Equality proofs and deferred type errors
A compiler pearl

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

José Pedro Magalhães
Utrecht University
jpm@cs.uu.nl

Abstract

The Glasgow Haskell Compiler is an optimizing compiler that expresses and manipulates first-class equality proofs in its intermediate language. We describe a simple, elegant technique that exploits these equality proofs to support *deferred type errors*. The technique requires us to treat equality proofs as possibly-divergent terms; we show how to do so without losing either soundness or the zero-overhead cost model that the programmer expects.

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 Type equalities, Deferred type errors, System FC

1. Introduction

In a compiler, 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 language provides a simple and powerful consistency check on the earlier stages of type inference and other optimizing program transformations. The Glasgow Haskell Compiler (GHC) has just such an intermediate language. This intermediate language has evolved in the last few years 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); and from System FC to System $F_C^{\uparrow}$, a calculus now fully equipped with *kind polymorphism* and *datatype promotion* (Yorgey et al. 2012).

The principal difference between System F and System $F_C^{\uparrow}$ is that, together with type information, System $F_C^{\uparrow}$ carries *equality proofs*: evidence that type equality constraints are satisfied. Such proofs are generated during the type inference process and are useful for type checking System $F_C^{\uparrow}$ programs. However, once type checking of the $F_C^{\uparrow}$ program is done, proofs – very much like types – can be completely (and statically) erased, so that they induce no runtime execution or allocation overhead.

Proof assistants and dependently typed languages (Bove et al. 2009; Norell 2007; The Coq Team) adopt a similar design with statically erasable proofs, including ones that go beyond equality to more complex program properties. However, there is one important difference: in proof assistants the proof language *is* the computation language, always a side-effect free and terminating language that guarantees logical consistency of the proofs. On the other hand, in System $F_C^{\uparrow}$ the computation language includes partial functions and divergent terms. To ensure logical consistency, $F_C^{\uparrow}$ keeps the equality proof language as a syntactically separate, consistent-by-construction set of equality proof combinators.

In this paper we investigate the opportunities and challenges of blurring the rigid proof/computation boundary, without threatening soundness, by allowing "proof-like" first-class values to be returned from *ordinary* (even divergent or partial) computation terms. We make the following contributions:

- The proofs-as-values approach opens up an entirely new possibility, that of *deferring type errors to runtime*. A common objection to static type systems is that the programmer wants to be able to run a program even though it may contain some type errors; after all, the execution might not encounter the error. Recent related work (Bayne et al. 2011) makes a convincing case that during prototyping or software evolution programmers wish to focus on getting *part* of their code right, without first having to get *all* of it type-correct. Deferring type errors seems to be just the right mechanism to achieve this. Our new approach gives a principled (and simple compared to Bayne et al. (2011)) way in which such erroneous programs can be run with complete type safety (Sections 3 and 5).
- The key to the almost effortless shift to proofs-as-values is based on a simple observation: System $F_C^{\uparrow}$, with the recent addition of kind polymorphism (Yorgey et al. 2012), already allows us to *define within the system* an ordinary first-class type for type equality (Section 4). As such, we can have ordinary values of that type, that are passed to or returned from arbitrary (even partial or divergent) terms. Moreover, deferring type errors aside, there are other compelling advantages of proofs-as-values in an evidence-passing compiler, as we outline in Section 6.
- Programmers think of types as static objects, with zero runtime overhead, and they expect the same of proofs about types. Treating type equality proofs as values seriously undermines this expectation. In Section 7 we address this challenge and show how the optimizer of GHC, with no changes whatsoever, can already eliminate the cost of equality proofs – except in corner cases where it would be wrong to do so.

Everything we describe is fully implemented. As far as we know, GHC is the first, and only, widely-used optimizing compiler that manipulates first-class proofs. We describe this paper as a "pearl" because it shows a simple and elegant way in which the apparently-esoteric notion of a "proof object" can be deployed to solve a very practical problem.

2. The opportunity: deferring type errors

Suppose you type this Haskell term into the interactive read-eval-print prompt in GHCi:

```
ghci> fst (True, 'a' && False)
```

This term does not "go wrong" when evaluated: you might expect to just get back the result *True* from projecting the first component of the pair. But in a statically typed language like Haskell you get the type error:

```
Couldn't match 'Bool' with 'Char'
In the first argument of '(&&)', namely 'a'
```

This behaviour is fine for programs that are (allegedly) finished, but some programmers would much prefer the term to evaluate to *True* when doing exploratory programming. After all, if the error is in a bit of the program that is not executed, it is doing no harm! In particular, when refactoring a large program it is often useful to be able to run parts of the completed program, but type errors prevent that. What we want is to defer type errors until they matter. We have more to say about motivation and related work in Section 8.

As we shall see, System $F_C^{\uparrow}$ allows us to offer precisely this behaviour, *without giving up type safety*. Here is an interactive session with `ghci -fdefer-type-errors`:

```
ghci> let foo = (True, 'a' && False)
Warning: Couldn't match 'Bool' with 'Char'
ghci> :type foo
(Bool, Bool)
ghci> fst foo
True
ghci> snd foo
Runtime error: Couldn't match 'Bool' with 'Char'
```

Notice that:

- The definition of *foo* produced a warning (rather than an error), but succeeds in producing an executable binding for *foo*.
- Since type checking of *foo* succeeded it has a type, which can be queried with `:type` to display its type, $(Bool, Bool)$.
- The term *fst foo* typechecks fine, *and also runs fine*, returning *True*.
- The term *snd foo* also typechecks fine, and runs; however the evaluation aborts with a runtime error giving *exactly the same error* as the original warning.

That is, the error message is produced lazily, at runtime, when and only when the requirement for *Char* and *Bool* to be the same type is encountered.

2.1 How deferring type errors works, informally

GHC's type inference algorithm works in two stages: first we *generate* type constraints, and then we *solve* them (Vytiniotis et al. 2011). In addition, inference elaborates the Haskell source term to an explicitly typed $F_C^{\uparrow}$ term, that includes the types and proofs ("evidence" in GHC jargon) computed by the constraint solver.

In the previous example, during type inference for the sub-term `'a'` && *False* we generate a type equality constraint, written $Char \sim Bool$. Usually the constraint solver would immediately reject such a constraint as insoluble, but with `-fdefer-type-errors` we take a different course: we generate "evidence" for $Char \sim Bool$, but ensure that if the (bogus) evidence is ever evaluated it brings the program to a graceful halt. More concretely, here is the $F_C^{\uparrow}$ term that we generate for *foo*:

$$\begin{aligned}&foo = \mathbf{let}\ (c : Char \sim Bool) = error\ \texttt{"Couldn't..."}\\ &\qquad\ \ \mathbf{in}\ (True, (cast\ \texttt{'a'}\ c)\ \&\&\ False)\end{aligned}$$

The elaborated *foo* contains a lazy binding of an evidence variable *c* of type $Char \sim Bool$ to a call to *error*. The latter is a built-in Haskell constant, of type $\forall a\ .\ String \to a$, that prints its argument string and brings execution to a halt.

When we evaluate *fst foo* the result is *True*; but if we evaluate *snd foo*, we must evaluate the result of (&&), which in turn evaluates its first argument, *cast* `'a'` *c*. The cast forces evaluation of *c*, and hence triggers the runtime error. Note that the exact placement of coercions, and thus which errors get deferred, depends on the internals of the type inference process; we discuss this in more detail in Section 5.4.

There is something puzzling about binding variable *c* with the type $Char \sim Bool$. The evidence variable *c* is supposed to be bound to a proof witnessing that *Char* and *Bool* are equal types, but is nevertheless bound to just a *term*, and in fact a crashing term, namely *error*! How can we then ensure soundness, and how can we get statically erasable proofs? It turns out that the type $Char \sim Bool$ is almost but *not quite* the type of a proof object. To explain how this works, we move on to present some more details on GHC's typed intermediate language, System $F_C^{\uparrow}$.

3. The $F_C^{\uparrow}$ language

System $F_C^{\uparrow}$ is a polymorphic and explicitly typed language, whose syntax is given in Figure 1. Our presentation closely follows the most recent work on $F_C^{\uparrow}$ Yorgey et al. (2012), and we will not repeat operational semantics and type soundness results; instead, we refer the reader to Yorgey et al. (2012) for the details.

A quick glance at Figure 1 will confirm that the term language *e* is mostly conventional, explicitly-typed, lambda calculus, with let-bindings, literals (*l*), data constructors (*K*), and case expressions. In addition, the language includes type *and kind* polymorphism: type $(\Lambda a{:}\eta\ .\ e)$ and kind $(\Lambda\chi\ .\ e)$ abstractions, and type $(e\ \varphi)$ and kind $(e\ \kappa)$ applications, respectively. Some motivation for kind abstractions and applications comes from previous work but, as we shall see in Section 6.2, kind polymorphism will play a key role here as well.

The distinctive feature of $F_C^{\uparrow}$ is the use of *coercions*, γ. A coercion γ of type $\tau \sim_{\#} \varphi$ is nothing more than a *proof of type equality* between the types φ and τ. Contrary to the notation used in Section 2.1 and in previous presentations of System $F_C^{\uparrow}$ notice that we use symbol $\sim_{\#}$ instead of $\sim$ for coercion types, and $Constraint_{\#}$ rather than *Constraint* for their kind – this is for a good reason that will become evident in Section 4.2.

The term $(e \triangleright \gamma)$ is a *cast* that converts a term *e* of type τ to one of type φ, when $\gamma : \tau \sim_{\#} \varphi$. Once again, this is deliberately different than the *cast* term that appeared in Section 2.1, as we discuss in Section 4.2. The only other place where a coercion γ may appear in the term language is in an application $(e\ \gamma)$, so coercions are not first-class values. Dually, one can abstract over such coercions with a coercion abstraction $\lambda(c{:}\tau \sim_{\#} \varphi)\ .\ e$.

$$
\begin{array}{llll}
\text{Terms} & & & \\
e,u & ::= & x \mid l \mid \lambda x{:}\sigma\,.\,e \mid e\;u & \\
 & \mid & \Lambda a{:}\eta\,.\,e \mid e\;\varphi \mid \Lambda\chi\,.\,e \mid e\;\kappa & \text{Type/kind polymorphism} \\
 & \mid & \lambda c{:}\tau\,.\,e \mid e\;\gamma & \text{Coercion abs/app} \\
 & \mid & K \mid \textbf{case}\;e\;\textbf{of}\;\overline{p \to u} & \\
 & \mid & \textbf{let}\;x{:}\tau = e\;\textbf{in}\;u & \\
 & \mid & e \rhd \gamma & \text{Cast} \\
p & ::= & K\;\overline{c{:}\tau}\;\overline{x{:}\tau} & \text{Patterns} \\
 & & & \\
\text{Types} & & & \\
\varphi,\sigma,\tau,\upsilon & ::= & a & \text{Variables} \\
 & \mid & H & \text{Constants} \\
 & \mid & F & \text{Type functions} \\
 & \mid & \varphi_1\;\varphi_2 & \text{Application} \\
 & \mid & \varphi\;\kappa & \text{Kind application} \\
 & \mid & \forall a{:}\eta\,.\,\varphi & \text{Polymorphic types} \\
 & \mid & \forall\chi\,.\,\tau & \text{Kind-polymorphic types} \\
\text{Type constants} & & & \\
H & ::= & T & \text{Datatypes} \\
 & \mid & (\to) & \text{Arrow} \\
 & \mid & (\sim_{\#}) & \text{Primitive equality type} \\
\text{Kinds} & & & \\
\kappa,\eta & ::= & \chi \mid \star \mid \kappa \to \kappa & \\
 & \mid & \forall\chi\,.\,\kappa & \text{Polymorphic kinds} \\
 & \mid & \mathit{Constraint}_{\#} & \text{Kind of static proofs} \\
 & & & \\
\text{Coercion values} & & & \\
\gamma,\delta & ::= & c & \text{Variables} \\
 & \mid & \gamma_1\;\gamma_2 & \text{Application} \\
 & \mid & \langle\varphi\rangle & \text{Reflexivity} \\
 & \mid & \gamma_1 \,;\, \gamma_2 & \text{Transitivity} \\
 & \mid & \mathit{sym}\,\gamma & \text{Symmetry} \\
 & \mid & \ldots & \text{Other coercion forms} \\
 & & & \\
\text{Environments} & & & \\
\Gamma,\Delta & ::= & \cdot \mid \Gamma, \mathit{bnd} & \\
\mathit{bnd} & ::= & \chi & \text{Kind variable} \\
 & \mid & a : \eta & \text{Type variable} \\
 & \mid & c : \sigma \sim_{\#} \varphi & \text{Coercion variable} \\
 & \mid & x : \sigma & \text{Term variable} \\
 & \mid & T : \forall\overline{\chi}\,.\,\overline{\kappa} \to \star & \text{Data type} \\
 & \mid & K : \forall\overline{\chi}\;(\overline{a{:}\eta_a})\,.\,\overline{\tau} \to T\;\overline{\chi}\;\overline{a} & \text{Data constructor} \\
 & \mid & \ldots & \text{Type families etc.} \\
\text{Notation} & & & \\
T\;\overline{\kappa}\;\overline{\tau} & \equiv & T\;\kappa_1 \ldots \kappa_m\;\tau_1 \ldots \tau_n & \\
\overline{\alpha} \to \alpha & \equiv & \alpha_1 \to \ldots \to \alpha_n \to \alpha & \\
 & & \text{for } \alpha \text{ either } \kappa \text{ or } \tau & \\
 & & & \\
\Gamma_0 & \equiv & \text{initial (closed) environment} & \\
 & \ni & (\sim_{\#}) : \forall\chi\,.\,\chi \to \chi \to \mathit{Constraint}_{\#} &
\end{array}
$$

Figure 1: Syntax of System FC (excerpt)

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

$$
\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash^{\mathsf{tm}} x : \tau}\;\textsc{EVar}
\qquad
\frac{(K{:}\sigma) \in \Gamma_0}{\Gamma \vdash^{\mathsf{tm}} K : \sigma}\;\textsc{ECon}
$$

$$
\frac{\begin{array}{c}\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \\ \Gamma \vdash^{\mathsf{ty}} \sigma : \star\end{array}}{\Gamma \vdash^{\mathsf{tm}} \lambda x{:}\sigma\,.\,e : \sigma \to \tau}\;\textsc{EAbs}
\qquad
\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{tm}} e : \sigma \to \tau \\ \Gamma \vdash^{\mathsf{tm}} u : \sigma\end{array}}{\Gamma \vdash^{\mathsf{tm}} e\;u : \tau}\;\textsc{EApp}
$$

$$
\frac{\begin{array}{c}\Gamma, (c{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \\ \Gamma \vdash^{\mathsf{ty}} \sigma : \mathit{Constraint}_{\#}\end{array}}{\Gamma \vdash^{\mathsf{tm}} \lambda c{:}\sigma\,.\,e : \sigma \to \tau}\;\textsc{ECAbs}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{tm}} e : (\sigma_1 \sim_{\#} \sigma_2) \to \tau \\ \Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim_{\#} \sigma_2\end{array}}{\Gamma \vdash^{\mathsf{tm}} e\;\gamma : \tau}\;\textsc{ECApp}
$$

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

$$
\frac{\Gamma, \chi \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \Lambda\chi\,.\,e : \forall\chi\,.\,\tau}\;\textsc{EKAbs}
\qquad
\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{tm}} e : \forall\chi\,.\,\tau \\ \Gamma \vdash^{\mathsf{k}} \kappa\end{array}}{\Gamma \vdash^{\mathsf{tm}} e\;\kappa : \tau[\kappa/\chi]}\;\textsc{EKApp}
$$

$$
\frac{\begin{array}{c}\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} u : \sigma \\ \Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash^{\mathsf{tm}} \textbf{let}\;x{:}\sigma = u\;\textbf{in}\;e : \tau}\;\textsc{ELet}
\qquad
\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{tm}} e : \tau \\ \Gamma \vdash^{\mathsf{co}} \gamma : \tau \sim_{\#} \varphi\end{array}}{\Gamma \vdash^{\mathsf{tm}} e \rhd \gamma : \varphi}\;\textsc{ECast}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{tm}} e : T\;\overline{\kappa}\;\overline{\sigma} \\ \text{For each branch } K\;\overline{x{:}\tau} \to u \\ \quad (K{:}\forall\overline{\chi}\;(\overline{a{:}\eta_a})\,.\,\overline{\sigma_1 \sim_{\#} \sigma_2} \to \overline{\tau} \to T\;\overline{\chi}\;\overline{a}) \in \Gamma_0 \\ \quad \varphi_i = \tau_i[\overline{\kappa}/\overline{\chi}][\overline{\sigma}/\overline{a}] \\ \quad \varphi_{1i} = \sigma_{1i}[\overline{\kappa}/\overline{\chi}][\overline{\sigma}/\overline{a}] \\ \quad \varphi_{2i} = \sigma_{2i}[\overline{\kappa}/\overline{\chi}][\overline{\sigma}/\overline{a}] \quad \Gamma, \overline{c{:}\varphi_1 \sim \varphi_2}\;\overline{x{:}\varphi} \vdash^{\mathsf{tm}} u : \sigma\end{array}}{\Gamma \vdash^{\mathsf{tm}} \textbf{case}\;e\;\textbf{of}\;\overline{K\;(\overline{c{:}\sigma_1 \sim_{\#} \sigma_2})\;(\overline{x{:}\tau}) \to u} : \sigma}\;\textsc{ECase}
$$

Figure 2: Well-formed terms

The syntax of coercions themselves (γ in Figure 1) includes coercion variables, constructors for reflexivity, transitivity, and symmetry, as well as other constructors (such as lifting type equalities over data constructors) that we do not need to discuss in this paper.

The well-formedness judgement for terms appears in Figure 2 and is mostly conventional. In particular, the rules for coercion abstraction and application (ECAbs and ECApp) mirror those for terms (EAbs and EApp). The rule for case expressions (ECase) is also standard but notice that it allows us to bind coercion variables, as well as term variables, in a pattern.

3.1 Types, kinds, and kind polymorphism

The type language of System $F_C^{\uparrow}$ includes variables and constants, type and kind application, as well as the type of type-polymorphic values ($\forall a{:}\eta\,.\,\varphi$) and the type of kind-polymorphic values ($\forall\chi\,.\,\tau$). The type constants H include data constructors (T), and the function constructor ($\to$) as well as the equality constructor ($\sim_{\#}$). The well-formedness judgement for types appears in Figure 3.

What should the kind of $\sim_{\#}$ be? We mentioned previously that we would like to classify any type $\tau \sim_{\#} \sigma$ as having kind $\mathit{Constraint}_{\#}$, but the kind of τ and σ can be *any* kind whatsoever. This indicates that $\sim_{\#}$ should be given the *polymorphic* kind:

$$\forall\chi\,.\,\chi \to \chi \to \mathit{Constraint}_{\#}$$

This kind, made valid because the syntax of kinds κ includes kind polymorphism, is recorded in the initial environment Γ_0 (bottom of Figure 1). Well-formedness of kinds ($\Gamma \vdash^{\mathsf{k}} \kappa$), for this presentation, amounts to well-scoping, so we omit the details from Figure 3. As a convention, we write $\tau \sim_{\#} \varphi$ to mean $(\sim_{\#})\;\kappa\;\tau\;\varphi$ in the rest of this paper, where the kind κ of τ and φ is clear from the context.

Finally, notice that well-formed arrow types[1] are allowed to accept an argument which is either $\mathit{Constraint}_{\#}$ or $\star$, to account for coercion or term abstraction, but may only return $\star$, hence disallowing any functions to return a value of type $\tau \sim_{\#} \varphi$. However, as we will

[1] For simplicity of presentation we do not include a binding for $(\to)$ in the initial environment Γ_0, hence only allowing fully applied arrow types, unlike Haskell.

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

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

$$\frac{\Gamma \vdash^{\mathsf{ty}} \tau_1 : \kappa_1 \quad \Gamma \vdash^{\mathsf{ty}} \tau_2 : \star \quad \kappa_1 \in \{\star, \mathit{Constraint}_{\#}\}}{\Gamma \vdash^{\mathsf{ty}} \tau_1 \to \tau_2 : \star}\ \textsc{TArr}$$

$$\frac{\Gamma \vdash^{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash^{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash^{\mathsf{ty}} \tau_1\ \tau_2 : \kappa_2}\ \textsc{TApp} \qquad \frac{\Gamma \vdash^{\mathsf{ty}} \tau : \forall \chi . \kappa \quad \Gamma \vdash^{\mathsf{k}} \eta}{\Gamma \vdash^{\mathsf{ty}} \tau\ \eta : \kappa[\eta/\chi]}\ \textsc{TKApp}$$

$$\frac{\Gamma,(a{:}\eta) \vdash^{\mathsf{ty}} \tau : \star \quad \Gamma \vdash^{\mathsf{k}} \eta}{\Gamma \vdash^{\mathsf{ty}} \forall a{:}\eta . \tau : \star}\ \textsc{TAll} \qquad \frac{\Gamma, \chi \vdash^{\mathsf{ty}} \tau : \star}{\Gamma \vdash^{\mathsf{ty}} \forall \chi . \tau : \star}\ \textsc{TKAll}$$

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

$$\ldots$$

$$\boxed{\Gamma \vdash^{\mathsf{k}} \kappa}$$

$$\ldots$$

Figure 3: Well-formed types and coercions

see in Section 4, a function can well return terms *that contain* such coercions.

3.2 $\mathrm{F}_\mathrm{C}^{\uparrow}$ datatypes with coercions

In $\mathrm{F}_\mathrm{C}^{\uparrow}$, coercions can appear as arguments to data constructors, a feature that is particularly useful for representing generalized algebraic datatypes (GADTs) (Peyton Jones et al. 2006). Consider this *source* Haskell program which defines and uses a GADT:

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

In $\mathrm{F}_\mathrm{C}^{\uparrow}$, we regard the GADT data constructor T_1 as having the type:

$$T_1 : \forall a . (a \sim_{\#} \mathit{Int}) \to \mathit{Int} \to T\ a$$

So T_1 takes three arguments: a type argument to instantiate a, a coercion witnessing the equality between a and *Int*, and a value of type *Int*. Here is the $\mathrm{F}_\mathrm{C}^{\uparrow}$ version of *main*:

$$\mathit{main} = f\ \mathit{Int}\ (T_1\ \mathit{Int}\ \langle \mathit{Int} \rangle\ 4)$$

The coercion argument has kind $\mathit{Int} \sim_{\#} \mathit{Int}$, for which the evidence is just $\langle \mathit{Int} \rangle$ (reflexivity). Similarly, pattern-matching on T_1 binds two variables: a coercion variable, and a term variable. Here is the $\mathrm{F}_\mathrm{C}^{\uparrow}$ elaboration of function f:

```
f = Λ(a : ⋆) . λ(x : T a) .
    case x of
      T1 (c : a ∼# Int) (n : Int)
          → (Cons (n+1) Nil) ▷ sym [c]
      T2 v → Nil
```

The cast converts the type of the result from $[\mathit{Int}]$ to $[a]$[2]. The coercion $\mathit{sym}\ [c]$ is evidence for (or a proof of) the equality of these types, lifting c (of type $a \sim_{\#} \mathit{Int}$) over lists ($[c]$, of type $[a] \sim_{\#} [\mathit{Int}]$), before applying symmetry. We urge the reader to consult Sulzmann et al. (2007) and Weirich et al. (2011) for more examples and intuition.

A final remark: we will be presenting and discussing a number of $\mathrm{F}_\mathrm{C}^{\uparrow}$ programs in the rest of the paper. For readability purposes we will sometimes omit type or kind applications in $\mathrm{F}_\mathrm{C}^{\uparrow}$ terms when these types or kinds are obvious from the context, making the syntax appear less verbose.

4. Two forms of equality

In Section 2 we sketched how to use type-equality evidence to support deferred type errors, using $\sigma \sim \tau$ as the type of equality evidence. Then in Section 3 we introduced our intermediate language, System $\mathrm{F}_\mathrm{C}^{\uparrow}$, in which explicit coercions of type $\sigma \sim_{\#} \tau$ represent evidence for the equality of two types. The distinction between $(\sim)$ and $(\sim_{\#})$ is crucial: it allows us to to marry a sound, erasable language of proofs with the potentially-unsound ability to have terms that compute proofs, as we discuss in this section.

4.1 The tension

Types have a very valuable property that programmers take for granted: they give strong static guarantees, but they carry *no runtime overhead*. This zero-overhead guarantee is formally justified by an *erasure* property: we can erase all the types before running the program, without changing the result.

Can we offer a similar guarantee for coercions? Yes, we can. System $\mathrm{F}_\mathrm{C}^{\uparrow}$ is carefully designed so that coercion abstractions, coercion applications, and casts, are all statically erasable, just like type abstractions and applications.

But this statement is manifestly in tension with our approach to deferred type errors. Consider once more the $\mathrm{F}_\mathrm{C}^{\uparrow}$ term

```
let (c : Char ∼ Bool) = error "Couldn't match..."
in snd (True, (cast 'a' c) && False)
```

Obviously we cannot erase the binding of c and the cast, leaving *snd* (*True*, `'a'` && *False*), because the latter will crash. So it seems that insisting on complete erasure of equalities kills the idea of deferring type errors stone dead!

4.2 The two equalities

We now present our resolution of this tension. We have carefully maintained a distinction between

- $(\sim_{\#})$, the type of primitive coercions γ in $\mathrm{F}_\mathrm{C}^{\uparrow}$, which are fully erasable, and
- $(\sim)$, type of evidence generated by the type inference engine, which cannot be erased.

However $(\sim)$ is not some magical built-in device. Rather, we can define it as a perfectly ordinary GADT (like the one we have already seen in Section 3.2), thus:

```
data a ∼ b where
  Eq# :: (a ∼# b) → a ∼ b
```

[2] We informally use Haskell's notation $[\tau]$ for the type list of τ, and *Cons* and *Nil* as its constructors.

This definition introduces a new algebraic data type constructor $(\sim)$, belonging in the T syntactic category of Figure 1. It has exactly one data constructor $Eq_{\#}$, whose (important) argument is a *static equality proof*. Readers familiar with proof assistants or type theory will immediately recognize in the definition of $(\sim)$ the type used traditionally in such systems to internalize *definitional equality* as a type (e.g. `refl_equal` in Coq).

Like $(\sim_{\#})$, the data type $(\sim)$ is polymorphically kinded:

$$\begin{array}{l} \sim\ : \forall \chi \,.\, \chi \to \chi \to \star \\ Eq_{\#} : \forall \chi \,.\, \forall (a:\chi)\,(b:\chi)\,.\,(a \sim_{\#} b) \to (a \sim b) \end{array}$$

As with $\tau \sim_{\#} \sigma$ we usually omit the kind application in $\tau \sim \sigma$ as a syntactic convenience.

The key point is that if $\gamma : \sigma \sim_{\#} \tau$, then a value $Eq_{\#}\ \gamma$ is an ordinary term, built with the data constructor $Eq_{\#}$, and having type $\sigma \sim \tau$. Given the GADT $(\sim)$ we can define the function *cast* that takes such a term-level equality witness and casts a value between equivalent types:

$$\begin{array}{l} cast : \forall (a\,b:\star)\,.\,a \to (a \sim b) \to b \\ cast = \Lambda (a\,b:\star)\,.\,\lambda(x:a)\,.\,\lambda(eq:a \sim b)\,. \\ \qquad\quad \mathbf{case}\ eq\ \mathbf{of}\ Eq_{\#}\ (c : a \sim_{\#} b) \to x \triangleright c \end{array}$$

Each use of *cast* forces evaluation of the coercion, *via* the **case** expression and, in the case of a deferred type error, that is what triggers the runtime failure.

Just as *cast* is a lifted version of $\triangleright$, we can lift all the coercion combinators from the $(\sim_{\#})$ type to $(\sim)$. For example:

$$\begin{array}{l} mkRefl :: \forall \chi \,.\, \forall (a:\chi)\,.\,a \sim a \\ mkRefl = \Lambda \chi \,.\, \Lambda (a:\chi)\,.\,Eq_{\#}\ \chi\ a\ a\ \langle a \rangle \\ mkSym :: \forall \chi \,.\, \forall (a\,b:\chi)\,.\,(a \sim b) \to (b \sim a) \\ mkSym = \Lambda \chi \,.\, \Lambda (a\,b:\chi)\,.\,\lambda(c:a \sim b)\,. \\ \qquad\quad \mathbf{case}\ c\ \mathbf{of}\ Eq_{\#}\ c \to Eq_{\#}\ \chi\ b\ a\ (sym\ c) \end{array}$$

The relationship between $(\sim)$ and $(\sim_{\#})$ is closely analogous to that between *Int* and $Int_{\#}$, described twenty years ago in our implementation of arithmetic in GHC (Peyton Jones and Launchbury 1991). Concretely, here is GHC's implementation of addition on *Int*:

$$\begin{array}{l} \mathbf{data}\ Int = I_{\#}\ Int_{\#} \\ plusInt :: Int \to Int \to Int \\ plusInt\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ I_{\#}\ x' \to \\ \qquad\qquad\quad \mathbf{case}\ y\ \mathbf{of}\ I_{\#}\ y' \to I_{\#}\ (x' +_{\#} y') \end{array}$$

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). This constructor has a single argument, of type $Int_{\#}$, which is the type of *unboxed integers*, a honest-to-goodness 32-bit integer value just like C's *int*. Finally $(+_{\#})$ is the machine 32-bit addition instruction. We may summarise the relationships thus:

- A value of type *Int*, or $\sigma \sim \tau$, is always heap-allocated; it is always represented by a pointer to the heap-allocated object; and the object can be a thunk.
- A value of type $Int_{\#}$, or $\sigma \sim_{\#} \tau$ is never heap-allocated; and it cannot be a thunk. There is no bottom value of type $Int_{\#}$, or $\sigma \sim_{\#} \tau$; we say that they are *unlifted types*.
- The *plusInt* function lifts the primitive addition $+_{\#}$ from $Int_{\#}$ to *Int*, by explicitly evaluating and unboxing its arguments; and the function *mkSym* works in just the same way.

The main difference between *Int#* and $a \sim_{\#} b$ is that the fomer is represented by a 32-bit unboxed value, whereas the latter has a structure that is *irrelevant* for the execution of a program, and can be represented by a zero-bit value, or entirely erased — it comes to the same thing in the end.

5. Type inference and deferral of type errors

With $F_C^{\uparrow}$ in hand we can now explain in more detail the mechanism of deferring type errors. We begin by sketching a little more about the type inference process.

5.1 Type inference by constraint generation

GHC's type inference algorithm works in two stages (Vytiniotis et al. 2011):

- Step 1: traverse the syntax tree of the input Haskell term, generating *type constraints* together with an *elaborated term*[3] in System $F_C^{\uparrow}$.
- Step 2: solve the constraints, creating $F_C^{\uparrow}$ bindings that give evidence for the solution.

For example, consider the term *show xs*, where $xs : [Int]$, and $show : \forall a\,.\,Show\ a \Rightarrow a \to String$. In Step 1 we generate:

$$\begin{array}{rl} \text{Elaborated term:} & show\ [Int]\ d_6\ xs \\ \text{Constraint:} & d_6 : Show\ [Int] \end{array}$$

The elaborated term looks much like the original except that *show* is now applied to a *type argument* $[Int]$ (corresponding to the "$\forall a$" in *show*'s type) and an *evidence argument* d_6 (corresponding to the "$Show\ a \Rightarrow$" in its type). The constraint is given a fresh name, d_6 in this example, which is mentioned in the elaborated term. Afficionados of Haskell's type-class system will recognise d_6 as *show*'s dictionary argument: it is simply a tuple of functions, the methods of the *Show* class.

When Step 1 is complete, the constraint solver solves the generated constraints, producing *evidence bindings*:

$$d_6 : Show\ [Int] = \$dShowList\ Int\ \$dShowInt$$

Here the solver has constructed a dictionary for $Show\ [Int]$, using the dictionary-construction functions that arise from the instance declarations of the class *Show*:

$$\begin{array}{rcl} \$dShowInt & : & Show\ Int \\ \$dShowList & : & \forall a\,.\,Show\ a \to Show\ [a] \end{array}$$

Finally, the evidence bindings are wrapped around the term in a **let** to make an executable term:

$$\begin{array}{l} \mathbf{let}\ d_6 = \$dShowList\ Int\ \$dShowInt \\ \mathbf{in}\ show\ [Int]\ d_6\ xs \end{array}$$

5.2 Equality constraints

This is all quite conventional. Somewhat less conventionally (but following the French school of type inference (Pottier and Rémy 2005)) GHC generates *equality constraints* as well as type-class constraints. We simplified the *show* example; in reality GHC generates the following:

$$\begin{array}{rl} \text{Elaborated term:} & show\ \alpha\ d_6\ (cast\ xs\ c_5) \\ \text{Constraints:} & d_6 : Show\ \alpha \\ & c_5 : [Int] \sim \alpha \end{array}$$

When instantiating *show*'s type in Step 1, the constraint generator does not yet know that it will be applied to the type $[Int]$, so instead it creates a fresh unification variable α, and uses that to instantiate *show*'s type. Later, when checking *show*'s argument x, it must ensure that *show*'s argument type α is equal to the actual type of *xs*, namely $[Int]$. It ensures this (eventual) equality by generating

[3] In reality we first generate an elaborated term by decorating the Haskell source term, and *then* desugar it, but we will ignore that extra step here.

an *equality constraint* $c_5 : [Int] \sim \alpha$, again with an arbitrary fresh name c_5 which names the evidence for the equality. Lastly, in the elaborated term, we use the cast term *cast xs* c_5 to convert *xs* into a term of type α. Notice that c_5's type $[Int] \sim \alpha$ uses the boxed equality $(\sim)$ rather than the primitive $F_C^{\uparrow}$ equality $(\sim_\#)$ (Section 4.2).

In Step 2, the constraint solver has a little extra work to do: as well as solving the constraints and giving bindings for the evidence variables, it must also produce bindings for the unification variables. In our running example, the solution to the constraints looks like this:

$$\begin{array}{rcl} \alpha & = & [Int] \\ c_5 : [Int] \sim \alpha & = & mkRefl\ [Int] \\ d_6 : Show\ [Int] & = & \$dShowList\ Int\ \$dShowInt \end{array}$$

The solver decided to eliminate α by substituting $[Int]$ for α, the first of the above bindings. The equality c_5 now witnesses the vacuous equality $[Int] \sim [Int]$, but it must still be given a binding, here *mkRefl* $[Int]$. (Recall that *mkRefl* was introduced in Section 4.2.)

Actually, as a matter of efficiency, in our real implementation the constraint generator solves many simple and immediately-soluble equalities (such as $\alpha \sim [Int]$) "on the fly" using a standard unification algorithm, rather than generating an equality constraint to be solved later. But that is a mere implementation matter; the implementation remains faithful to the semantics of generate-and-solve. Moreover, it certainly cannot solve *all* equalities in this way, because of GADTs and type families (Vytiniotis et al. 2011).

5.3 Deferring type errors made easy

In a system generating equality proofs using the $(\sim)$ datatype, which has values that can be inhabited by ordinary terms, it is delightfully easy to support deferred type errors. During constraint generation, we generate a type-equality constraint *even for unifications that are manifestly insoluble*. During constraint solving, instead of emitting an error message when we encounter an insoluble constraint, we emit a warning, and create a value binding for the constraint variable, which binds it to a call to *error*, applied to the error message string that would otherwise have been emitted at compile time. And that's all there is to it.

It is worth noting several features of this implementation technique:

- Each $F_C^{\uparrow}$ term given above *is a well-typed* $F_C^{\uparrow}$ term, even though some are generated from a type-incorrect Haskell term. Of course it can fail at run-time, but it does so in a civilized way, by raising an exception, not by causing a segmentation fault, or performing (&&) of a character and a boolean. You might consider that a program that fails at runtime in this way is not well-typed, in Milner's sense of "well typed programs do not go wrong". But Haskell programs can *already* "go wrong" in this way — consider (*head* []) for example — so matters are certainly no worse than in the base language.

  In short, *we have merely deferred the type errors to runtime; we have not ignored them!*

- Deferring type errors is not restricted to interpreted expressions typed at the interactive GHCi prompt. You can compile any module with `-fdefer-type-errors` and GHC will produce a compiled binary, which can be linked and run.

- There is no reflection involved, nor run-time type checking. Indeed there is no runtime overhead whatsoever: the program runs at full speed unless it reaches the point where a runtime error is signalled, in which case it halts. (This claim assumes the optimisations described in Section 7.)

- The technique makes elegant use of laziness. In a call-by-value language, a strict binding of *c* in Section 2.1 would be evaluated by the call *fst foo*, or even when *foo* is bound, and it would obviate the entire exercise if that evaluation triggered the runtime error! Nevertheless, the idea can readily be adapted for call-by-value, by simply making $(\sim)$ into a sum type:

  $$\begin{array}{l} \textbf{data}\ a \sim b\ \textbf{where} \\ \quad Eq_\# \ \ :: (a \sim_\# b) \to a \sim b \\ \quad Error :: String \quad\ \to a \sim b \end{array}$$

  Now, the "evidence" for an erroneous type constraint would be an *Error* value, and evaluating that is fine. We simply need to adjust *cast* to deal with the *Error* case:

  $$\begin{array}{l} cast = \Lambda(a\ b : \star)\,.\ \lambda(x : a)\,.\ \lambda(eq : a \sim b)\,. \\ \qquad \textbf{case}\ eq\ \textbf{of} \\ \qquad\quad Eq_\#\ (c : a \sim_\# b) \to x \triangleright c \\ \qquad\quad Error\ s \to error\ s \end{array}$$

- The technique works uniformly for *all* type constraints, not only for equality ones. For example, in Section 5.1, suppose there was no instance for *Show* $[Int]$. Then the constraint d_6 : *Show* $[Int]$ would be insoluble, so again we can simply emit a warning and bind d_6 to an error thunk. Any program that needs the evidence for d_6 will now fail; those that don't, won't.

- We can defer all *type* errors in terms, but not *kind* errors in types. For example, consider

  $$\begin{array}{l} \textbf{data}\ T = MkT\ (Int\ Int) \\ f\ (MkT\ x) = x \end{array}$$

  The type *Int Int* simply does not make sense – applying *Int* to *Int* is a kind error, and we do not have a convenient way to defer *kind* errors, only type errors.

5.4 The placement of errors

Since many different parts of a program may contribute to a type error, there may be some non-determinism about *how delayed* a deferred type error will be. Suppose that $upper : [Char] \to [Char]$, and consider the term:

$$upper\ [True, \texttt{'a'}]$$

There are two type incompatibilities here. First, the boolean *True* and character `'a'` cannot be in the same list. Second, the function *upper* expects a list of characters but is given a list with a boolean in it. Here is one possible elaboration:

$$\begin{array}{rl} \text{Elaborated term:} & upper\ [cast\ True\ c_7, \texttt{'a'}] \\ \text{Constraints:} & c_7 : Bool \sim Char \end{array}$$

But the program could also be elaborated in another way:

$$\begin{array}{rl} \text{Elaborated term:} & upper\ (cast\ [True, cast\ \texttt{'a'}\ c_8]\ c_9) \\ \text{Constraints:} & c_8 : Char \sim Bool \\ & c_9 : [Bool] \sim [Char] \end{array}$$

In this case, type inference has cast `'a'` to *Bool* using c_8, so that it can join *True* to form a list of *Bool*; and then cast the list $[Bool]$ to $[Char]$ using c_9 to make it compatible with *upper*. The two elaborated programs have slightly different runtime behaviour. If the term is bound to *tm*, then *head* (*tail tm*) will run successfully (returning `'A'`) in the first elaboration, but fail with a runtime error in the second.

We might hope that the type inference engine inserts as few casts as possible, and that it does so as near to the usage site as possible. In fact this turns out to be the case, because the type inference engine uses the (known) type of *upper* to type-check its argument, expecting the result to be of type $[Char]$. This idea of "pushing down" the expected type into an expression, sometimes called *bidirectional* or *local* type inference (Pierce and Turner 2000), is already implemented in GHC to improve the quality of error messages; see Peyton Jones et al. (2007, Section 5.4) for details. Although it is a heuristic, and not a formal guarantee, the mechanism localises the casts very nicely in practice.

Nevertheless, the bottom line is that the dynamic semantics of a type-incorrect program depends on hard-to-specify implementation details of the type inference algorithm. That sounds bad! But we feel quite relaxed about it:

- The issue arises only for programs that contain errors. Type correct programs have their usually fully-specified semantics.
- Regardless of the precise placement of coercions, the elaborated program is type correct. This is not a soundness issue.
- The imprecision of dynamic semantics is no greater a shortcoming than the lack of a formal specification of the precise type error message(s) produced by a conventional type inference engine for a type-incorrect program. And yet no compiler or paper known to us gives a formal specification of what type errors are produced for a type-incorrect program.

That said, it is interesting to reflect on approaches that might tighten up the specification, and we do so in Section 9.

5.5 Summary

There are many reasons why evidence-based type elaboration, using constraint generation and subsequent constraint solving, is desirable (Vytiniotis et al. 2011):

- It is *expressive*, readily accommodating Haskell's type classes, implicit parameters, and type families.
- It is *modular*. The constraint *generator* accepts a very large input language (all of Haskell), so it has many cases, but each case is very simple. In contrast, the constraint *solver* accepts a very small input language (the syntax of constraints) but embodies a complex solving algorithm. Keeping the two separate is incredibly wonderful.
- It is *robust*: for example it does not matter whether the constraint generator traverses the term left-to-right or right-to-left: the resulting constraint is the same either way.
- Neither the constraint generator nor the solver need be trusted; a very simple, independent checker can type-check the elaborated $F_C^{\uparrow}$ term.

To this list we can now add a completely new merit: it is dead easy to implement deferred type errors, a feature that is desired by many Haskell programmers.

6. Discussion

Now that we have discussed type inference in more detail, we pause to reflect on our design choices.

6.1 Evidence uniformity

We've seen that deferring type errors provides a good motivation for treating coercions as term-level constructs. But there is another way in which treating coercions as values turns out to be very convenient. In the Haskell source language, *equality constraints* are treated uniformly with *type-class constraints* and *implicit-parameter constraints*; anywhere a class constraint can appear, an equality constraint can appear, and vice versa. Class constraints and implicit-parameter constraints definitely cannot be erased: by design their evidence carries a runtime value. Treating some constraints as non-erasable values and others (the equalities) as type-like, erasable constructs, led to many annoying special cases in the type inference and source-to-$F_C^{\uparrow}$ elaboration of Haskell programs.

The most troublesome example of this non-uniformity arises when treating Haskell's superclasses. Consider the declaration

$$\textbf{class}\ (a \sim F\ b, Eq\ a) \Rightarrow C\ a\ b\ \textbf{where} \ldots$$

Here $Eq\ a$ is a superclass of $C\ a\ b$, meaning that from evidence for $C\ a\ b$ one can extract evidence for $Eq\ a$. Concretely this extraction is witnessed by a field selector:

$$sc_2 : C\ a\ b \to Eq\ a$$

which takes a dictionary (i.e. record of values) for $C\ a\ b$ and picks out the $Eq\ a$ field. In just the same way one should be able to extract evidence for $a \sim F\ b$, which suggests a selector function with type

$$sc_1 : C\ a\ b \to (a \sim F\ b)$$

Before we distinguished $(\sim)$ and $(\sim_{\#})$ we could not write this function because there simply is no such function in $F_C^{\uparrow}$; indeed the type $C\ a\ b \to (a \sim_{\#} F\ b)$ is not even well kinded in Figure 3. There is a good reason for this: dictionaries can be recursively defined and can diverge (Lämmel and Peyton Jones 2005), so the selector function may diverge when evaluating its arguments – but the type $a \sim_{\#} F\ b$ cannot represent divergence, because that would be unsound.

The tension is readily resolved by $(\sim)$; the type of sc_1 is well formed and its definition looks something like this:

$$\begin{aligned} sc_1 = \Lambda ab.\ \lambda (d : C\ a\ b).&\\ \textbf{case}\ d\ \textbf{of}&\\ MkC\ (c : a \sim_{\#} F\ b)\ (eq : Eq\ a) \ldots \to Eq_{\#}\ c& \end{aligned}$$

This accumulation of infelicities led us to the two-equality plan, and in fact we had fully implemented this design even *before* we ever thought of deferring type errors. Now, we get the ability to defer errors regarding type unification, missing class constraints, and implicit parameters, all in one go.

6.2 Why kind polymorphism is important

We have mentioned that both equalities $(\sim_{\#})$ and $(\sim)$ are kind-polymorphic, but we have not yet said why. Right from the beginning Haskell has featured kinds other than $\star$, such as $\star \to \star$. During type inference, when unifying, say, $\alpha\ \beta \sim Maybe\ Int$, the inference engine — or, more precisely, the constraint solver — must decompose the equality to give $\alpha \sim Maybe$ and $\beta \sim Int$. The former equality is at kind $\star \to \star$, so it follows that the $(\sim)$ type constructor itself must be either (a) magically built in or (b) poly-kinded. And similarly $(\sim_{\#})$.

Solution (a) is entirely possible: we could add $\sigma \sim \tau$ and $\sigma \sim_{\#} \tau$ to the syntax of types, and give them their own kinding rules. But there are unpleasant knock-on effects. The optimizer would need to be taught how to optimize terms involving $(\sim)$. Worse, it turns out that we need equalities between equalities, thus $(\sigma_1 \sim \tau_1) \sim (\sigma_2 \sim \tau_2)$, which in turn leads to the need for new coercion combinators to decompose such types.

Happily there are many other reasons for wanting kind polymorphism in $F_C^{\uparrow}$ (Yorgey et al. 2012), and once we have kind polymorphism we can readily make the equality type constructors kind-polymorphic.

6.3 What the Haskell programmer sees

A salient feature of Haskell's qualified types (type classes, implicit parameters, equality constraints) is that the type inference engine fills in the missing evidence parameters. So if f has type

$$f :: (Num\ b, a \sim F\ b) \Rightarrow a \to b \to b$$

then given a source-language call $(f\ e_1\ e_2)$, the type inference will generate the elaborated call $(f\ \sigma\ \tau\ d\ c\ e_1\ e_2)$, where σ and τ are the types that instantiate a and b, and d and c are evidence terms that witness that $Num\ \tau$ holds and that $\sigma \sim F\ \tau$, respectively.

One might wonder whether one can (in Haskell) also write

$$g :: (a \sim_{\#} F\ b) \Rightarrow a \to b \to b$$

and have the above evidence-generation behaviour. No, you cannot. The whole point of the $(\sim)$ type is that it can be treated uniformly with other evidence (bound in letrec, returned as a result of a call), whereas $(\sim_{\#})$ cannot. So in the source language $\sigma \sim_{\#} \tau$ is not a type constraint you can write before the "$\Rightarrow$" in a Haskell type, and have it participate in constraint solving. The entire constraint generation and solving process works exclusively with well-behaved, uniform, boxed constraints. Only when constraint solving is complete does $(\sim_{\#})$ enter the picture, as we discuss next.

7. Optimizing equalities

We now have a correct implementation, but it looks worryingly expensive. After all, the constraint generation/solving process may generate a program littered with coercion bindings and casts, all of which are invisible to the programmer, have no operational significance, and merely ensure that "well typed programs don't go wrong". Yet each will generate a heap-allocated $Eq_{\#}$ box, ready to be evaluated by *cast*. Happily, almost all of these boxes are eliminated by existing optimizations within GHC, as this section describes.

7.1 Eliminating redundant boxing and unboxing

The fact that we have defined $(\sim)$ as an ordinary GADT means that is fully exposed to GHC's optimizer. Consider a Haskell 98 program that turns out to be well typed. The constraint generator will produce many constraints that are ultimately solved by reflexivity, because the two types really are equal. Here is a typical case of an elaborated term:

$$\begin{array}{l}\textbf{let}\ (c : Char \sim Char) = mkRefl\ Char \\ \textbf{in} \ldots (cast\ e\ c) \ldots\end{array}$$

(Recall that *mkRefl* and *cast* were defined in Section 4.2.) As it stands, the **let** will heap-allocate a thunk which, when evaluated by the *cast*, will turn out to be an $Eq_{\#}$ constructor wrapping a reflexive coercion $\langle Char \rangle$. All this is wasted work. But GHC's optimizer can inline the definitions of *mkRefl* and *cast* from Section 4.2 to get

$$\begin{array}{l}\textbf{let}\ (c : Char \sim Char) = Eq_{\#}\ \star\ Char\ Char\ \langle Char \rangle \\ \textbf{in} \ldots (\textbf{case}\ c\ \textbf{of}\ Eq_{\#}\ c' \to e \triangleright c') \ldots\end{array}$$

Now it can inline c at its use site, and eliminate the case expression, giving

$$\ldots (e \triangleright \langle Char \rangle) \ldots$$

Remembering that primitive casts $(\triangleright)$ can be erased, we have eliminated the overhead. Moreover, the optimizations involved have all been in GHC for years; there is no new special purpose magic.

What happens when a deferred type error means that a cast cannot, and should not, be erased? Consider once more the $F_C^{\uparrow}$ term

$$\begin{array}{l}\textbf{let}\ (c : Char \sim Bool) = error\ \texttt{"Couldn't match..."} \\ \textbf{in}\ snd\ (True, (cast\ \texttt{'a'}\ c)\ \&\&\ False)\end{array}$$

Now, simply by inlining *cast* and c, the optimizer can transform to

$$\begin{array}{l}snd\ (True, (\textbf{case}\ error\ \texttt{"..."}\ \textbf{of}\ \{Eq_{\#}\ c \to \texttt{'a'} \triangleright c\}) \\ \qquad\qquad \&\&\ False)\end{array}$$

After inlining (&&), and simplifying **case**-of-*error* to just a call of *error*, both standard transformations in GHC's optimizer) we get

$$snd\ (True, error\ \texttt{"..."})$$

Even erroneous programs are optimized by removing their dead code! The point is this: by exposing the evaluation of coercions, we allow the *existing* optimizer transformations to work their magic.

7.2 Equalities and GADTs

Let us reconsider the GADT example given in Section 3.2:

$$\begin{array}{l}\textbf{data}\ T\ a\ \textbf{where} \\ \quad T_1 :: Int \to T\ Int \\ \quad T_2 :: a \quad \to T\ a\end{array}$$

There we said that the constructor T_1 is typed thus:

$$T_1 : \forall a\,.\,(a \sim_{\#} Int) \to Int \to T\ a$$

That is true in System $F_C^{\uparrow}$. But the Haskell programmer, who knows only of the type $\sigma \sim \tau$, considers T_1 to have this type:

$$T_1 :: \forall a\,.\,(a \sim Int) \to Int \to T\ a$$

It would be perfectly sound to adopt the latter type for T_1 in the elaborated program; for example, function f from Section 3.2 would be elaborated thus:

$$\begin{array}{l}f = \Lambda a\,.\,\lambda(x : T\ a)\,. \\ \quad \textbf{case}\ x\ \textbf{of} \\ \qquad T_1\ (c : a \sim Int)\ (n : Int) \\ \qquad\quad \to cast\ (Cons\ (n+1)\ Nil)\ (mkSym\ [c]) \\ \qquad T_2\ v \to Nil\end{array}$$

Since an argument of type $a \sim Int$ has a lifted type with a boxed representation, it would take up a whole word in every T_1 object. Moreover, since c is bound by the pattern match, the **case** expression in *mkSym* will not cancel with an *Eq#* box in the binding for c. This is not good! What has become of our zero-overhead solution?

The solution is simple: we desugar GADTs to contain *unlifted*, rather than *lifted*, equalities. We can do this in such a way that the Haskell programmer still sees only the nice well-behaved $(\sim)$ types, as follows. First, in the elaborated program the type of T_1 is:

$$T_1 : \forall a\,.\,(a \sim_{\#} Int) \to Int \to T\ a$$

However, the elaborator replaces every source-language call of T_1 with a call of a constructor wrapper function, *T1wrap*, defined like this:

$$\begin{array}{l}T1wrap : \forall a\,.\,(a \sim Int) \to Int \to T\ a \\ T1wrap = \Lambda(a : \star)\,.\,\lambda(c : a \sim Int)\,.\,\lambda(n : Int)\,. \\ \qquad\qquad \textbf{case}\ c\ \textbf{of}\ Eq_{\#}\ c_1 \to T_1\ c_1\ n\end{array}$$

The wrapper deconstructs the evidence and puts the payload into T_1 where, since it is erasable, it takes no space.

Dually, a source-program pattern match is elaborated into a $F_C^{\uparrow}$ pattern match together with code to re-box the coercion. So our function f is elaborated thus:

$$\begin{array}{l}f = \Lambda a\,.\,\lambda(x : T\ a)\,. \\ \quad \textbf{case}\ x\ \textbf{of} \\ \qquad T_1\ (c_1 : a \sim_{\#} Int)\ (n : Int)\end{array}$$

$$\begin{array}{l}
\quad\quad \rightarrow \textbf{let}\; c = \mathit{Eq}_{\#}\; c_1 \quad \text{-- Re-boxing} \\
\quad\quad\quad \textbf{in}\; \mathit{cast}\;(\mathit{Cons}\;(n+1)\;\mathit{Nil})\;(\mathit{mkSym}\;[c]) \\
\quad T_2\; v \rightarrow \mathit{Nil}
\end{array}$$

Now the earlier optimizations will get rid of all the boxing and unboxing and we are back to nice, efficient code. The technique of unboxing strict arguments on construction, and re-boxing on pattern matching (in the expectation that the re-boxing will be optimized away) is precisely what GHC's `UNPACK` pragma on constructor arguments does. So, once more, coercions can hitch a free ride on some existing machinery.

7.3 How much is optimized away?

We have argued that the boxing and unboxing, introduced in the elaborated program by the type checker, will largely be eliminated by standard optimizations. But not always! Indeed that is the point: the main reason for going to all of this trouble is to handle smoothly the cases (deferred type errors, recursive dictionaries) when equalities cannot, and should not, be erased. But still, one might reasonably ask, can we offer any guarantees at all?

Consider a type-correct Haskell program that contains (a) no equality superclasses, and (b) no functions that take or return a value of type $\sigma \sim \tau$, apart from GADT constructors. This includes all Haskell 98 programs, and all GADT programs. After typechecking and elaboration to $\mathrm{F}_\mathrm{C}^{\uparrow}$, suppose that we inline every use of *mkRefl*, *mkSym*, etc, and the GADT constructor wrappers. Then

- Every use of a variable of type $\sigma \sim \tau$ will be a case expression that scrutinises that variable, namely the unboxing **case** expressions in *mkRefl*, *mkSym*, etc, and GADT constructor wrappers.
- Every binding of an evidence variable of type $\sigma \sim \tau$ will be a **let** whose right hand side returns a statically-visible $\mathit{Eq}_{\#}$ box.

By inlining these **let**-bound evidence variables at their use sites, we can cancel the **case** with the $\mathit{Eq}_{\#}$ constructors, thereby eliminating all boxing and unboxing. To illustrate, consider once more the elaboration of function f at the end of the previous subsection. If we inline *mkSym* and *cast* we obtain:

$$\begin{array}{l}
f = \Lambda a\,.\,\lambda(x : T\;a)\,. \\
\quad \textbf{case}\; x\; \textbf{of} \\
\quad\quad T_1\;(c_1 : a \sim_{\#} \mathit{Int})\;(n : \mathit{Int}) \\
\quad\quad\quad \rightarrow \textbf{let}\; c\; = \mathit{Eq}_{\#}\; c_1\; \textbf{in} \quad \text{-- Re-boxing} \\
\quad\quad\quad\quad \textbf{let}\; c_2 = \textbf{case}\; c\; \textbf{of}\; \mathit{Eq}_{\#}\; c' \rightarrow \mathit{Eq}_{\#}\;(\mathit{sym}\;[c']) \\
\quad\quad\quad\quad \textbf{in case}\; c_2\; \textbf{of} \\
\quad\quad\quad\quad\quad \mathit{Eq}_{\#}\; c_2' \rightarrow \mathit{Cons}\;(n+1)\;\mathit{Nil} \triangleright c_2' \\
\quad\quad T_2\; v \rightarrow \mathit{Nil}
\end{array}$$

The right hand side of c_2 comes from inlining *mkSym*, while the "**case** $c_2\ldots$" comes from inlining *cast*. Now we can inline c in "**case** $c\ldots$", and c_2 in "**case** $c_2\ldots$", after which the cancellation of boxing and unboxing is immediate.

When does this *not* work? Consider exception (b) above, where a programmer writes a function that is explicitly abstracted over an equality:

$$\begin{array}{l}
f : \forall a\,.\,F\;a \sim \mathit{Int} \Rightarrow [F\;a] \rightarrow \mathit{Int} \\
f\;x = \mathit{head}\;x + 1
\end{array}$$

The elaborated form will look like this:

$$\begin{array}{l}
f : \forall a\,.\,F\;a \sim \mathit{Int} \rightarrow [F\;a] \rightarrow \mathit{Int} \\
f = \Lambda a\,.\,\lambda(c : F\;a \sim \mathit{Int})\,.\,\lambda(x : [F\;a])\,. \\
\quad\quad \mathit{head}\;(\mathit{cast}\;x\;c) + 1
\end{array}$$

Since c is lambda-bound, there is no $\mathit{Eq}_{\#}$ box for the *cast* to cancel with. However we can perform the same worker/wrapper split to this user-defined function that we did for constructors, thus

$$\begin{array}{l}
f : \forall a\,.\,F\;a \sim \mathit{Int} \rightarrow [F\;a] \rightarrow \mathit{Int} \\
f = \Lambda a\,.\,\lambda(c : F\;a \sim \mathit{Int})\,.\,\lambda(x : [F\;a])\,. \\
\quad\quad \textbf{case}\; c\; \textbf{of}\; \mathit{Eq}_{\#}\; c' \rightarrow \mathit{fwrk}\; c'\; x \\
\mathit{fwrk} : \forall a\,.\,F\;a \sim_{\#} \mathit{Int} \rightarrow [F\;a] \rightarrow \mathit{Int} \\
\mathit{fwrk} = \Lambda a\,.\,\lambda(c' : F\;a \sim_{\#} \mathit{Int})\,.\,\lambda(x : [F\;a])\,. \\
\quad\quad \textbf{let}\; c = \mathit{Eq}_{\#}\; c'\; \textbf{in}\; \mathit{head}\;(\mathit{cast}\;x\;c) + 1
\end{array}$$

Now in *fwrk* the boxing and unboxing cancels; and dually we are free to inline the wrapper function f at its call sites, where the unboxing will cancel with the construction. This worker/wrapper mechanism is precisely what GHC already implements to eliminate boxing overheads on strict function arguments (Peyton Jones and Santos 1998), so it too comes for free. There *is* a small price to pay, however: the transformation makes the function strict in its equality evidence, and that in turn might trigger a deferred error message slightly earlier than strictly necessary. In our view this is a trade-off worth making. In our current implementation the worker/wrapper transform on equalities is only applied when the function really is strict in the equality; we have not yet added the refinement of *making* it strict in all equalities.

7.4 Summary

In short, although we do not give a formal theorem (which would involve formalizing GHC's optimizer) we have solid grounds, backed by observation of optimized code, for stating that the uniform representation of equality evidence can be successfully optimized away in all but the cases in which it cannot and should not be eliminated, namely for deferred type errors and functions that must accept or return term-level equalities (such as selectors for equality superclasses). Of course, introducing and then eliminating all these boxes does mean a lot more work for the compiler, but this has not proved to be a problem in practice.

8. Related work

8.1 Relation to gradual and hybrid type systems

There is a very large body of work on gradual and hybrid typing that addresses the problem of deferring unprovable goals at compile time as static runtime checks, with a particular emphasis on refinement types and blame assignment, and interoperation between statically and dynamically typed parts of a language (Flanagan 2006; Siek and Taha 2006; Siek and Vachharajani 2008; Tobin-Hochstadt and Felleisen 2006).

Our treatment of coercions does not *replace* static type errors by runtime checks, but rather *delays* triggering a static error until the offending part of the program is evaluated, at runtime. For instance, consider the following program, which contains a static error but is compiled with `-fdefer-type-errors`:

$$\begin{array}{l}
f :: \forall a\,.\,a \rightarrow a \rightarrow a \\
f\;x\;y = x\;\&\&\;y
\end{array}$$

There is a static error in this program because f is supposed to be polymorphic in the type of its arguments x and y, which are nevertheless treated as having type *Bool*. At runtime, even if we evaluate the application of f on arguments of type *Bool*, such as *f True False* we will get a type error "Couldn't match type a with *Bool*", despite the fact that the arguments of f *are* of type *Bool* at runtime. In contrast, a truly dynamic type-check would not trigger a runtime error.

In the context of GHC, there is no straightforward way to incorporate runtime *checks* instead of error triggers at runtime, unless dynamic type information is passed along with polymorphic types. It may be possible to do something along these lines, following previous work that incorporates polymorphism with hybrid type checking via dynamic types (Ahmed et al. 2011).

Finally, as we have seen in Section 5.3, there exists some non-determinism in the dynamic placement of the type error and "blame assignment". Previous work (Haack and Wells 2004) has focused on identifying parts of a program that contribute to a type error and would be potentially useful for reducing this non-determinism both for static error messages or for better specifying the dynamic behaviour of an erroneous program.

8.2 Deferring type errors

DuctileJ is a plugin to the Java compiler that converts a normal Java program to one in which type checking is deferred until runtime (Bayne et al. 2011). The authors provide an extensive discussion of the software engineering advantages of deferring type errors, under two main headings.

- During *prototyping*, programmers often comment out partly written or temporarily out-of-date code, while prototyping some new part. Commenting out is tiresome because one must do it consistently: if you comment out f you must comment out everything that calls f, and so on. Deferring type errors is a kind of lazy commenting-out process.
- During *software evolution* of a working system it can be burdensome to maintain global static type correctness. It may be more productive to explore a refactoring, or change of data representation, in part of a software system, and test that part, without committing to propagating the change globally.

We urge the reader to consult this excellent paper for a full exposition, and a good evaluation of the practical utility of deferring type errors both during prototyping and for software evolution.

Note however that although our motivations are similar, our implementation differs radically from that in DuctileJ. The latter works by a "de-typing" transformation that uses Java's runtime type information and reflection mechanisms to support runtime type checks. This imposes a significant runtime cost – the paper reports a slowdown between 1.1 and 7.8 times. In contrast, our implementation performs no runtime reflection and runs at full speed until the type error itself is encountered. The DuctileJ de-typing transformation is also not entirely faithful to the original semantics — unsurprisingly, the differences involve reflection — whereas ours is fully faithful, even for programs that involve Haskell's equivalent of reflection, the *Typeable* and *Data* classes.

Deferring type errors is also a valuable tool in the context of IDE development. In an IDE it is essential to provide feedback to the programmer even if the program has errors. The Visual Basic IDE uses a hierarchy of analysis to provide gradually more functionality depending on the type of errors present (Gertz 2005). For instance, if there are type errors, but no parse errors, smart indentation and automatic code pretty-printing can already be applied. However, more complicated refactorings require type information to be available. Some IDEs use error-correcting parsers to be able to provide some functionality in the presence of parsing errors, but a type error will require a correction by the user before the IDE can offer functionality that depends on the availability of types. Deferring type errors allows the compiler to complete type checking without fixing the type errors, allowing for a Haskell IDE to remain functional even for programs that do not type-check.

8.3 Proof erasure

Coq (The Coq Team) uses a *sort-based* erasure process by introducing a special universe for propositions, *Prop*, which is analogous to our $Constraint_\#$ kind. Terms whose type lives in *Prop* are erased even when they are applications of functions (lemmas) to computational terms. This is sound in Coq, since the computation language is also strongly normalizing. Extending the computation language of $F_C^\uparrow$ proofs or finding a way to restrict the ordinary computation language of $F_C^\uparrow$ using kinds in order to allow it to construct *primitive* equalities is an interesting direction towards true dependent types for Haskell.

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) presented 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. There also exist approaches that lie in between sort-based erasure and irrelevance-based erasure: for instance, in *implicit calculus of constructions* (Miquel 2001) explicitly marked static information (not necessarily *Prop*-bound) does not affect computation and can be erased (Barras and Bernardo 2008). In $F_C^\uparrow$ the result of a computation cannot depend on the structure of an equality proof, by construction: there is no mechanism to decompose the structure of a coercion at all at the term-level. Hence a coercion value needs no structure (since it cannot be decomposed), which allows us to perform full erasure without any form of irrelevance analysis.

This idea – of separating the "computational part" of a proof-like object, which always has to run before we get to a zero-cost "logical part" – is reminiscent of a similar separation that A-normal forms introduce in refinement type systems, for instance (Bengtson et al. 2008) or the more recent work on value-dependent types (Borgstrom et al. 2011; Swamy et al. 2011). This line of work seems the closest in spirit to ours, with similar erasure concerns, and there is rapidly growing evidence of the real-world potential of these ideas – see for instance the discussion and applications reported by Swamy et al. (2011).

9. Future work and conclusions

Error coercion placement This paper has been about an *implementation* technique that uses first-class proof-like objects to allow for deferred type errors with very low overhead. A natural next step would be towards a declarative specification of the "elaboration" process from source to a target language which specifies the placement of the deferred error messages on potentially erroneous sub-terms. Recent related work on coercion placement in the context of coercive subtyping is the work of Luo (2008) and Swamy et al. (2009); these would certainly be good starting points for investigations on a declarative specification of deferring type errors. The canonical reference for coercion placement in a calculus with type-dynamic is the work of Henglein (1994), but it seems somewhat disconnected from our problem as we do not have currently any way of dynamically passing type information or executing programs that contain static errors but are safe dynamically.

In general, this problem seems very hard to tackle without exposing some of the operation of the underlying constraint solver. In the other direction, a principled approach to deferring type errors might actually provide *guidance* on the order in which constraints should be solved. For instance, when solving the constraints $C_1 \cup C_2 \cup C_3$ arising from the expressions e_1, e_2, and e_3 in the term

if e_1 **then** e_2 **else** e_3, we might want to prioritise solving the constraint C_1. In this way, if an error is caused by the interaction of the expressions e_2 or e_3 with e_1, we would still be able to execute the condition of the branch e_1 before we emit a deferred type error for e_2 or e_3. Otherwise we run the danger of the term e_2 or e_3 forcing some unification that makes constraint C_1 insoluble, giving rise to an earlier error message (during evaluation of the condition term e_1). However, it is not clear what should happen when C_2 and C_3 have a common unification variable, and there is freedom in deferring either one, for instance. Therefore this problem is certainly worth further investigation.

The equality type Internalizing definitional equality ($\sim_\#$) as a type ($\sim$) is pretty standard in languages with dependent types (Licata and Harper 2005). For example, programming with first-class equality witnesses is sometimes convenient to avoid shortcomings of implementations of dependent pattern matching.

Recent work on higher-dimensional type theory (Licata and Harper 2012) goes one step further to show that the ($\sim$) datatype can be *extended* with yet another constructor for term-level isomorphisms between types. Interestingly the usual definitional equality inference rules apply for this extended equality type. Moreover they show that the term language can be equipped with an equational theory that is rich enough, so that types enjoy canonical forms. Of course the language they address is simpler in some respects (no partiality or divergence, no polymorphism), nor is there a reduction semantics. In a real compiler, being able to extend the ($\sim$) datatype with true computational isomorphisms and maintain soundness and providing a transparent treatment of these isomorphisms with minimal programmer intervention is an interesting direction for future research.

Conclusions In this paper we have proposed a simple and lightweight mechanism for deferring type errors, in a type-safe way that requires no program rewriting, and preserves the semantics of the program until execution reaches the part that contains a type error. We have shown that this can be done in an entirely safe way in the context of a typed intermediate language, and in fact without requiring any modifications to System $F_C^\uparrow$ or the compiler optimizer. This work is fully implemented in GHC, where it has in addition greatly simplified the implementation of type inference and elaboration of Haskell to $F_C^\uparrow$.

Acknowledgments

Particular thanks are due to Etienne Laurin, who suggested to us the idea of deferring type errors (GHC Trac ticket 5624), although his implementation was very different to the one described here. We thank Stephanie Weirich and Brent Yorgey for their detailed comments and suggestions, and the ICFP reviewers for the helpful feedback.

This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT) via the SFRH/BD/35999/2007 grant, and by EPSRC grant number EP/J010995/1.

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*, pages 57–71. Springer, 2011.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 201–214, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926409.

Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computation Structure*, pages 365–379, 2008. doi: 10.1007/978-3-540-78499-9_26.

Michael Bayne, Richard Cook, and Michael Ernst. Always-available static and dynamic feedback. In *Proceedings of 33rd International Conference on Software Engineering (ICSE'11)*, pages 521–530, Hawaii, 2011.

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.

Johannes Borgstrom, Juan Chen, and Nikhil Swamy. Verifying stateful programs with substructural state and Hoare types. In *Proceedings of the 5th ACM Workshop on Programming Languages meets Program Verification*, PLPV '11, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag.

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.

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

Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. ACM. doi: 10.1145/1111037.1111059.

Matthew Gertz. Scaling up: The very busy background compiler. *MSDN Magazine*, 6 2005. URL `http://msdn.microsoft.com/en-us/magazine/cc163781.aspx`.

Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50:189–224, March 2004. ISSN 0167-6423.

Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22:197–230, June 1994. ISSN 0167-6423.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 301–331. Springer London, 2010. doi: 10.1007/978-1-84882-912-1_14.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 204–215, New York, NY, USA, 2005. ACM. doi: 10.1145/1086365.1086391.

Daniel R. Licata and Robert Harper. A formulation of dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 337–348, New York, NY, USA, 2012. ACM. doi: 10.1145/2103656.2103697.

Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, August 2008. ISSN 0960-1295. doi: 10.1017/S0960129508006804.

Alexandre Miquel. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3-540-41960-8.

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, September 2007.

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 and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-309-3.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1–82, 2007. doi: 10.1017/S0956796806006034.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100.

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

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)*, pages 106–124, July 2004.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. doi: 10.1145/1408681.1408688.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Types in Language 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.

Nikhil Swamy, Juan Chen, Cedric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP'11, pages 266–278. ACM, September 2011. doi: 10.1145/2034773.2034811.

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

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. doi: 10.1145/1176617.1176755.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular Type inference with local assumptions. *Journal of Functional Programming*, 21, 2011.

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.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. doi: 10.1145/2103786.2103795.