate the elaborated term. A constraint solver is not typically concerned with producing small or readable witnesses; indeed GHC's constraint solver can produce large and complex coercions that can make the elaborated term practically impossible to understand and debug.

Moreover, GHC's optimizer transforms FC terms to FC terms. Any decent optimizer contains a symbolic evaluator, and GHC is no exception, so it faithfully implements the operational semantics of Figure 8. Some of these rules, notably the "push" rules, increase the size of coercion terms – and may be repeatedly applied.

Here is such a coercion, produced by GHC's constraint solver.[3]

```
axiom  Cn a :: N a ~ forall xy. a x -> a y
axiom  Cf   :: F () ~ Maybe

nth 2 (inst (inst (trans
                   (sym (Cn Maybe))
                   (trans (N (sym Cf))
                          (Cn (F ()))))
            xa) ya) :: Maybe ya ~ F () ya
```

Coercion simplification is designed to reduce the size of the proof terms inside an FC term. The above coercion is transformed by GHC's simplifier that we describe in the rest of this section to:

```
(sym Cf) ya :: Maybe ya ~ F () ya
```

A clear improvement!

4.1 Coercion simplification rules

Coercion simplification is given as a non-deterministic algorithm, presented in Figure 10. In this figure we use some syntactic conventions: Namely, for sequences of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$, we write $\overline{\gamma_1\,;\,\gamma_2}$ for the sequence of pointwise transitive compositions and $sym\,\overline{\gamma}_1$ for pointwise application of symmetry. We write $good(\gamma)$ iff γ *contains* some variable x or axiom application $C\ \overline{\gamma}$.

We define coercion evaluation contexts, $\mathcal{G}$, as coercion terms with holes inside them. The syntax of $\mathcal{G}$ allows us to rewrite anywhere inside a coercion. The main coercion evaluation rule is CoEval. If we are given a coercion γ, we first decompose it to an evaluation context $\mathcal{G}$ with γ_1 in its hole. Rule CoEval works up to associativity of transitivity; for example, we will allow the term $(\gamma_1\,;\,\gamma_2;)\,;\,\gamma_3$ to be written as $\mathcal{G}[\gamma_2\,;\,\gamma_3]$ where $\mathcal{G} = \gamma_1\,;\,\Box$. This treatment of transitivity is extremely convenient, but we must be careful to ensure that our argument for termination remains robust under associativity (Section 4.3). Once we have figured out a decomposition $\mathcal{G}[\gamma_1]$, CoEval performs a single step of rewriting $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$. Since we are allowed to rewrite coercions under a type environment ($\forall a{:}\eta\,.\,\mathcal{G}$ is a valid coercion evaluation context), Δ (somewhat informally) enumerates the type variables bound by $\mathcal{G}$. Finally we may simply return $\mathcal{G}[\gamma_2]$. The soundness property for the $\longrightarrow$ relation is given by the following theorem.

Theorem 4.1 (Coercion subject reduction)**.** *If* $\vdash^{\mathsf{co}} \gamma : \sigma \sim \varphi$ *and* $\gamma \longrightarrow \gamma'$ *then* $\vdash^{\mathsf{co}} \gamma : \sigma \sim \varphi$.

The rewriting judgement $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ satisfies a similar property.

Lemma 4.2. *If* $\Delta \vdash^{\mathsf{co}} \gamma_1 : \sigma \sim \varphi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\Delta \vdash^{\mathsf{co}} \gamma_2 : \sigma \sim \varphi$.

To explain coercion simplification, we now present the reaction rules for the $\rightsquigarrow$ relation, organized in several groups.

[3] We will give its transcription to our mathematical notation later; for brevity we do not explicitly show uses of reflexivity in the example.

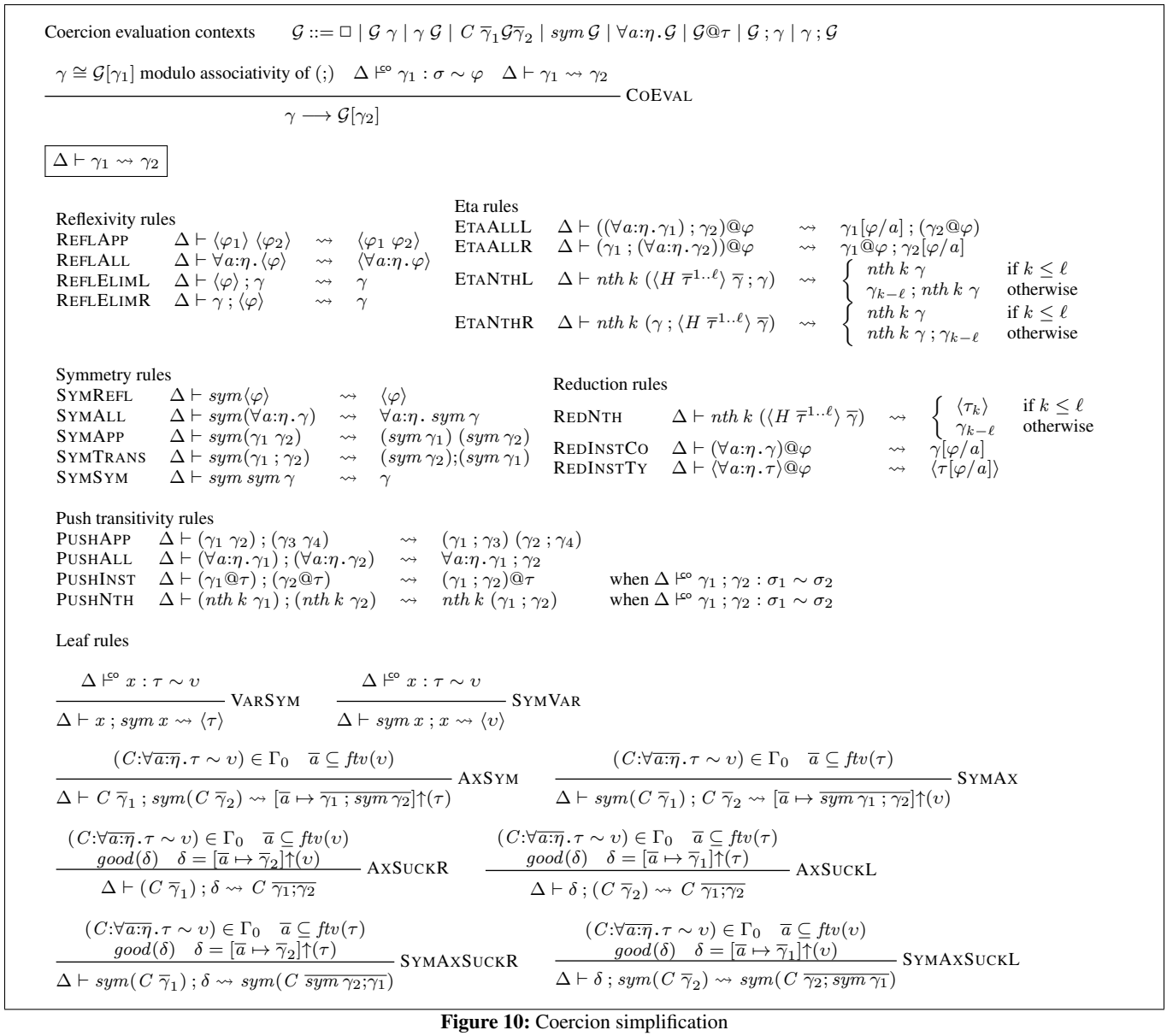

Figure 10: Coercion simplification

Pulling reflexivity up Rules ReflApp, ReflAll, ReflElimL, and ReflElimR, deal with uses of reflexivity. Rules ReflApp and ReflAll "swallow" constructors from the coercion language (coercion application, and quantification respectively) into the type language (type application, and quantification respectively). Hence they pull reflexivity as high as possible in the tree structure of a coercion term. Rules ReflElimL and ReflElimR simply eliminate reflexivity uses that are composed with other coercions.

Pushing symmetry down Uses of symmetry, contrary to reflexivity, are pushed as close to the leaves as possible or eliminated, (rules SymRefl, SymAll, SymApp, SymTrans, and SymSym) only getting stuck at terms of the form $sym\,x$ and $sym(C\ \overline{\gamma})$. The idea is that by pushing uses of symmetry towards the leaves, the rest of the rules may completely ignore symmetry, except where symmetry-pushing gets stuck (variables or axiom applications).

Reducing coercions Rules RedNth, RedInstCo, and RedInstTy are the first interesting group of rules. They eliminate uses of injectivity and instantiation. Rule RedNth is concerned with the case where we wish to decompose a coercion between $H\ \overline{\varphi} \sim H\ \overline{\sigma}$, where the coercion term contains H in its head. Notice, that H is a type and may already be applied to some type arguments $\overline{\tau}^{1..\ell}$, and hence the rule has to account for selection from the first ℓ arguments, or a later argument. Rule RedInstCo deals with instantiation of a polymorphic coercion with a type. Thanks to Lemma 3.3 we may freely substitute a type inside a coercion term in the place of a type variable, since the type variable will be "guarded" by uses of $\langle\cdot\rangle$. Rule RedInstTy deals with the instantiation of a polymorphic coercion that is *just* a type.

Eta expanding and subsequent reducing Redexes of RedNth and RedInstCo or RedInstTy may not be directly visible. Consider $nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma}\ ;\ \gamma)$. The use of transitivity stands in our way for the firing of rule RedNth. To the rescue, rules EtaAllL,

ETAALLR, ETANTHL, and ETANTHR, push decomposition or instantiation through transitivity and eliminate such redexes. We call these rules "eta" because in effect we are η-expanding and immediately reducing one of the components of the transitive composition. Here is a decomposition of ETAALLL in smaller steps that involve an η-expansion (of γ_2 in the second line):

$$\begin{array}{ll} & ((\forall a{:}\eta\,.\,\gamma_1)\,;\gamma_2)@\varphi \\ \rightsquigarrow & ((\forall a{:}\eta\,.\,\gamma_1)\,;(\forall a{:}\eta\,.\,\gamma_2@a))@\varphi \\ \rightsquigarrow & (\forall a{:}\eta\,.\,\gamma_1\,;\gamma_2@a)@\varphi \quad \rightsquigarrow \quad \gamma_1[\varphi/a]\,;\gamma_2@\varphi \end{array}$$

We have merged these steps in a single rule to facilitate the proof of termination. In doing this, we do not lose any reactions, since all of the intermediate terms can also reduce to the final coercion.

There are many design possibilities for rules that look like our η-rules. For instance one may wonder why we are not always expanding terms of the form $\gamma_1\,;(\forall a{:}\eta\,.\,\gamma_2)$ to $\forall a{:}\eta\,.\,\gamma_1@a\,;\gamma_2$, whenever γ_1 is of type $\forall a{:}\eta\,.\,\tau \sim \forall a{:}\eta\,.\,\varphi$. We experimented with several variations like this, but we found that such expansions either complicated the termination argument, or did not result in smaller coercion terms. Our rules in effect perform η-expansion *only* when there is a firing reduction directly after the expansion.

Pushing transitivity down Rules PUSHAPP, PUSHALL, PUSHNTH, and PUSHINST push uses of transitivity *down* the structure of a coercion term, towards the leaves. These rules aim to reveal more redexes at the leaves, that will be reduced by the next (and final) set of rules. Notice that rules PUSHINST and PUSHNTH impose side conditions on the transitive composition $\gamma_1\,;\gamma_2$. Without these conditions, the resulting coercion may not be well-formed. Take $\gamma_1 = \forall a{:}\eta\,.\,\langle T\ a\ a\rangle$ and $\gamma_2 = \forall a{:}\eta\,.\,\langle T\ a\ \mathit{Int}\rangle$. It is certainly the case that $(\gamma_1@\mathit{Int})\,;(\gamma_2@\mathit{Int})$ is well formed. However, $\vdash^{\text{co}} \gamma_1 : \forall a{:}\eta\,.\,T\ a\ a \sim \forall a{:}\eta\,.\,T\ a\ a$ and $\vdash^{\text{co}} \gamma_2 : \forall a{:}\eta\,.\,T\ a\ \mathit{Int} \sim \forall a{:}\eta\,.\,T\ a\,;\mathit{Int}$, and hence $(\gamma_1\,;\gamma_2)@\mathit{Int}$ is not well-formed. A similar argument applies to rule PUSHNTH.

Leaf reactions When transitivity and symmetry have been pushed as low as possible, new redexes may appear, for which we introduce rules VARSYM, SYMVAR, AXSYM, SYMAX, AXSUCKR, AXSUCKL, SYMAXSUCKR, SYMAXSUCKL.

- Rules VARSYM and SYMVAR are entirely straightforward: a coercion variable (or its symmetric coercion) meets its symmetric coercion (or the variable) and the result is the identity.
- Rules AXSYM and SYMAX are more involved. Assume that the axiom $(C{:}\forall\overline{a{:}\eta}\,.\,\tau \sim \upsilon) \in \Gamma_0$, and a well-formed coercion of the form: $C\ \overline{\gamma}_1\,;\mathit{sym}(C\ \overline{\gamma}_2)$. Moreover $\Delta \vdash^{\text{co}} \overline{\gamma}_1 : \overline{\sigma}_1 \sim \overline{\varphi}_1$ and $\Delta \vdash^{\text{co}} \overline{\gamma}_2 : \overline{\sigma}_2 \sim \overline{\varphi}_2$. Then we know that $\Delta \vdash^{\text{co}} C\ \overline{\gamma}_1\,;\mathit{sym}(C\ \overline{\gamma}_2) : \tau[\overline{\sigma}_1/\overline{a}] \sim \tau[\overline{\sigma}_2/\overline{a}]$. Since the composition is well-formed, it must be the case that $\upsilon[\overline{\varphi}_1/\overline{a}] = \upsilon[\overline{\varphi}_2/\overline{a}]$. If $\overline{a} \subseteq \mathit{ftv}(\upsilon)$ then it must be $\overline{\varphi}_1 = \overline{\varphi}_2$. Hence, the pointwise composition $\overline{\gamma_1\,;\mathit{sym}\,\gamma_2}$ is well-formed and of type $\overline{\sigma}_1 \sim \overline{\sigma}_2$. Consequently, we may replace the original coercion with the lifting of τ over a substitution that maps $\overline{a}$ to $\overline{\gamma_1\,;\mathit{sym}\,\gamma_2}$: $[\overline{a} \mapsto \overline{\gamma_1\,;\mathit{sym}\,\gamma_2}]\uparrow(\tau)$. The side condition is essential for the rule to be sound. Consider the following example:

  $$C : \forall a{:}\star\,.\,F\ [a] \sim \mathit{Int} \in \Gamma_0$$

  Then $(C\ \langle\mathit{Int}\rangle)\,;\mathit{sym}(C\ \langle\mathit{Bool}\rangle)$ is well-formed and of type $F\ [\mathit{Int}] \sim F\ [\mathit{Bool}]$, but $\langle F\rangle\ (\langle\mathit{Int}\rangle\,;\mathit{sym}\langle\mathit{Bool}\rangle)$ is not well-formed! Rule SYMAX is symmetric and has a similar soundness side condition on the free variables of τ this time.
- The rest of the rules deal with the case when an axiom meets a lifted type – the reaction swallows the lifted type inside the axiom application. For instance, here is rule AXSUCKR:

  $$\frac{(C{:}\forall\overline{a{:}\eta}\,.\,\tau \sim \upsilon) \in \Gamma_0 \quad \overline{a} \subseteq \mathit{ftv}(\upsilon) \quad \mathit{good}(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\upsilon)}{\Delta \vdash (C\ \overline{\gamma}_1)\,;\delta \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \text{AXSUCKR}$$

  This time let us assume that $\Delta \vdash^{\text{co}} \overline{\gamma}_1 : \overline{\sigma}_1 \sim \overline{\varphi}_1$. Consequently $\Delta \vdash^{\text{co}} C\ \overline{\gamma}_1 : \tau[\overline{\sigma}_1/\overline{a}] \sim \upsilon[\overline{\varphi}_1/\overline{a}]$. Since $\overline{a} \subseteq \mathit{ftv}(\upsilon)$ it must be that $\Delta \vdash^{\text{co}} \overline{\gamma}_2 : \overline{\varphi}_1 \sim \overline{\varphi}_3$ for some $\overline{\varphi}_3$ and we can pointwise compose $\overline{\gamma_1;\gamma_2}$ to get coercions between $\overline{\sigma}_1 \sim \overline{\varphi}_3$. The resulting coercion $C\ \overline{\gamma_1;\gamma_2}$ is well-formed and of type $\tau[\overline{\sigma}_1/\overline{a}] \sim \upsilon[\overline{\varphi}_3/\overline{a}]$. Rules AXSUCKL, SYMAXSUCKL, and SYMAXSUCKR involve a similar reasoning.
  The side condition $\mathit{good}(\delta)$ is not restrictive in any way – it merely requires that δ contains some variable x or axiom application. If not, then δ can be converted to reflexivity:[4]
  Lemma 4.3. *If* $\vdash^{\text{co}} \delta : \sigma{\sim}\varphi$ *and* $\neg\ \mathit{good}(\delta)$, *then* $\vdash^{\text{co}} \gamma \longrightarrow^* \langle\varphi\rangle$.
  Reflexivity, when transitively composed with any other coercion, is eliminable via REFLELIML/R or and consequently the side condition is not preventing any reactions from firing. It will, however, be useful in the simplification termination proof in Section 4.3.

The purpose of rules AXSUCKL/R and SYMAXSUCKL/R is to eliminate intermediate coercions in a big transitive composition chain, to give the opportunity to an axiom to meet its symmetric version and react with rules AXSYM and SYMAX. In fact this rule is *precisely* what we need for the impressive simplification from the beginning of this section:

$$\begin{array}{lll} C_n : \forall a{:}\star \to \star\,.\,N\ a \sim \forall xy\,.\,a\ x \to a\ y & \in & \Gamma_0 \\ C_f : F\ () \sim \mathit{Maybe} & \in & \Gamma_0 \end{array}$$

Our coercion term is:

$$\mathit{nth}\,2\ (((\mathit{sym}\ C_n\ \langle\mathit{Maybe}\rangle;\langle N\rangle\ (\mathit{sym}\ C_f);C_n\ \langle F\ ()\rangle)@x_a)@y_a)$$

Its simplification is given in Figure 11. Notably, rules AXSUCKL/R and SYMAXSUCKL/R rely on axiom applications be of the form $C\ \overline{\gamma}$ instead of the simpler $C\ \overline{\tau}$ found in previous FC papers.

4.2 Coercion simplification in GHC

To assess the usefulness of coercion simplification we added it to GHC. For Haskell programs that make no use of GADTs or type families, the effect will be precisely zero, so we took measurements on two bodies of code. First, our regression suite of 151 tests for GADTs and type families; these are all very small programs. Second, the `Data.Accelerate` library that we know makes use of type families (Chakravarty et al. 2011). This library consists of 18 modules, containing 8144 lines of code.

We compiled each of these programs with and without coercion simplification, and measured the percentage reduction in size of the coercion terms with simplification enabled. This table shows the minimum, maximum, and aggregate reduction, taken over the 151 tests and 18 modules respectively. The "aggregate reduction" is obtained by combining all the programs in the group (testsuite or `Accelerate`) into one giant "program", and computing the reduction in coercion size.

	Testsuite	Accelerate
Minimum	-97%	-81%
Maximum	$+14\%$	0%
Aggregate	$\mathbf{-58\%}$	$\mathbf{-69\%}$

There is a substantial aggregate decrease of around 60% in the testsuite and 70% in `Accelerate`, with a massive 97% decrease

[4] Technically, we need a (still true) generalization of this lemma and $\longrightarrow$ to open environments, but we refrain from giving the details for lack of space.

$nth\ 2\ (((sym\ C_n\ \langle Maybe\rangle\ ;\ \langle N\rangle\ (sym\ C_f)\ ;\ C_n\ \langle F\ ()\rangle\ )@x_a)@y_a)$

(AxSuckL) $\rightsquigarrow$

$nth\ 2\ ((\ (sym\ C_n\ \langle Maybe\rangle\ ;\ C_n\ ((sym\ C_f)\ ;\langle F\ ()\rangle))\ @x_a)@y_a)$

(SymAx) $\rightsquigarrow$

$nth\ 2\ (((\forall xy.(sym\langle Maybe\rangle\ ;\ sym\ C_f\ ;\langle F\ ()\rangle)\ \langle x\rangle\ \langle\rightarrow\rangle\ (sym\langle Maybe\rangle\ ;\ sym\ C_f\ ;\langle F\ ()\rangle)\ \langle y\rangle)@x_a)@y_a)$

(ReflElimL, ReflElimR, SymRefl) $\rightsquigarrow^*$

$nth\ 2\ (((\forall xy\,.\,(sym\ C_f)\ \langle x\rangle\ \langle\rightarrow\rangle\ (sym\ C_f)\ \langle y\rangle)@x_a)@y_a)$

(RedInstCo) $\rightsquigarrow^*$ $\quad nth\ 2\ ((sym\ C_f)\ \langle x_a\rangle\ \langle\rightarrow\rangle\ (sym\ C_f)\ \langle y_a\rangle)$ $\quad$ (RedNth) $\rightsquigarrow$ $\quad (sym\ C_f)\ \langle y_a\rangle$

Figure 11: Simplification example

in special cases. These special cases should not be taken lightly: in one program the types and coercions taken together were five times bigger than the term they decorated; after simplification they were "only" twice as big. The coercion simplifier makes the compiler less vulnerable to falling off a cliff.

Only one program showed an increase in coercion size, of 14%, which turned out to be the effect of this rewrite:

$$sym(C\ ;\ D) \quad \longrightarrow \quad (sym\ D)\ ;\ (sym\ C)$$

Smaller coercion terms make the compiler faster, but the normalization algorithm itself consumes some time. However, the effect on compile time is barely measurable (less than 1%), and we do not present detailed figures.

4.3 Termination and confluence

We have demonstrated the effectiveness of the algorithm in practice, but we must also establish termination. This is important, since it would not be acceptable for a compiler to loop while simplifying a coercion, no matter what axioms are declared by users. Since the rules fire non-deterministically, and some of the rules (such as RedInstCo or AxSym) create potentially larger coercion trees, termination is not obvious.

To formalize a termination argument, we introduce several definitions in Figure 12. The *axiom polynomial* of a coercion over a distinguished variable z, $p(\cdot)$, returns a polynomial with natural number coefficients that can be compared to any other polynomial over z. The *coercion weight* of a coercion is defined as the function $w(\cdot)$ and the *symmetry weight* of a coercion is defined with the function $sw(\cdot)$ in Figure 12. Unlike the polynomial and coercion weights of a coercion, $sw(\cdot)$ does take symmetry into account. Finally, we will also use the *number of coercion applications and coercion* $\forall$*-introductions*, denoted with $intros(\cdot)$ in what follows.

Our termination argument comprises of the lexicographic left-to-right ordering of:

$$\mu(\cdot) = \langle p(\cdot), w(\cdot), intros(\cdot), sw(\cdot)\rangle$$

We will show that each of the $\rightsquigarrow$ reductions reduces this tuple. For this to be a valid termination argument for $(\longrightarrow)$ we need two more facts about *each* component measure, namely that (i) $(=)$ and $(\langle)$ are preserved under arbitrary contexts, and (ii) each component is invariant with respect to the associativity of $(;)$.

Lemma 4.4. *If* $\Delta \vdash^{co} \gamma_1 : \tau \sim \sigma$ *and* $\gamma_1 \cong \gamma_2$ *modulo associativity of* $(;)$*, then* $p(\gamma_1) = p(\gamma_2)$*,* $w(\gamma_1) = w(\gamma_2)$*,* $intros(\gamma_1) = intros(\gamma_2)$*, and* $sw(\gamma_1) = sw(\gamma_2)$*.*

Proof. This is a simple inductive argument, the only interesting case is the case for $p(\cdot)$ where the reader can calculate that $p(\gamma_1\ ;\ (\gamma_2\ ;\ \gamma_3)) = p((\gamma_1\ ;\ \gamma_2)\ ;\ \gamma_3)$ and by induction we are done. □

Lemma 4.5. *If* $\Gamma, \Delta \vdash^{co} \gamma_i : \tau \sim \sigma$ *(for* $i = 1, 2$*) and* $p(\gamma_1)\langle p(\gamma_2)$ *then* $p(\mathcal{G}[\gamma_1])\langle p(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vdash^{co} \mathcal{G}[\gamma_i] : \varphi \sim \varphi'$*. Similarly if we replace* $(\langle)$ *with* $(=)$*.*

Proof. By induction on the shape of $\mathcal{G}$. The only interesting case is the transitivity case again. Let $\mathcal{G} = \gamma\ ;\ \mathcal{G}'$. Then $p(\gamma\ ;\ \mathcal{G}'[\gamma_1]) = p(\gamma) + p(\mathcal{G}'[\gamma_1]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_1])$ whereas $p(\gamma\ ;\ \mathcal{G}'[\gamma_2]) = p(\gamma) + p(\mathcal{G}'[\gamma_2]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_2])$. Now, either $p(\gamma) = 0$, in which case we are done by induction hypothesis for $\mathcal{G}'[\gamma_1]$ and $\mathcal{G}'[\gamma_2]$, or $p(\gamma) \neq 0$ in which case again induction hypothesis gives us the result since we are multiplying $p(\mathcal{G}'[\gamma_1])$ and $p(\mathcal{G}'[\gamma_2])$ by the same polynomial. The interesting "trick" is that the polynomial for transitivity contains both the product of the components *and* their sum (since product alone is not preserved by contexts!). □

Lemma 4.6. *If* $\Gamma, \Delta \vdash^{co} \gamma_i : \tau \sim \sigma$ *and* $w(\gamma_1)\langle w(\gamma_2)$ *then* $w(\mathcal{G}[\gamma_1])\langle w(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vdash^{co} \mathcal{G}[\gamma_i] : \varphi \sim \varphi'$*. Similarly if we replace* $(\langle)$ *with* $(=)$*.*

Lemma 4.7. *If* $\Gamma, \Delta \vdash^{co} \gamma_i : \tau \sim \sigma$ *and* $intros(\gamma_1)\langle intros(\gamma_2)$ *then* $intros(\mathcal{G}[\gamma_1])\langle intros(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vdash^{co} \mathcal{G}[\gamma_i] : \varphi \sim \varphi'$*. Similarly if we replace* $(\langle)$ *with* $(=)$*.*

Lemma 4.8. *If* $\Gamma, \Delta \vdash^{co} \gamma_i : \tau \sim \sigma$*,* $w(\gamma_1) \leq w(\gamma_2)$*, and* $sw(\gamma_1)\langle sw(\gamma_2)$ *then* $sw(\mathcal{G}[\gamma_1])\langle sw(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vdash^{co} \mathcal{G}[\gamma_i] : \varphi \sim \varphi'$*.*

Proof. The only interesting case is when $\mathcal{G} = sym\,\mathcal{G}'$ and hence we have that $sw(\mathcal{G}[\gamma_1]) = sw(sym\,\mathcal{G}'[\gamma_1]) = w(\mathcal{G}'[\gamma_1]) + sw(\mathcal{G}'[\gamma_1])$. Similarly $sw(\mathcal{G}[\gamma_2]) = w(\mathcal{G}'[\gamma_2]) + sw(\mathcal{G}'[\gamma_2])$. By the precondition for the weights and induction hypothesis we are done. The precondition on the weights is not restrictive at all, since $w(\cdot)$ has higher precedence than $sw(\cdot)$ inside $\mu(\cdot)$. □

The conclusion is the following theorem.

Theorem 4.9. *If* $\gamma \cong \mathcal{G}[\gamma_1]$ *modulo associativity and* $\Delta \vdash^{co} \gamma_1 : \sigma \sim \varphi$*, and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *such that* $\mu(\gamma_2)\langle\mu(\gamma_1)$*, it is the case that* $\mu(\mathcal{G}[\gamma_2])\langle\mu(\gamma)$*.*

Corollary 4.10. $(\longrightarrow)$ *terminates on well-formed coercions if each of the* $\rightsquigarrow$ *transitions reduces* $\mu(\cdot)$*.*

We finally show that indeed each of the $\rightsquigarrow$ steps reduces $\mu(\cdot)$.

Theorem 4.11 (Termination)**.** *If* $\Delta \vdash^{co} \gamma_1 : \sigma \sim \varphi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\mu(\gamma_2)\langle\mu(\gamma_1)$*.*

Proof. It is easy to see that the reflexivity rules, the symmetry rules, the reduction rules, and the η-rules preserve or reduce the polynomial component $p(\cdot)$. The same is true for the push rules but the proof is slightly more interesting. Let us consider PushApp, and let us write p_i for $p(\gamma_i)$. We have that $p((\gamma_1\ \gamma_2)\ ;\ (\gamma_3\ \gamma_4)) = p_1 + p_2 + p_3 + p_4 + p_1p_3 + p_2p_3 + p_1p_4 + p_2p_4$. On the other hand $p((\gamma_1\ ;\ \gamma_3)\ \ (\gamma_2\ ;\ \gamma_4)) = p_1 + p_3 + p_1p_3 + p_2 + p_4 + p_2p_4$ which is a smaller or equal polynomial than the left-hand side polynomial. Rule PushAll is easier. Rules PushInst and PushNth have exactly the same polynomials on the left-hand and

Axiom polynomial			Coercion weight			Symmetry weight		
$p(sym\,\gamma)$	$=$	$p(\gamma)$	$w(sym\,\gamma)$	$=$	$w(\gamma)$	$sw(sym\,\gamma)$	$=$	$w(\gamma) + sw(\gamma)$
$p(C\ \overline{\gamma})$	$=$	$z \cdot \Sigma p(\gamma_i) + z + 1$	$w(C\ \overline{\gamma})$	$=$	$\Sigma w(\gamma_i) + 1$	$sw(C\ \overline{\gamma})$	$=$	$\Sigma sw(\gamma_i)$
$p(x)$	$=$	1	$w(x)$	$=$	1	$sw(x)$	$=$	0
$p(\gamma_1\ ;\gamma_2)$	$=$	$p(\gamma_1) + p(\gamma_2) + p(\gamma_1) \cdot p(\gamma_2)$	$w(\gamma_1\ ;\gamma_2)$	$=$	$1 + w(\gamma_1) + w(\gamma_2)$	$sw(\gamma_1\ ;\gamma_2)$	$=$	$sw(\gamma_1) + sw(\gamma_2)$
$p(\langle\varphi\rangle)$	$=$	0	$w(\langle\varphi\rangle)$	$=$	1	$sw(\langle\varphi\rangle)$	$=$	0
$p(nth\ k\ \gamma)$	$=$	$p(\gamma)$	$w(nth\ k\ \gamma)$	$=$	$1 + w(\gamma)$	$sw(nth\ k\ \gamma)$	$=$	$sw(\gamma)$
$p(\gamma @ \varphi)$	$=$	$p(\gamma)$	$w(\gamma @ \varphi)$	$=$	$1 + w(\gamma)$	$sw(\gamma @ \varphi)$	$=$	$sw(\gamma)$
$p(\gamma_1\ \gamma_2)$	$=$	$p(\gamma_1) + p(\gamma_2)$	$w(\gamma_1\ \gamma_2)$	$=$	$1 + w(\gamma_1) + w(\gamma_2)$	$sw(\gamma_1\ \gamma_2)$	$=$	$sw(\gamma_1) + sw(\gamma_2)$
$p(\forall a{:}\eta\,.\,\gamma)$	$=$	$p(\gamma)$	$w(\forall a{:}\eta\,.\,\gamma)$	$=$	$1 + w(\gamma)$	$sw(\forall a{:}\eta\,.\,\gamma)$	$=$	$sw(\gamma)$

Figure 12: Metrics on coercion terms

the right-hand side so they are ok. Rules VARSYM and SYMVAR reduce $p(\cdot)$. The interesting bit is with rules AXSYM, SYMAX, and AXSUCKR/L and SYMAXSUCKR/L. We will only show the cases for AXSYM and AXSUCKR as the rest of the rules involve very similar calculations:

- Case SYMAX. We will use the notational convention $\overline{p}_1$ for $p(\overline{\gamma}_1)$ (a vector of polynomials) and similarly $\overline{p}_2$ for $p(\overline{\gamma}_2)$. Then the left-hand side polynomial is:

  $$\begin{aligned}&(z\Sigma\overline{p}_1{+}z{+}1) + (z\Sigma\overline{p}_2{+}z{+}1){+}\\ &\qquad (z\Sigma\overline{p}_1{+}z{+}1)\cdot(z\Sigma\overline{p}_2{+}z{+}1) =\\ &(z^2{+}2z)\Sigma\overline{p}_1 + (z^2{+}2z)\Sigma\overline{p}_2 + z^2\Sigma\overline{p}_1\Sigma\overline{p}_2 + (z^2{+}4z{+}3)\end{aligned}$$

  For the right-hand side polynomial we know that each γ_{1i} ; $sym\,\gamma_{2i}$ will have weight $p_{1i} + p_{2i} + p_{1i}p_{2i}$ and it cannot be repeated inside the lifted type more than a finite number of times (bounded by the maximum number of occurrences of a type variable from $\overline{a}$ in type τ), call it k. Hence the right-hand side polynomial is smaller or equal to:

  $$k\Sigma\overline{p}_1 + k\Sigma\overline{p}_2 + k\Sigma(p_{1i}p_{2i}) \leq k\Sigma\overline{p}_1 + k\Sigma\overline{p}_2 + k\Sigma\overline{p}_1\Sigma\overline{p}_2$$

  But that polynomial is strictly smaller than the left-hand side polynomial, hence we are done.
- Case AXSUCKR. In this case the left-hand side polynomial is going to be greater or equal to (because of reflexivity inside δ and because some of the $\overline{a}$ variables may appear more than once inside υ it is not exactly equal to) the following:

  $$\begin{aligned}&(z\Sigma\overline{p}_1 + z + 1) + \Sigma\overline{p}_2 + (z\Sigma\overline{p}_1 + z + 1)\Sigma\overline{p}2 =\\ &\qquad \Sigma\overline{p}_1\Sigma\overline{p}_2 + z\Sigma\overline{p}_1 + z\Sigma\overline{p}_2 + 2\Sigma\overline{p}_2 + z + 1\end{aligned}$$

  On the other hand, the right-hand side polynomial is:

  $$\begin{aligned}&z\Sigma(p_{1i} + p_{2i} + p_{1i}p_{2i}) + z + 1 \leq\\ &\qquad z\Sigma\overline{p}_1 + z\Sigma\overline{p}_2 + z\Sigma\overline{p}_1\Sigma\overline{p}_2 + z + 1\end{aligned}$$

  We observe that there is a difference of $2\Sigma\overline{p}_2$, but we know that δ satisfies $good(\delta)$, and consequently there must exist some variable or axiom application inside one of the $\overline{\gamma}_2$. Therefore, $\Sigma\overline{p}_2$ is *non-zero* and the case is finished.

It is the arbitrary copying of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$ in rules AXSYM and SYMAX that prevents simpler measures that only involve summation of coercions for axioms or transitivity. Other reasonable measures such as the height of transitivity uses from the leaves would not be preserved from contexts, due to AXSYM again.

So far we've shown that all rules but the axiom rules preserve the polynomials, and the axiom rules reduce them. We next show that in the remaining rules, some other component reduces, lexicographically. Reflexivity rules reduce $w(\cdot)$. Symmetry rules preserve $w(\cdot)$ and $intros(\cdot)$ but reduce $sw(\cdot)$. Reduction rules and η-rules reduce $w(\cdot)$. Rules PUSHAPP and PUSHALL preserve or reduce $w(\cdot)$ but certainly reduce $intros(\cdot)$. Rules PUSHINST and PUSHNTH reduce $w(\cdot)$. □

We conclude that $(\longrightarrow)$ terminates.

Confluence Due to the arbitrary types of axioms and coercion variables in the context, we do not expect confluence to be true. Here is a short example that demonstrates the lack of confluence; assume we have the following in our context:

$$\begin{aligned}C_1 &: \forall a{:}\star \to \star\,.\,F\ a \sim a \quad \in \quad \Gamma_0\\ C_2 &: \forall a{:}\star \to \star\,.\,G\ a \sim a \quad \in \quad \Gamma_0\end{aligned}$$

Consider the coercion:

$$(C_1\ \langle\sigma\rangle)\ ;\ sym(C_2\ \langle\sigma\rangle)$$

of type $F\ \ \sigma\ \sim\ G\ \ \sigma$. In one reduction possibility, using rule AXSUCKR, we may get $C_1\ (sym(C_2\ \langle\sigma\rangle))$. In another possibility, using SYMAXSUCKL, we may get $sym(C_2\ \ (sym(C_1\ \ \langle\sigma\rangle))))$. Although the two normal forms are different, it is unclear if one of them is "better" than the other.

Despite this drawback, confluence or syntactic characterization of normal forms is, for our purposes, of secondary importance (if possible at all for open coercions in such an under-constrained problem!), since we never reduce coercions for the purpose of comparing their normal forms.

5. Related and future work

5.1 Coercion erasure

There is a substantial volume of related work on proof erasure in the context of dependent type theory. Our method for sound, runtime, but zero-cost equality proof terms lies in the middle ground between two other general methodologies.

Type-based erasure On the one hand, Coq (The Coq Team) uses a *type-based* erasure process by introducing a special universe for propositions, *Prop*. Terms whose type lives in *Prop* are erased even when they are applications of functions (lemmas) to computational terms. This is sound since in Coq the computation language is also strongly normalizing. As we have seen, this is not sound in FC.

Irrelevance-based erasure On the other hand, *irrelevance-based* erasure is another methodology proposed in the context of pure type systems and type theory. In the context of Epigram, Brady et al. (2003) present an erasure technique where term-level indices of inductive types can be erased even when they are deconstructed inside the body of a function, since values of the indexed inductive datatype will be simultaneously deconstructed and hence the indices are irrelevant for the computation. In the Agda language (Norell 2007) there exist plans to adopt a similar irrelevance-based erasure strategy. Other related work (Abel 2011; Mishra-Linger and Sheard 2008) proposes erasure in the context of PTSs guided with lightweight programmer annotations.

Finally, our approach of separating the "computational part" of a proof, which always has to run before we get to a zero-cost "logical part" is reminiscent of the separation that A-normal forms introduce in refinement type systems, for instance (Bengtson et al. 2008). It is interesting future work to determine whether our treatment of coercions is also applicable to types and hopefully paves the way towards full-spectrum dependent types.

5.2 Coercion simplification

To our knowledge, most literature on coercions is not concerned with coercion simplification but rather with inferring the placement of coercions in source-level programs. Some recent examples are (Luo 2008) and (Swamy et al. 2009).

One of the few comprehensive studies of coercions *and their normalization* is that of Henglein (1994), motivated by coercion placement in a language with *type dynamic*. His coercion language differs to ours in that (i) coercions there are not symmetric, (ii) do not involve polymorphic axiom schemes and (iii) may have computational significance. Unlike us, Henglein is concerned with characterizations of minimal coercions and confluence, fixes an equational theory of coercions, and presents a normalization algorithm for that equational theory. In our case, in the absence of a denotational semantics for System FC and its coercions, such an axiomatization would be no more ad-hoc than the algorithm and hence not particularly useful: for instance we could consider adding type-directed equations like $\Delta \vdash \gamma \rightsquigarrow \langle \tau \rangle$ when $\Delta \stackrel{\mathsf{co}}{\vdash} \gamma : \tau \sim \tau$, or other equations that only hold in consistent or confluent axiom sets. It is certainly an interesting direction for future work to determine whether there even exists a maximal syntactic axiomatization of equalities between coercions with respect to some denotational semantics of System FC.

More recently, Rémy and Yakobowski (2010) present xMLF, a calculus with coercions that capture *instantiation* instead of equality, that serves as target for the MLF language. Although they are not directly concerned with normalization as part of an intermediate language simplifier, their translation of the graph-based instantiation witnesses does produce xMLF normal proofs.

Finally, another future work direction would be to determine whether we can encode coercions as λ-terms, and derive coercion simplification by normalization in some suitable λ-calculus.

Acknowledgments Thanks to Tom Schrijvers for early discussions and for contributing a first implementation.

References

Andreas Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures, 14th International Conference, FOSSACS 2011, Saarbrucken, Germany*, 2011. To appear.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3182-3.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3.

James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

Fritz Henglein. Dynamic typing: syntax and proof theory. *Sci. Comput. Program.*, 22:197–230, June 1994. ISSN 0167-6423.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. *Fun with type functions*. Springer, 2010.

Daniel R. Licata and Robert Harper. 2-dimensional Directed Type Theory. Draft, 2011.

Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin / Heidelberg, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Dominic Orchard and Tom Schrijvers. Haskell type constraints unleashed. In Matthias Blume, Naoki Kobayashi, and Germn Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin / Heidelberg, 2010.

Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional programming language. In *FPCA91: Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, NY, August 1991. ACM Press.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-309-3.

Didier Rémy and Boris Yakobowski. A Church-style intermediate language for MLF. In Matthias Blume, Naoki Kobayashi, and German Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin / Heidelberg, 2010.

Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*, pages 106–124, July 2004.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 329–340, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

The Coq Team. *Coq*. URL `http://coq.inria.fr`.

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.