2011

Generative Type Abstraction and Type-level Computation

Stephanie Weirich  
*University of Pennsylvania, sweirich@cis.upenn.edu*

Dimitrios Vytiniotis  
*Microsoft Research*

Simon Peyton Jones  
*Microsoft Research*

Stephan A. Zdancewic  
*University of Pennsylvania, stevez@cis.upenn.edu*

*Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)*

*Part of the [Computer Sciences Commons](http://repository.upenn.edu/cis_papers)*

**Recommended Citation**


© ACM, 2011. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proc. of the ACM Symposium on Principles of Programming Languages, { (2011)}* [http://doi.acm.org/10.1145/1925844.1926411](http://doi.acm.org/10.1145/1925844.1926411)  

Email permissions@acm.org

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/573](http://repository.upenn.edu/cis_papers/573)

For more information, please contact repository@pobox.upenn.edu.
Generative Type Abstraction and Type-level Computation

Abstract
Modular languages support generative type abstraction, ensuring that an abstract type is distinct from its representation, except inside the implementation where the two are synonymous. We show that this well-established feature is in tension with the non-parametric features of newer type systems, such as indexed type families and GADTs. In this paper we solve the problem by using kinds to distinguish between parametric and non-parametric contexts. The result is directly applicable to Haskell, which is rapidly developing support for type-level computation, but the same issues should arise whenever generativity and non-parametric features are combined.

Disciplines
Computer Sciences

Comments

doi> 10.1145/1925844.1926411

© ACM, 2011. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proc. of the ACM Symposium on Principles of Programming Languages, { (2011)} http://doi.acm.org/10.1145/1925844.1926411” Email permissions@acm.org
Abstract

Modular languages support generative type abstraction, ensuring that an abstract type is distinct from its representation, except inside the implementation where the two are synonymous. We show that this well-established feature is in tension with the non-parametric features of newer type systems, such as indexed type families and GADTs. In this paper we solve the problem by using kinds to distinguish between parametric and non-parametric contexts. The result is directly applicable to Haskell, which is rapidly developing support for type-level computation, but the same issues should arise whenever generativity and non-parametric features are combined.

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, Newtype deriving, Type functions

1. Introduction

Generative type abstraction allows programmers to introduce new type constants in their programs that are isomorphic to existing types; examples include ML's module system [Milner et al. 1997; Pierce 2005, Ch. 8], and Haskell's newtype construct [Peyton Jones et al. 2003]. Type generativity is important because it supports modularity by enforcing abstraction: the implementor of a module can move freely between the abstract and representation types, whereas to the client of the module the two types are completely distinct.

There is growing interest in languages that support some form of type-level computation including Haskell's type classes [Hall et al. 1996] and indexed type families [Kiselyov et al. 2010]. However, there is a fundamental tension between type-level computation and generative type abstraction, at least in the latter's more flexible forms. To summarize very briefly, the conflict is this:

- To maximize re-use and convenience, it is very desirable for the implementor to be able to treat the abstract type \( A \) and its concrete representation type \( C \) as synonymous – we call this flexible type generativity.

- However, given type-level function \( F \) the result of \((FA)\) and \((FC)\) may differ, so \( A \) and \( C \) cannot be synonymous.

Resolving this conflict is the subject of this paper. Specifically our contributions are:

- We show in Section 2 that the naive combination of type generativity and non-parametric type-level features can violate soundness; a problem that already manifests in the Glasgow Haskell Compiler, and affects not only type-level non-parametric functions, but also other forms of non-parametric constructs, such as generalized algebraic datatypes (GADTs) [Cheney and Hinze 2003; Hinze et al. 2002; Peyton Jones et al. 2006; Xi et al. 2003].

- We formalize a solution to this problem that reconciles flexible type generativity and non-parametric type functions in Section 3. Our language, FC\(_2\), builds on GHC's existing core language, System FC [Sulzmann et al. 2007], which supports erasable type-level coercions. The key ingredient in our solution is to employ kinds decorated with roles to mark the distinctions that type contexts may make.

- We prove that FC\(_2\) programs are type safe, provided that user axioms and definitions give rise to consistent axiom sets (Section 4).

- We give sufficient conditions for showing the consistency of an axiom set. For our proofs, we introduce a rewrite system for types that is novel in two dimensions: rewriting (i) is role-sensitive, and (ii) need not be strongly normalizing (Section 4.2).

- We present a Haskell-specific result: we show how Haskell source programs that may involve non-parametric type contexts and flexible type generativity can be translated to yield provably consistent FC\(_2\) axiom sets. (Section 5)

- Our core language is an improvement of System FC since it permits safe flexible type generativity, but also unsaturated type functions. Perhaps surprisingly, our language is additionally a significant simplification of the original System FC, which removes several of the original coercion constructs that we have identified to be encodable (even in the original System FC). We discuss these differences in Section 6.2.

For the sake of concreteness we build our presentation around Haskell and System FC, since this setting allows us to demonstrate our points with real code, instead of using a hypothetical \(\lambda\)-calculus. However, we stress that our work is applicable whenever flexible type generativity and non-parametric type-level features are combined. For example, the very same issues could arise in extensions of the ML module system. We discuss related work in Section 7.
Some of the technical material has been elided from this version of the paper for space. Although this paper is self-contained, more details are available at http://www.cis.upenn.edu/~sweirich/newtypes.pdf.

2. The problems with generative type abstraction

Generative type abstraction is an extremely useful mechanism for enforcing abstraction barriers and for refining interfaces. To see this benefit, let us consider the Haskell incarnation of type generativity, namely newtype definitions. In Haskell, the programmer may declare a newtype Age, with concrete representation type Int, thus:

newtype Age = MkAge Int

The implementor uses the “data constructor” MkAge to coerce an Int to an Age, and pattern matching to effect the inverse coercion. For example:

addAge :: Age -> Int -> Age
addAge (MkAge a) n = MkAge (a+n)

The client, in contrast, can be prevented from making such conversions, by using the module system to hide the MkAge constructor:

module AgeModule( Age, addAge, ... ) where
  -- Age definition and implementation

2.1 Coercion lifting

In describing MkAge, we wrote “data constructor” in quotes because although it behaves in many ways like a data constructor, its cost model is different. Specifically, a newtype definition guarantees that the abstract type really is represented by the concrete type, so the runtime conversion cost is zero. That would not be true if Age were instead declared with data instead of newtype.

So Age can be coerced to an Int, and vice versa, for free—i.e. without runtime cost—because it is an Int. Notationally, we say that Int \sim Age, where we use \sim for type equality\footnote{There are too many sorts of “\sim”!}. But what about, say, Maybe Age?

data Maybe a = Nothing | Just a

Obviously, Maybe Age should be freely coercible to Maybe Int, because the two are represented identically. Alas, in Haskell 98 one would have to write

cvtAge :: Maybe Age \rightarrow Maybe Int
cvtAge t = mapMaybe (\(\lambda\)(MkAge a) \rightarrow a) t

This is unsavory for several reasons: (a) it is tedious for the programmer; (b) it may be difficult to implement the necessary “map” function; (c) it is hard for the compiler to eliminate a runtime call to the map function (let alone to guarantee to do so) especially if the map function is recursive. As an example of (b) consider the mapping function for the type T shown below, with co- and contravariance, and higher kinds:

data T a f = T1 a |
| T2 (a \rightarrow Int) |
| T3 (f (T a f) (T a f)) |

These difficulties are frustrating, because we know that, say, T Age Maybe is represented identically to T Int Maybe. So, let us imagine a hypothetical extension of Haskell that provides lifted coercions; that is, it implements the following rule:

Coercion lifting: if for two types \(\varphi\) and \(\psi\) we have \(\varphi \sim \psi\) (for example, if they are the abstract and concrete types of a newtype declaration), then \(T \varphi \sim T \psi\) for any type constructor \(T\).

In fact this extension is not so hypothetical, because such a rule is the basis of the so-called “newtype deriving” feature implemented by GHC. It is not difficult to use newtype deriving to define an identity function whose type is

cvt :: \forall \{c\}. c \rightarrow c

ML supports coercion lifting even more directly: within a structure the abstract and the representation types are considered entirely interchangeable. For example in Standard ML one might say

signature AGE = sig
  type age
  val addAge : age \rightarrow int \rightarrow age...
end

structure AgeModule :: AGE = struct
  type age = int
  fun addAge a n = a+n

Inside AgeModule the two types are synonymous, and so addAge need not convert in either direction.

However, the point of this paper is that the innocuous and obvious-seeming “lifting” of type identities becomes unsound when combined with type-level computation, as we show in the next section.

2.2 Type-level computation

One very popular extension to Haskell is that of Generalized Algebraic Data Types (GADTs) [Peyton Jones et al. 2006], with which we shall assume the reader is somewhat familiar. In GHC one could declare a GADT with two nullary constructors like this:

data K a where
  KAge :: K Age
  KInt :: K Int

Now, consider these definitions (using cvt from Section 2.1):

kint :: K Age
get :: K Age \rightarrow String
kint = cvt KInt
get KAge = "Age"

Since get’s type signature declares that its argument is of type K Age, the patterns in get are exhaustive. But consider the call (get kint). It is patently well typed, yet the pattern match in get will fail (recall that cvt is operationally the identity function), and if the compiler assumed otherwise a runtime crash could result.

In the last few years we have gone beyond GADTs, by extending GHC with type-level functions [Chakravarty et al. 2005a,b; Kiselyov et al. 2010]. The reader is urged to consult these papers for motivated examples of type functions, but for the purposes of this paper we content ourselves with a small but contrived example:

type family F a :: *

type instance F Age = Char

type instance F Int = Bool

Here the type function F maps the type Age to the type Char, but it maps Int to Bool.

However, the existence of such a type-level function threatens not just pattern exhaustiveness but type soundness itself. Consider the type Bool. This type is equivalent to F Int by the equation
for \( F \); and by coercion lifting that should be equivalent to \( F \) \( Age \); and that is equivalent to \( Char \) by the other equation for \( F \). Altogether we can coerce \( Bool \) to \( Char \), which is obvious nonsense.

What went wrong? Maybe it should be illegal for a type function to behave differently on two coercible types, such as \( Age \) and \( Int \)? But in fact Haskell programmers often use \texttt{newtypes} precisely so that they can give a different type-class instance (for comparison, say) for \( Age \) than for the underlying \( Int \). Type functions are no different; indeed, they are often introduced as an “associated type” of a type class [Kiselyov et al. 2010], and hence, just as the type class distinguishes between the abstract and concrete type, so must the type function.

How else might we fix the problem? Perhaps, in the definition of coercion lifting we should not allow \( T \) to range over type functions such as \( F \) or \( GADTs \) such as \( K \)? Indeed we should not, but that is not enough. Consider

\[
\begin{aligned}
data \ TF \ a &= \texttt{MkTF} (F \ a) \\
\text{Now, should coercion lifting allow us to coerce } TF \ Age \text{ to } TF \ Int? \text{ Obviously not! Otherwise we could write}
\end{aligned}
\]

\[
\begin{align*}
to :: & \texttt{Bool} \to \ TF \ Int \quad \text{from :: } TF \ Age \to \ Char \\
to b &= \texttt{MkTF} b \\
\text{and now the composition } &\texttt{from} \circ \texttt{cvt} \circ \text{to is well-typed (via coercion lifting) but obviously unsound. So in the the definition of coercion lifting, as well as type functions and GADTs we must exclude data types like } TF \text{ that use their type arguments non-parametrically.}
\]

2.3 Summary

At this point it should be clear that a naive combination of:

- type-level dispatch, whether by \( GADTs \) or by type functions
- unrestricted coercion lifting

simply does not work. This interaction was far from obvious to us initially, and indeed GHC has a well-documented type-soundness bug\(^2\) that arises directly from this unforeseen interaction. Yet both type-level dispatch and coercion lifting (suitably restricted) are valuable. The purpose of this paper is to show how they may be soundly combined.

This problem is important not only because it arises in GHC (which is our main source of motivation), but also because the same issues will arise in any type system that combines type-level dispatch and coercion lifting. Haskell is the first programming language that has pushed the type system far enough for these problems to arise in practice, but others may do so in the future. However, these different languages may expose the coercions between abstract and concrete types in different ways, possibly explicitly (as in Haskell) or implicitly (as in ML), in a way that is intimately connected with type inference. To avoid the complications associated with type inference, in this paper, we focus on an intermediate language, in which (runtime-erasable) coercions are explicit. Whether they come directly from the source program, or from elaboration by the type checker, is secondary.

3. The FC\(_2\) language: codes versus types

Our goal is to resolve the tension between generative types, which allow programmers to express the intent that two types have identical representations, and type functions, which can distinguish two types even if they have the same underlying representation.

The inspiration for our solution comes from encodings of generic programming in dependently typed languages [Benke et al. 2003; Dybjer 2000]. These encodings use “codes” to represent types as a form of data. For example, the code \( \texttt{IntCode} \) may be the code for the type \( \texttt{Int} \). Nonparametric functions branch on codes and thus have a different type from parametric functions. In this context, we can view \( \texttt{Age} \) and \( \texttt{IntCode} \) as two different codes that both map to the type \( \texttt{Int} \). However, this encoding requires significant syntactic overhead as we must have a code for every type and must explicitly map codes to types when they are used to classify expressions.

However, the important distinction between codes and types is that they have different definitions of equality. In the encoding above, the codes \( \texttt{Age} \) and \( \texttt{IntCode} \) are different codes, but their interpretations are equal types. Therefore, we use this idea in FC\(_2\) to define roles, which support different notions of equality for the same data. In this language, \( \texttt{Age} \) and \( \texttt{Int} \) are distinct when viewed at role \texttt{code} but equal when viewed at role \texttt{type}. Code equality is used to reason about the meaning of type-indexed functions and is finer-grained than type equality, which is used to determine which type coercions are safe. Importantly, FC\(_2\) distinguishes type functions by what equivalences they respect. Parametric functions respect the role \texttt{type}, whereas functions that distinguish between \( \texttt{Age} \) and \( \texttt{Int} \) do not.

3.1 FC\(_2\): an overview

These ideas are best explained in terms of an intermediate language that exposes the differences between the code and type roles and makes explicit the uses of the two kinds of equality mentioned above. Thus, the remainder of this section describes FC\(_2\), our new variant of System FC [Sulzmann et al. 2007]—a model of the intermediate language used in GHC. As such, it is expressive enough to capture indexed type functions, \texttt{newtype} and \texttt{newtype deriving}, GADTs, existential and nested datatypes, and much more.

Figure 1 summarizes the syntax of FC\(_2\), which, at the term level (\( e \)), is just the polymorphic lambda calculus with two extensions. First, FC\(_2\) provides polymorphic datatypes, introduced by data constructors \( K \). These datatypes are eliminated using a \texttt{case} construct that should be familiar from Haskell or ML-style functional programming—we describe how datatypes and \texttt{case} are typechecked in Section 3.5.

Second, FC\(_2\) includes first-class proofs of type equality that witness safe coercions introduced during compilation. Programs in FC\(_2\) can abstract over coercions reflecting a particular type equality (written \( \Lambda e : \varphi \sim \varphi' \), e) pass a coercion as an argument to such a function (written \( \gamma e \gamma )\), and use a coercion to cast a term from one type to another (written \( e \gamma )\). These explicit coercions, written \( \gamma \), make typechecking FC\(_2\) programs syntax-directed—that is, the syntax of an FC\(_2\) term encodes its typing derivation. Why is this important? The idea is that the compiler’s front end performs perhaps-complex type inference on the source program, and records the proofs generated by inference directly in the syntax of the FC\(_2\) intermediate language. The optimiser transforms FC\(_2\) terms, perhaps radically. At any point one can check the consistency of the resulting FC\(_2\) program using a simple, fast, syntax-directed typechecker; this consistency check has proven to be an extremely powerful aid to getting the compiler right. It is just as easy to find the type of an arbitrary FC\(_2\) term, an ability that is used extensively inside GHC.

Figure 2 shows the typing rules for the terms of FC\(_2\). The first five rules, \texttt{EEVAR} through \texttt{ETAPP}, are completely standard. We defer explanation of the remaining rules until we build up technical machinery having to do with FC\(_2\)’s kind-level distinction between codes and types and the rules by which explicit coercions can themselves be combined. We describe these aspects of FC\(_2\) next.
### Figure 1. Syntax

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\eta ::= \star</td>
<td>\kappa \to \eta$</td>
</tr>
<tr>
<td>$R ::= C</td>
<td>\top$</td>
</tr>
<tr>
<td>$\kappa ::= \eta/R$</td>
<td>kind and role</td>
</tr>
<tr>
<td>$H ::= T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>datatypes</td>
</tr>
<tr>
<td></td>
<td>functions/newtypes</td>
</tr>
<tr>
<td></td>
<td>arrow</td>
</tr>
<tr>
<td></td>
<td>equality</td>
</tr>
<tr>
<td>$\varphi, \sigma, \psi ::= a</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>variables</td>
</tr>
<tr>
<td></td>
<td>constants</td>
</tr>
<tr>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td>polymorphism</td>
</tr>
<tr>
<td>$\gamma ::= c \varphi</td>
<td>\varphi</td>
</tr>
<tr>
<td></td>
<td>assumption</td>
</tr>
<tr>
<td></td>
<td>reflexivity</td>
</tr>
<tr>
<td></td>
<td>symmetry</td>
</tr>
<tr>
<td></td>
<td>transitivity</td>
</tr>
<tr>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td>injectivity</td>
</tr>
<tr>
<td></td>
<td>polymorphism</td>
</tr>
<tr>
<td></td>
<td>instantiation</td>
</tr>
<tr>
<td>$e ::= x</td>
<td>\lambda x : \sigma. e</td>
</tr>
<tr>
<td></td>
<td>variable</td>
</tr>
<tr>
<td></td>
<td>abstraction</td>
</tr>
<tr>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td>type abstraction</td>
</tr>
<tr>
<td></td>
<td>type application</td>
</tr>
<tr>
<td></td>
<td>data constructor</td>
</tr>
<tr>
<td></td>
<td>case analysis</td>
</tr>
<tr>
<td></td>
<td>proof abstraction</td>
</tr>
<tr>
<td></td>
<td>proof application</td>
</tr>
<tr>
<td></td>
<td>coercion</td>
</tr>
<tr>
<td>$brs ::= { K_1 \Rightarrow e_1 }_{i \in 1..n}$</td>
<td>branches</td>
</tr>
<tr>
<td>$bnd ::= a : \kappa</td>
<td>H : \eta</td>
</tr>
<tr>
<td></td>
<td>type variable</td>
</tr>
<tr>
<td></td>
<td>type constant</td>
</tr>
<tr>
<td></td>
<td>coercion</td>
</tr>
<tr>
<td></td>
<td>term variable</td>
</tr>
<tr>
<td></td>
<td>data constructor</td>
</tr>
<tr>
<td>$\Gamma ::= \cdot</td>
<td>\Gamma, bnd$</td>
</tr>
<tr>
<td>$\Delta ::= \cdot</td>
<td>a : \kappa, \Delta$</td>
</tr>
<tr>
<td>$\rho ::= e</td>
<td>\varphi</td>
</tr>
<tr>
<td>$\Theta ::= \cdot</td>
<td>\sigma, \Theta</td>
</tr>
<tr>
<td></td>
<td>empty</td>
</tr>
<tr>
<td></td>
<td>expression type</td>
</tr>
<tr>
<td></td>
<td>type variable</td>
</tr>
<tr>
<td></td>
<td>equality</td>
</tr>
<tr>
<td>$v ::= \lambda \sigma : \sigma. e</td>
<td>\Lambda a : \kappa. e</td>
</tr>
<tr>
<td>$\tau ::= T</td>
<td>\to</td>
</tr>
</tbody>
</table>

### Figure 2. Typing rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \sigma \in \Gamma \vdash x : \sigma$</td>
<td>EEVER</td>
<td>$\Gamma \vdash x : \sigma$</td>
</tr>
<tr>
<td>$\Gamma, x : \sigma_1 \vdash e : \sigma_2$</td>
<td>EEABS</td>
<td>$\Gamma; x : \sigma_1 \vdash \lambda x : \sigma_1. e : \sigma_1 \to \sigma_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash e_2 : \sigma_2$</td>
<td>EEAPP</td>
<td>$\Gamma \vdash e_1 ; e_2 : \sigma_1$</td>
</tr>
<tr>
<td>$\Gamma, a : \kappa \vdash e : \sigma$</td>
<td>ETABS</td>
<td>$\Gamma; a : \kappa.e : \forall a : \kappa. \sigma$</td>
</tr>
<tr>
<td>$\Gamma \vdash \varphi : \kappa$</td>
<td>ETAPP</td>
<td>$\Gamma \vdash e \varphi : \sigma[a \mapsto \varphi]$</td>
</tr>
<tr>
<td>$T : \pi \to * \in \Gamma$</td>
<td>EDATACON</td>
<td>$\Gamma \vdash c ; \pi \vdash \sigma : * / T$</td>
</tr>
<tr>
<td>$K : \forall \pi. \forall \Theta. T \pi \in \Gamma \vdash \Gamma$</td>
<td>ECASE</td>
<td>for each $K_i \in \text{consts}(T)$</td>
</tr>
<tr>
<td>$K_i : \forall \pi. \forall \Theta. \Lambda \psi \in \Gamma$</td>
<td>ECABS</td>
<td>$\psi[\sigma \mapsto \varphi] = \forall \Theta. T \pi$</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \forall \Theta. \sigma$</td>
<td>ECAPP</td>
<td>$\Gamma \vdash \varphi : \kappa$</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \sigma$</td>
<td>ECOERC</td>
<td>$\Gamma \vdash e : \sigma_1 \to \sigma_2 \in \ast / T$</td>
</tr>
</tbody>
</table>

### Figure 3. Kinding

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1 \preceq R_2$</td>
<td>C$\leq$T</td>
<td>$\Gamma \vdash \varphi : \kappa$</td>
</tr>
<tr>
<td>$R \subseteq R$</td>
<td>RREFL</td>
<td>$\Gamma \vdash \alpha : \eta / R_1 \in \Gamma \vdash \Gamma \vdash \Gamma \vdash \alpha : \eta / R_2$</td>
</tr>
<tr>
<td>$\Gamma \vdash a : \eta / R_1$</td>
<td>PVAR</td>
<td>$\Gamma \vdash \varphi : \kappa$</td>
</tr>
<tr>
<td>$H : \eta \in \Gamma \vdash \Gamma$</td>
<td>PCONST</td>
<td>$\Gamma \vdash : H : \eta / R$</td>
</tr>
<tr>
<td>$\Gamma \vdash \varphi_1 : (\eta_1 / R_2 \to \eta_2) / R_1$</td>
<td>PAPP</td>
<td>$\Gamma \vdash \varphi_1 \varphi_2 : \eta_2 / R_1$</td>
</tr>
<tr>
<td>$\Gamma \vdash \varphi_2 : \eta_1 / \text{min}(R_1, R_2)$</td>
<td>PALL</td>
<td>$\Gamma \vdash \varphi_1 \varphi_2 : \eta_2 / R_1$</td>
</tr>
<tr>
<td>$\Gamma, a : \kappa \vdash \varphi : \ast / R$</td>
<td>PNI</td>
<td>$\Gamma \vdash \forall a : \kappa. \varphi : \ast / R$</td>
</tr>
<tr>
<td>$\Gamma \vdash \cdot : \cdot$</td>
<td>PCONS</td>
<td>$\Gamma \vdash \varphi, \overline{\varphi} : \kappa, \overline{\kappa}$.</td>
</tr>
</tbody>
</table>
3.2 FC₂ types and kinds

Types in FC₂ are classified by pairs \( \kappa \) of the form \( \eta / R \), where the kind \( \eta \) ensures (as usual) that types are well-formed structurally, and the role \( R \) determines which comparison must be applied. Codes (which distinguish \( \text{Age} \) and \( \text{Int} \)) have role \( \text{C} \), whereas types (which identify them) have role \( \text{T} \). This syntax is summarized at the top of Figure 1, while the kinding rules are in Figure 3.

The distinction between codes and types allows us to give informative kinds to type constructors:

- The \text{Maybe} type (Section 2.1) has kind \( \times / \text{T} \rightarrow \times \), indicating that \text{Maybe} treats its argument parametrically.

- The types \( \text{K}, \text{F}, \) and \( \text{TF} \) (Section 2.2) all use type indexing and therefore have kind \( \times / \text{C} \rightarrow \times \).

These kinds in turn support the key insight of this paper: it is only safe to lift coercions through functions with parametric kinds. So \( \text{Maybe Age} \rightarrow \text{Maybe Int} \) holds but \( \text{TF Age} \not\rightarrow \text{TF Int} \).

The rules for constructing coercions are given in Section 3.3, but first we must deal with the kinding rules for types. The critical rule is \( \text{PA} \) in Figure 3, which deals with type application. It is quite conventional except for the treatment of roles. Consider these type declarations:

\[
\begin{align*}
data \text{TF1} a &= \text{MkTF1} (\text{F (Maybe a)}) \\
data \text{TF2} a &= \text{MkTF2} (\text{Maybe (F a)}) \\
data \text{TF3} a &= \text{MkTF3} (\text{Maybe (Maybe a)}) \\
\end{align*}
\]

In the first two cases, type variable \( a \) is used non-parametrically (intuitively, under a type function), so \( \text{TF1} : \times / \text{C} \rightarrow \times \), and similarly for \( \text{TF2} \). In contrast, \( \text{TF3} \) treats its argument parametrically, so \( \text{TF1} : \times / \text{T} \rightarrow \times \). The unusual “\( \min \)” in rule \( \text{PA} \) achieves these kindings by combining the role of the context and the argument role of the function to get the role of the argument.

In fact, \( \min \) is just the least upper bound induced by an inclusion relation \( \text{C} \leq \text{T} \) on roles (see the top of Figure 3). This inclusion makes explicit the fact that code equality implies type equality but not vice-versa. For example, consider

\[
\begin{align*}
data \text{TF4} a &= \text{MkTF4} (\text{F a}) a \\
\end{align*}
\]

Here \( a \) is used both non-parametrically (as an argument of \( \text{F} \)) and parametrically (as a plain argument of \( \text{MkTF4} \)). So \( a \) has role \( \times / \text{C} \), which makes sense; just because a type \( \text{can} \) be analyzed does not mean that it \emph{must} be.

Despite these non-standard kinds, the types of FC₂ are mostly standard. Codes \( \varphi \) and types \( \sigma \) are drawn from the same syntax (see Figure 1), although we use two different metavariables as a reminder of the intended role. The type language includes type variables \( \kappa \), type constants \( H \), applications \( \varphi_1 \varphi_2 \), and polymorphic types \( \forall \gamma : \kappa . \sigma \).

Type constants, \( H \), include datatypes \( T \), and type functions \( F \). For the most part, datatypes and type functions are treated uniformly, but there are two important distinctions:

- Datatypes must be injective, while type functions need not be. Injectivity is important because equalities between applications of injective functions imply equalities between their arguments; see rule \( \text{CNTHT} \) in Section 3.3.

- Datatypes are inhabited by values, but type functions are not—there are no values \( v \) with types that are headed by \( F \). (There are coerced values with such types, but no raw values.) This distinction is key to the definition of consistency in Section 4.1.

Newtypes are not inhabited by raw values, so we treat them like type functions, ranged over by \( F \). Unlike type functions, however, newtypes are injective at role \( C \)—after all, the essence of generativity is that newtypes create a fresh constant—but for now we will not take advantage of that fact, leaving it for discussion in Section 6.1.

The set of type constants also includes the familiar arrow type constructor \( (\rightarrow) \), and a kind-indexed family of constructors \( (\sim) \), which construct functions that abstract over coercions. The kinds of these constants are:

\[
\begin{align*}
(\rightarrow) & : \star / \text{T} \rightarrow \star / \text{T} \rightarrow \star \\
(\sim) & : \eta / \text{C} \rightarrow \eta / \text{C} \rightarrow \star / \text{T} \rightarrow \star
\end{align*}
\]

(We discuss in Sections 3.3 and 3.6 why the kind of \( (\sim) \) must require role \( \text{C} \) for its first two arguments.) Type constants are generally applied prefix, but for these two constants we define infix syntactic sugar:

\[
\sigma \rightarrow \sigma' \quad \equiv \quad (\rightarrow) \sigma \sigma'
\]

\[
(\varphi_1 \sim \varphi_2) \Rightarrow \sigma \quad \equiv \quad (\sim) \varphi_1 \varphi_2 \sigma
\]

In the latter case, because the syntactic sugar lacks the \( \eta \) annotation, we only use this notation in contexts where the kind of \( \varphi_1 \) and \( \varphi_2 \) is irrelevant. Rule \( \text{ECABS} \) follows this convention—it shows that this family of type constructors is used to give a type to terms of the form \( \Lambda c : \varphi_1 \sim \varphi_2 . e \), which abstracts over an equality proof in the body \( e \). Note that this rule applies only to code equalities; abstraction over type equalities is not needed for compilation of Haskell and permitting it, while straightforward, would require extra syntactic complexity.

Figure 3 defines the kinding rules for FC₂ using judgments of the form \( \Gamma \vdash \varphi : \kappa \). Here, the context \( \Gamma \) maps type variables to their kind/role pairs and type constants and type functions to their kinds. This judgment relies on a judgment \( \Gamma \vdash \sigma \) (shown in the extended version) that checks that contexts are well formed. These checks ensure that constants \( (\rightarrow) \) and \( (\sim) \) have the right kinds, that data constructors and term variables have well-formed types and that coercion axioms introduce equalities between well-formed codes and types.

Rule \( \text{PConst} \) allows a type constant to play either role. Although a type constant may be given role \( \text{C} \), its \emph{kind} may well involve arguments with role \( \text{T} \); for example, see the signature for \( (\rightarrow) \) above. It follows that closed types can play both roles, independently of whether they contain newtypes or type functions. For example \( (\text{Int, Age}) : \star / \text{C} \) and \( (\text{Int, Age}) : \star / \text{T} \) both hold.

Together these kinding rules implement a subrelation that includes codes into the language of types:

**Lemma 1.** If \( \Gamma \vdash \varphi : \eta / \text{C} \) then \( \Gamma \vdash \varphi : \eta / \text{T} \).

On the other hand, types have only one kind regardless of their role:

**Lemma 2.** If \( \Gamma \vdash \varphi : \eta_1 / \text{R}_1 \) and \( \Gamma \vdash \varphi : \eta_2 / \text{R}_2 \) then \( \eta_1 = \eta_2 \).

Note that the kinding judgment, like term typing, is syntax-directed (see the remarks in Section 3.1). In particular, the role component \( \text{R} \) of the \( \kappa \) in this judgment is treated as \emph{input} to the algorithm, and the \( \eta \) is an \emph{output}—the context in which \( \varphi \) is used determines what the role should be. The only interesting case from this perspective is \( \text{PA} \), in which \( \varphi_1 \) must be checked first to obtain \( \text{R}_2 \) so that the minimum of \( \text{R}_1 \) and \( \text{R}_2 \) can be passed as an input when checking \( \varphi_2 \).

3.3 Coercions and equality

In FC₂ a \emph{coercion} is a proof term that witnesses the equality of two types. Coercions are used to change the type of a term, thus (Figure 2):

\[
\begin{align*}
\Gamma \vdash e : \sigma_1 \\
\Gamma \vdash \gamma : \sigma_1 \sim \sigma_2 \in \star / \text{T} \\
\end{align*}
\]

\[
\frac{
\Gamma \vdash e : \sigma_1 \\
\Gamma \vdash \gamma : \sigma_1 \sim \sigma_2 \\
\sigma_1 \rightarrow \sigma_2}
\Gamma \vdash e : \sigma_2 \\
(ECoerce)
\]

Here, \( \gamma \) is a coercion witnessing the equality \( \sigma_1 \sim \sigma_2 \) in role \( \text{T} \); given that \( e \) has type \( \sigma_1 \), we can use \( \gamma \) to let us treat the term as hav-
More generally we permit axiom schemes. For example, the source language declaration

**type instance** \( F \) (Maybe \( a \) ) = (a, a)

would create the axiom scheme

\[ \text{axF3:} (a: \ast/C). F \) (Maybe \( a \) ) \sim (a, a)/C \]

In general, as shown in Figure 1, the context \( \Gamma \) includes bindings of the form \( c: \Delta. \varphi_1 \sim \varphi_2/R \) for coercion axioms. The metavariable \( \Delta \) stands for a type variable context: a list of quantified type variable bindings of the form \( \alpha : \kappa \). The same binding form is used both for axioms introduced at top level, and (with empty \( \Delta \)) for local assumptions introduced in ECABs (Figure 2).

Of course it is important to know that the top-level axioms are consistent—it would be unsafe to assert that \( \text{Bool} \sim \text{Char}/\text{T} \), for example. Section 4 gives a sufficient set of conditions for ensuring that source programs generate consistent axioms.

Next, we need a way to compose coercions together to construct other coercions. Our goal is to have rules that allow the creation of composite coercions such as:

- \( \text{List mkAge} : \text{List Age} \sim \text{List Int/T} \)
- \( \text{List axF2} : \text{List} (F \text{Int}) \sim \text{List} \text{Bool/T} \)

On the other hand, the coercion formation rules should disallow the formation of a coercion of the form \( \gamma : \text{Age} \sim F \text{Int/T} \), which creates the unsoundness described in Section 2.2.

Figure 1 gives the syntax of coercion terms, \( \gamma \), and Figure 4 gives their typing rules. Coercions are typechecked using the judgment: \( \Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/R \), which asserts that the type of a coercion \( \gamma \) is an equality proposition \( \varphi_1 \sim \varphi_2 \in \eta/R \). This proposition in turn implies that \( \varphi_1 \) and \( \varphi_2 \) both have kind \( \eta \) and are equal in role \( R \). It is technically convenient to include \( \eta \) in the syntax of the judgement to enforce that both types have the same kind. However, this component is not always relevant, so we sometimes omit the \( \in \) part, as we have done in the examples above.

Rule \( \text{CASSM} \) instantiates an axiom scheme with types \( \psi \), using an auxiliary judgment \( \Gamma \vdash \psi : \pi \) defined at the bottom of Figure 3 to ensure that each variable is instantiated with a type of the matching kind and role. The notation \( \pi : \pi \) zips together a list of type variables and a list of kinds to create a type variable context \( \Delta \). These two lists must have the same length for the notation to be well-defined. The notation \( \varphi[\pi \mapsto \psi] \) applies a multi-substitution of the types \( \psi \) for each of the corresponding variables in the list \( \pi \).

Rule \( \text{CREFL} \) shows that any type \( \varphi \) can be lifted to a reflexive coercion \( \varphi \), while \( \text{CSYM} \) and \( \text{CTRANS} \) add symmetry and transitivity, ensuring that equality is an equivalence relation. The rules \( \text{CAPP} \) and \( \text{CALL} \) extend equality compatibility over applications and polymorphic types; their structure is analogous to the corresponding kinding rules in Figure 3. Rule \( \text{CAPP} \) is particularly important, because it implements the key coercion lifting idea we discussed in Section 2.1, using kinds to prevent the formation of the bogus coercion

\[ \text{FmkAge} : F \text{ Age} \sim F \text{ Int/T} \]

To see why, recall that \( F \) has kind \( \ast/C \rightarrow \ast \), but the \( \text{FmkAge} \) axiom holds only at role \( T \)—type equalities cannot be lifted through code functions. Another example of a coercion that is correctly rejected by the application rule (because of the kind of \( \sim \)) is

\[ (\sim) \text{FmkAge Int } \sigma \]

This coercion proves \( \text{Age} \sim \text{Int} \Rightarrow \sigma \sim (\text{Int} \sim \text{Int}) \Rightarrow \sigma \), an equality that could be used to introduce a bogus assumption that \( \text{Age} \sim \text{Int/C} \) and satisfy it with reflexivity for \( \text{Int} \).

As well as composing coercions to witness the equality of bigger types, it is also essential to do the reverse: to decompose equalities over complex types to give equalities of simpler types.
Decomposition is required by FC\textsubscript{2}'s operational semantics (Section 3.6), and it also makes the language usefully more expressive. Rule CINST allows equalities between polymorphic types to be instantiated. The other, more important decomposition rule is CNthT, which decomposes the application of a datatype constructor to arguments. For example, given a coercion $\gamma : \text{List Int} \sim \text{List } a/T$ we can use $\text{n}th \ 0 \ \gamma$ to conclude that $\text{Int} \sim a/T$.

The soundness of this rule depends on the fact that datatypes are injective. In general, type functions are not, and hence CNthT is restricted to datatypes $T$.

In rule CNthT, the notation $T[x]$ abbreviates the multi-application $((T \varphi_1) \ldots \varphi_n)$ for $\varphi_1 \ldots \varphi_n$ in $\varphi$. In the conclusion of the rule, the notation $\text{n}th \ k \ \psi$, accesses the $k$th element of the sequence of types, and $\text{n}th \ k \ \pi$, accesses the kind of the $k$th variable binding. Both of these notations are undefined if $k$ is not less than the length of the sequence. The sequence of types $\pi$ is determined by the kind of the datatype, using the following notation.

**Definition 3.** We define $\pi \rightarrow \eta$ by induction on $\pi$ as follows:

\begin{align*}
\cdot & \rightarrow \eta = \eta \\
\kappa, \pi \rightarrow \eta & = \kappa \rightarrow (\pi \rightarrow \eta)
\end{align*}

The coercion judgment satisfies a number of sanity checking properties. First, coercions are between well-formed types.

**Lemma 4 (Coercion regularity).** If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \kappa$ then $\Gamma \vdash \varphi_1 : \kappa$ and $\Gamma \vdash \varphi_2 : \kappa$.

Next, each coercion proof is for one pair of codes/types.

**Lemma 5 (Unique types).** If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/R$ and $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta'/R$ then $\varphi_1 = \varphi_1'$ and $\varphi_2 = \varphi_2'$ and $\eta = \eta'$.

Finally, equality for Cs is a refinement of that for Ts; that is, code equality implies type equality, but not vice versa.

**Lemma 6.** If $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/C$ then $\Gamma \vdash \gamma : \varphi_1 \sim \varphi_2 \in \eta/T$.

### 3.4 Coercion lifting

The application rule CAPP described in the previous section allows us to lift equalities through arbitrary type constructors; that is, for all datatypes $T$ of kind $*/T \rightarrow *$, we have a coercion that shows $T \text{Age} \sim T \text{Int} \in */T$.

In fact, this notion of coercion lifting is not restricted to datatypes (like List), but is available for more general contexts. More precisely, given an arbitrary type $\sigma$ with free variable $a$ of kind $*/T$, we can also create a coercion $\sigma[a \rightarrow \text{Age}] \sim \sigma[a \rightarrow \text{Int}] \in */T$.

We create such coercions with the lifting operation. This operation replaces type variables by coercions in types to produce a new coercion, relying on the fact that the syntax of types is a subset of the syntax of coercions.

**Definition 7 (Lifting Operation).** Define the lifting operation, written $\varphi[a \rightarrow \gamma]$, by induction on $\varphi$.

\[
\begin{align*}
    a[a \rightarrow \gamma] & = \gamma \\
b[a \rightarrow \gamma] & = b \\
H[a \rightarrow \gamma] & = H \\
(\varphi \psi)[a \rightarrow \gamma] & = (\varphi[a \rightarrow \gamma]) \cdot (\psi[a \rightarrow \gamma]) \\
(\forall b : \kappa)[a \rightarrow \gamma] & = \forall b : \kappa. (\sigma[a \rightarrow \gamma])
\end{align*}
\]

The lifting operation produces a result as long as the role of the lifted coercion matches the role of the type variable in the type.

**Lemma 8 (Lifting).** If $\Gamma, a : \eta \vdash \sigma : \kappa$ and $\Gamma \vdash \gamma : \varphi \sim \varphi' \in \eta/R$, then $\Gamma \vdash \sigma[a \rightarrow \gamma] : \sigma[a \rightarrow \varphi] \sim \sigma[a \rightarrow \varphi'] \in \kappa$.

In other words, if $a$ were used in some indexed context in $\sigma$, that is, if its role were $C$, then we would not be able to lift the coercion $\text{Age} \sim \text{Int} \in */T$ in $\sigma$. We generalize lifting to replace multiple type variables simultaneously in the obvious way, with notation $\sigma[\pi[1] \rightarrow \pi[2]]$.

### 3.5 Pattern matching and datatypes

FC\textsubscript{2} includes a formalization of recursive datatypes. These datatypes include all GHC extensions to standard datatypes: empty datatypes, nested datatypes, existential types, first-class polymorphism and GADTs. Both datatypes $T$ and data constructors $K$ must be declared in a context $\Gamma$ before they can be used. For example, using the syntax for $\Gamma$ in Figure 1, the declarations for the data constructors of List are:

- List $: \ast/T \rightarrow \ast$
- Nil $: (\ast : \ast/T). \text{List } a$
- Cons $: (\ast : \ast/T). a \rightarrow \text{List } a \rightarrow \text{List } a$

What about GADTs? Here's an example in Haskell:

```haskell
data Rep a where
Rint :: Rep Int
Rlist :: Rep (List a)
```

Although a Haskell programmer writes the data constructors of a GADT with non-parametric result types, in the internal type system it is simpler in the formalization for the result type of a data constructor to take the form $(T \ a_1 \ldots a_n)$, where the $\sigma$ are the type parameters, using equality constraints to express the indexing, thus:

```haskell
Rep : \ast/C \rightarrow \ast
Rint : (a : \ast/C). (a \sim \text{Int}) \Rightarrow \text{Rep } a
Rlist : (a : \ast/C). (\forall b : \ast/C). (a \sim \text{List } b) \Rightarrow \text{Rep } b \rightarrow \text{Rep } a
```

Notice here that Rep's kind expresses that its argument is an index (role C) rather than a parameter (role T). In fact, the C role for variable $a$ falls out naturally because $a$ appears as argument to the $(\sim)$ constructor (in the type of $\text{Rint}$ and $\text{RList}$), which in turn requires it to have role C.

More generally, we use the notation of *telescopes* [de Bruijn 1991] to conveniently express the types of data constructors. Figure 1 defines a telescope $\Theta$ like this:

$$\Theta ::= \cdot | a : \kappa, \Theta | \varphi_1 \sim \varphi_2, \Theta | \sigma, \Theta$$

A telescope is like a mini-context: a list of type variable bindings, equality propositions (between codes only) and types that classify each argument of the data constructor. (Note that a type variable context $\Delta$ is also a telescope.) We define the syntactic sugar $\forall \Theta. \sigma$ as follows:

**Definition 9 (Telescope syntactic sugar).**

\[
\begin{align*}
    \forall \cdot & \sigma = \sigma \\
    \forall (a : \kappa, \Theta). \sigma & = \forall a : \kappa. (\forall \Theta. \sigma) \\
    \forall (\varphi_1 \sim \varphi_2, \Theta). \sigma & = (\varphi_1 \sim \varphi_2) \Rightarrow (\forall \Theta. \sigma) \\
    \forall (\sigma', \Theta). \sigma & = \sigma' \rightarrow (\forall \Theta. \sigma)
\end{align*}
\]

At each use of a data constructor (rule EDATACON), we check that the declaration of a data constructor is of the form $K : \forall \pi. \pi. (\forall \Theta. \pi \Theta \pi)$, where $\pi. \pi$ lists the type parameters of the datatype. $\Theta$ describes the types of the arguments to the data constructor, and $T \pi$ is syntactic sugar for the application of $T$ to those parameters (i.e. $(T \ a_1 \ldots a_n)$). For example, the List type has a single parameter $a : */T$, the constructor Cons has telescope $(a, \text{List } a \ldots)$, and constructs type List $a$. Likewise, the Rep type has the parameter $a : */C$, the telescope for RList is $(b : */C, a \sim \text{List } b, (\text{Rep } b) \ldots)$, and the result type is Rep $a$. 
3.6 Operational semantics: pushing coercions

The operational semantics of FC₂ is based on a small-step call-by-name operational semantics for a polymorphic lambda calculus. Although this language includes explicit type abstraction and application (as well as explicit coercion proof abstraction and application), types and coercions are not relevant at run time. All such abstractions and applications, as well as uses of coercions e₁ → γ may be erased prior to execution and so impose no run-time costs.

As alluded to above, this operational semantics preserves coercion proofs, which allows us to establish type safety using standard syntactic proofs of progress and preservation (described in the next section). The most important rules of the operational semantics are those that “push” coercions when they appear in active positions so that they do not interfere with reduction. Figure 5 shows the relevant pushing rules. (The complete rules of the operational semantics are listed in the extended version.) In some of the push rules, the coercion must be checked to constrain the types that it proves equal. This checking happens in a global context Σ that contains only the definitions of data constructors, data types, and coercion axioms. Note that these checks are not actually necessary at run-time, in a consistent context these checks will always succeed.

The first three rules show how in an application of a coerced abstraction, the term steps to a new application, where the coercion has been decomposed into a coercion for the body of the abstraction, and a coercion of the argument. For example, consider SSPUSH. Here, the γ is a coercion between some function types σ₁ → σ₂ and σ₂ → γ. The rule uses nth and sym to decompose γ into two coercions, one from σ₂ → σ₁ (the order is reversed to account for contra-variance) and one from σ₁ → σ₂. These new coercions can be pushed to the body of the lambda and the function argument, exposing the reduction. Rules SSTPUSH and SSCPUSH work analogously.

Note that SSCPUSH justifies the kind of (≈), which requires that the first two arguments be codes. If we had assigned (≈) the parametric kind /T → /T → *, then the coercions γ₀ and γ₁ in the rule would both be type coercions. However, type coercions cannot be composed with γᵢ to form a code coercion, which is the role required for the right hand side of the rule to typecheck.

The last rule, SSKPUSH, pushes the coercion of a data constructor in the scrutinee position of a case expression into coercions of the arguments of the data constructor. The arguments of a data constructor, notated ρ, can either be an expression, a type, or a coercion.

```
Γ ⊢ ρ : Θ
Γ ⊢ e : σ  Γ ⊢ ρ : Θ
Γ ⊢ e, ρ : (σ, Θ)
```

If the declared type of the data constructor K is ∀π : .∀Θ, T π, and the type parameters are σ, then we know that the arguments can be typed using Γ ⊢ ρ : Θ[σ ↦ σ']. (see Figure 6). However, pushing the coercion changes the type parameters to be σ', so the new arguments must have type Θ[σ ↦ σ'], with the coercion generated by lifting.

These new arguments are produced by coercing the list of arguments ρ with the coercion generated by lifting as described above. (The notation σ[σ ↦ nth γ] means that variable aᵢ is lifted to coercion nth aᵢ γ.) Once this coercion has been defined by lifting, we use it to coerce the list of arguments of the data constructor with the following operation.

**Definition 10.** Define argument coercion ρ ⊢ γ by induction on ρ:

- (e, ρ) ⊢ γ₁ → γ₂ = (e ⊢ γ₁), (ρ ⊢ γ₂)
- (φ, ρ) ⊢ ∀a : κ. γ₂ = ρ ⊢ γ₂
- (γ, ρ) ⊢ (γ₁ ⊨ γ₂) ⇒ γ₃ = (sym γ₁ : γ₂), ρ ⊢ γ₃

Argument coercion coerces a list of arguments as described by the following lemma.

**Lemma 11.** If Γ ⊢ ρ : Θ and Γ ⊢ γ : ∀Θ, σ → ∀Θ′, σ' ∈ ∗/T then Γ ⊢ (ρ ⊢ γ) : Θ′
4.1 Preservation and progress

The preservation proof for FC2 is standard, relying on the usual regularity and substitution lemmas for the various judgement forms. For space reasons, we omit those definitions here and instead refer the reader to the extended version.

**Theorem 12** (Preservation). If $\Gamma \vdash e_1 : \sigma$ and $e_1 \rightsquigarrow e_2$ then $\Gamma \vdash e_2 : \sigma$.

The progress theorem holds only for closed, consistent contexts. A context is closed if it does not contain any term variable bindings. We use the metavariable $\Sigma$ for closed contexts.

The definition of consistency and the canonical forms lemma (necessary to show the progress theorem) are both stated using the notions of uncoerced values and their types, value types. Value types include all types except those that are headed by a variable $a$ or type function/newtype $F$. Formally, we define values $v$ and value types $\tau$, with the following grammar:

$$\tau ::= T \mid (\rightarrow) \mid (\sim) \mid \forall a : \kappa. \tau \mid \tau_1 \circ \tau_2$$

$$v ::= \lambda x : \sigma. e \mid \lambda a : \kappa. e \mid \Lambda c \vdash e \rightsquigarrow \tau_1. e \mid K \varphi \rho$$

The canonical forms lemma tells us that the shape of a value is determined by its type:

**Lemma 13** (Canonical Forms). Say $\Sigma \vdash v : \sigma$. Then $\sigma$ is a value type. Furthermore,

1. If $\sigma = \sigma_1 \rightarrow \sigma_2$ then $v$ is $\lambda x : \sigma_1. e$ or $K \varphi \rho$.
2. If $\sigma = \forall a : \kappa. \sigma'$ then $v$ is $\Lambda a : \kappa. e$ or $K \varphi \rho$.
3. If $\sigma = (\varphi_1 \sim \varphi_2)$ then $v$ is $\Lambda c : \sigma_1 \rightsquigarrow \sigma_2. e$ or $K \varphi \rho$.
4. If $\sigma = T \varphi_1$ then $v$ is $K \varphi \rho$.

In FC2, not all irreducible forms are values. Evaluation can also produce a coerced value of the form $v \rightsquigarrow \gamma$, which erases to a value when coercions are dropped. To prove the progress theorem, we must reason about what sort of coercion $\gamma$ could be so that we can appropriately apply the "push" rules in Figure 5. In the statement of the progress theorem, because $\gamma$ coerces the value $v$, we know (by ECOERCE and canonical forms) that $\gamma : \tau \rightarrow \sigma$—the left type is a value type $\tau$. Consistency of the axiom set assures us that if $\sigma$ is also a value type, it must have the same head form.

**Definition 14** (Consistency). A context $\Gamma$ is consistent if whenever $\Gamma \vdash \gamma : \tau_1 \rightarrow \tau_2 \in \eta/\Pi$ it is the case that

1. If $\tau_1 = T \varphi_1$ then $\tau_2 = T \varphi_2$.
2. If $\tau_1 = (\rightarrow) \varphi_1$ then $\tau_2 = (\rightarrow) \varphi_2$.
3. If $\tau_1 = (\sim) \varphi_1$ then $\tau_2 = (\sim) \varphi_2$.
4. If $\tau_1 = \forall a : \kappa. \sigma_1$ then $\tau_2 = \forall a : \kappa. \sigma_2$.

Putting these observations together, we obtain:

**Theorem 15** (Progress). If $\Sigma$ is consistent and $\Sigma \vdash e_1 : \sigma$ and $e_1$ is not a value $v$ or a coerced value $v \rightsquigarrow \gamma$, then there exists an $e_2$ such that $e_1 \rightsquigarrow e_2$.

4.2 Determining consistency

Although the previous subsection gives a definition of when contexts are consistent, it does not provide any mechanism for determining whether a set of axioms leads to a consistent context.

This subsection defines sufficient conditions (written $\text{Good } \Gamma$) for establishing context consistency—these conditions are not the only way to show consistency (they are not necessary) but they are permissive enough to cover the axioms generated by compilation of type family declarations and newtype definitions.

---

3 Note that all forms of value type must include partial applications of data constructors.
Reflexivity of the rewrite relation is crucial for joinability of type applications with multiple arguments. Consider the axiom
\[ c : F \text{ Int} \leadsto (F \text{ Int}, F \text{ Int})/C \]
and the following two types:
\[ \varphi_1 = T \ (F \text{ Int}) \ (F \text{ Int}, F \text{ Int}) \]
\[ \varphi_2 = T \ (F \text{ Int}, F \text{ Int}) \ (F \text{ Int}) \]
We'd certainly like \( \varphi_1 \) and \( \varphi_2 \) to be reducible to a common reduct (after all, there exists a coercion term between them), but the first argument of \( T \) in \( \varphi_1 \) “lags behind” compared to the first argument of \( T \) in \( \varphi_2 \), whereas the second argument in \( \varphi_1 \) “advances ahead” compared to the second argument in \( \varphi_2 \). Reflexivity allows us to freeze the reduction of the second argument in \( \varphi_1 \) while reducing the first, and similarly for \( \varphi_2 \)—which allows us to join the two types.

4.2.2 Soundness and completeness of the rewrite relation

We now give sufficient conditions on contexts which ensure that the rewrite relation of Figure 7 is sound and complete with respect to coercibility. Hence, these are sufficient conditions for consistency.

**Definition 18 (Good contexts).** We have Good \( \Gamma \) when the following conditions hold:

1. All axioms in \( \Gamma \) are of the form \( c : \Delta, F \varphi \leadsto \psi/R \). Let \( \pi = \text{dom}(\Delta) \) and \( \pi \) be the corresponding kinds. For every well kinded \( \varphi \) and result types \( \varphi' \), such that \( \Gamma \vdash (\varphi[\pi \mapsto \varphi']) \leadsto \varphi'' \in R/R \) it must be \( \varphi'' = \varphi[\pi \mapsto \varphi'] \) for some \( \varphi' \) with \( \Gamma \vdash \varphi \leadsto \varphi' \). The second condition is simply a strong non-overlapping condition.

2. There is no overlap between axioms. For each \( F \varphi \) there exists at most one prefix \( \varphi_1 \) of \( \varphi \) such that there exist \( \psi, \varphi' \) where \( \Gamma \vdash c; \psi : F \varphi_1 \leadsto \psi \in \kappa \). This \( c \) is unique for every matching \( F \varphi_1 \).

The first condition above restricts the declared arguments of type functions to behave like patterns. It merely states that all the reductions that can possibly happen when we substitute types for \( \Delta \) do not involve reductions from \( \varphi \) but only reductions inside those substituted types. This condition is a generalized (and more lenient) form of the current GHC restrictions on type family declarations, which only allow value types or variables as \( \varphi \). The second condition is simply a strong non-overlapping condition. Interestingly, the first condition on the arguments to type functions restricts the kind that a type function may have. For example, recall the axioms for \( F \) from Section 2.2:

\[ \text{axFl:} F \text{ Int} \leadsto \text{Bool}/C \]
\[ \text{axF2:} F \text{ Age} \leadsto \text{Char}/C \]

For this to be a Good axiom set, the kind of \( F \) must be \( \star/C \rightarrow \star \) because the newtype \( \text{Age} \) is irreducible only in \( C \).

In the rest of this section, we sketch the proof that our conditions are sufficient for consistency. Soundness is a straightforward proof.

**Theorem 19 (Soundness).** If Good \( \Gamma \) and \( \Gamma \vdash \varphi_1 \leadsto \varphi_2 \in \kappa \) then there is some \( \gamma \) such that \( \Gamma \vdash \gamma : \varphi_1 \leadsto \varphi_2 \in \kappa \).

The three key lemmas of the completeness proof are that joinability is preserved under application and substitution and a local diamond property of rewriting.

**Lemma 20 (Application).** If Good \( \Gamma \) and \( \Gamma \vdash \varphi \leadsto \varphi_1 \in \eta, \Gamma \vdash \varphi \leadsto \varphi_2 \in \eta/\min(R_1, R_2) \) then \( \Gamma \vdash \varphi_1 \leadsto \varphi_2 \in \eta_1 / R_2 \). Then \( \Gamma \vdash \varphi_1 \leadsto \varphi_2 \in \eta_1 / R_2 \).

**Theorem 21 (Local diamond property).** If Good \( \Gamma \) and \( \Gamma \vdash \varphi \leadsto \varphi_1 \in \kappa \) and \( \Gamma \vdash \varphi \leadsto \varphi_2 \in \kappa \) then there exists a \( \varphi_3 \) such that \( \Gamma \vdash \varphi_1 \leadsto \varphi_3 \in \kappa \) and \( \Gamma \vdash \varphi_2 \leadsto \varphi_3 \in \kappa \).

A different novelty of our approach compared to previous work [Suzlmann et al. 2007] is that establishing the completeness theorem, and therefore type soundness, does not depend on strong normalization. Our definition of Good \( \Gamma \) that (i) forces the declared type family arguments to behave like patterns, and (ii) imposes strong (stronger than the previous work) non-overlapping conditions, is sufficient to show completeness.

With this new approach, a programmer can be confident that, even in the presence of possibly non-terminating type functions, if the compiler can show that the program is well-typed then the program will not crash. Although type checking is undecidable in general without strong normalization, there may be many programs that the compiler can actually type check, and this approach shows that those programs are type sound.

5. Compilation from source Haskell

In the previous sections we have informally presented the translations of Haskell source features such as datatype, type family, and newtype declarations into FC2. We summarize them here:

\[
\begin{align*}
\text{data} & \ T \ : \ \Delta \ \text{where} \ K_i : \sigma \\
& \ T : \ \forall \Delta, \text{st}, K_i : \Delta, \sigma \\
\text{type family} & \ F : \ \eta/C \rightarrow \eta \\
& \ F : \ \eta/C \rightarrow \eta \\
\text{type instance} & \ F \varphi = \psi \\
& \ cF : \ \Delta, F \varphi \leadsto \psi \in \eta/C \\
\text{newtype} & \ M, \ \Delta, \ M \mapsto \eta/C : \ M \mapsto \eta/C \\
& \ M : \ \Delta, \ \text{st}, \ MMK \ : \ \Delta, M, \ \Delta \mapsto \eta \in \star/T
\end{align*}
\]

In addition, uses of the data constructor of a newtype, both in terms and patterns, are translated to a use of the corresponding coercion in the obvious way. The important parts of these definitions are: (i) type families accept code arguments, (ii) type instances give rise to code equalities, and (iii) newtype definitions give rise to type equality axioms. The bindings generated in this way can be easily checked for well-formedness. If, in addition, the resulting context is Good (see Section 4.2)—the only possible problem is the potential to generate overlapping type instance declarations—then the context is consistent, which in turn guarantees type safety.

Notice that the source language features in this translation have been already annotated with their kinds. This is a reasonable assumption. Prior to type inference, which translates a source declaration to an FC2 binding, we must have determined the kinds involved in the declarations. For the purposes of this paper, we assume that the kinds are given—in practice they would be the output of a kind inference process, potentially guided by the user to disambiguating role information and higher-order kinds.
Newtype deriving  

Generative type abstraction is achieved in Haskell using the `newtype deriving` mechanism, which allows type classes to be automatically lifted to new types. For instance, we may write

```haskell
newtype Age = MkAge Int deriving Eq
```

The type class `Eq a` is a standard class in Haskell (signature, in ML terminology) that defines one method, `eq::a → a → Bool`. For the type checker, a type class is nothing but a record type containing the method `eq`. The `deriving` line automatically generates an implementation of `Eq` `Age` from a pre-existing instance `Eq Int`. This can be done by simply applying a coercing Eq `Int` `Eq` `Age` to the old record. It is straightforward to construct this coercion: lift the `Age` `Int` axiom from the `newtype` definition over the `Eq` constructor and apply symmetry. Importantly, this lifting is safe because `Eq` has a parametric kind of the form `⋆/T` → `⋆`. However, if a type class has an indexed kind of the form `⋆/C` → `⋆`, `newtype deriving` is no longer sound. For example

```haskell
class C x where
  op :: F x → x
instance C Int where ...
newtype Age = MkAge Int deriving deriving C
```

It is unsound to lift the `C` `Int` implementation to a `C` `Age`, precisely because `x` has role `C`. The documented GHC bug mentioned earlier arises precisely as the result of such an unsound lifting over a non-parametric type class.

6. Discussion

6.1 Relaxing decomposition

Recall the coercion decomposition rule, `CnTHT`, from Figure 4. This rule allows us to deconstruct an equality of the form `Γ ⊢ γ : T ∼ T' ∈ η/T`. In effect, it asserts that data constructors are injective. The rule is important because it is used in the operational semantics to ensure subject reduction. However, the decomposition rule may be somewhat restrictive for some Haskell source programs. Consider the following:

```haskell
newtype M a = MkM (Maybe [a])
data Eq a b where EQ :: Eq a a
  f :: Eq (M a) (M b) → a → b
f EQ x = x
```

Pattern matching against the `EQ` constructor introduces a coercion between `M a` and `M b`, which cannot be decomposed using the `CnTHT` rule to obtain `a ∼ b`, so this program cannot be typed. Nevertheless, we know that `M` is injective, because `M a` is defined to be equal to `Maybe [a]`, which is clearly injective.

On the other hand, the following `newtype` is not injective.

```haskell
type instance G a = Char
newtype N a = MkN (G a)
```

Here, it is possible to derive `Γ ⊢ γ : N Int ∼ N Char ∈ */T`, using the axiom for `G`, even though `Int` is not coercible to `Char`.

It turns out that `newtype` definitions are always injective with respect to code equality, but they might not be injective with respect to type equality (as illustrated by the two examples above). Thus it would be sound and potentially useful (but not necessary for type soundness) to introduce yet another decomposition rule for `newtype` definitions that takes advantage of injectivity with respect to codes (we use metavariable `N` newtypes):

```haskell
Γ ⊢ γ : N ∼ N ∈ η/C
N ∼ ∈ η → η ∈ C
η/R = nth k π
Γ ⊢ nth k γ : nth k ∼ nth k ∈ η/R
```

The only subtle part of this rule is that `R` is not related to `R'`, since the equality `Γ ⊢ γ : N ∼ N ∈ η/C` is a C-equality (and `min(C, R') = C`).

Arguably, decomposition for injective type functions is also desirable, were we able to effectively specify and check that property.

6.2 Other technical differences of FC₂ from System FC

The intermediate language FC₂ described in this paper is a significant modification of System FC [Sulzmann et al. 2007] due to the introduction of codes. However, FC₂ also makes a number of technical simplifications:

- The original System FC presentation includes coercion kinds, `σ₁ ∼ σ₂`. The original coercion language includes three additional constructs, one to coerce coercions, and two more to decompose coercion kinds. By treating `(ϕ₁ ∼ ϕ₂) ⇒ ϕ₂` as the application of a constructor (`(ϕ₁)`) we no longer need any of these constructs in the operational semantics, nor have we identified any uses of these constructs that are not encodable.

- The operational semantics rules of FC₂ in Figure 5 not only use simpler coercion constructs, but are also expressed without need for substitutions, contrary to their original FC versions.

- FC₂ replaces the FC coercions `left` and `right`, which decomposed arbitrary type applications, by `nth`, which decomposes only the application of a datatype constructor. This allows us to lift a tiresome restriction in in System FC, namely that type functions were required to be saturated. Type family saturation was necessary in FC, in order to prevent the decomposition of equalities as `F a ∼ Maybe [a]` via `left` or `right`. Allowing unsaturated functions increases the expressiveness of FC₂, because we can now abstract over type functions, and also opens new directions for future research on type inference in the presence of unsaturated type functions.

- However, using `nth` instead of `left` or `right` does carry a small price. Generalizing the example from Section 6.1, should this program be well typed?

```haskell
data Eq a b where EQ :: Eq a a
  f :: Eq (p q) (r s) → q → s
f EQ x = x
```

To type it we must decompose a proof that `p q ∼ r s` to get a proof that `q ∼ s`, which is possible but not `nth` can do.

Some other differences are presentational:

- System FC used a common syntax for types and coercions, which is a convenient pun, but has turned out to be more confusing than helpful. In FC₂ we use a distinct syntax for types and coercions (Figure 1).

- In FC₂ we define top-level axiom schemes `c : Δ ∼ ϕ₁ ∼ ϕ₂/R` directly, and fully instantiate them at every occurrence with the form `c ∼` (Figure 1). System FC instead defined a top-level axiom scheme as an equality between polymorphics, thus `c : ∀Δ.ϕ₁ ∼ ∀Δ.ϕ₂`. Here again FC is confusing (but not wrong) so in FC₂ we opt for telling the story more directly, albeit with slightly more syntax. Moreover the kinding rules for `∀ (P AliN CALL)` can insist that the body of the forall has kind `⋆` as is conventional.
• Using telescopes in the treatment of datatypes simplifies the operational semantics rules but is also (only slightly) more expressive: The types of data constructors do not have to have their quantified variables preceding their coercion and term arguments. Instead, telescopes allow arbitrary interleavings.

7. Related work

Previous work on System FC [Sulzmann et al. 2007] discusses a significant amount of related work, in typed languages with explicit proof witnesses [Licata and Harper 2005; Shao et al. 2005], or in calculi that support coercions [Breazu-Tannen et al. 1991]. Below, we present related work in generativity and abstraction, type-indexed constructs and the separation between codes and types.

Generativity, abstraction, and module systems Generativity and abstraction has been studied extensively in the context of ML module systems [Milner et al. 1997]. Russo shows how generativity in module systems is connected to existential quantification [Russo 1999] and Dreyer [2005] has studied this connection in the presence of recursive modules. Montagu and Rémy [2009] refine this connection by introducing “open” existential types. Rossberg [2008] uses flexible generativity to explain ML-style module sealing.

Type abstraction can be understood in terms of dynamic name generation [Rossberg 2003; Vytiniotis et al. 2005], which can re-establish abstraction properties in languages with dynamic type analysis. Neis et al. [2009] prove a parametricity theorem in this setting. In addition, they use a translation from polymorphism to generative types to establish the parametric behavior of certain functions although they work in a non-parametric language.

Although many of these languages support type generativity and non-parametric features, they do not exhibit the soundness problems described in the paper, mainly due to the absence of type-level type dispatching. Nevertheless, the techniques developed in the aforementioned related work would be valuable in the formal study of the parametricity properties of FC2.

Type-indexed types Although many systems for generic programming support dynamic computation based on types, very few systems allow the structure of types to be destructed to produce other types. However, such facility is often necessary to describe the type of generic programs. For example, Harper and Morrisett include a Typerec operator to their typed intermediate language AML [Harper and Morrisett 1995], to describe type-directed optimizations. (They credit NuPRL’s mechanism of ‘Universe Elimination’ in NuPRL as the inspiration for this operation [Constable 1982; Constable and Zlatan 1984].)

To support generic programming in source languages, Hinze, Jeuring and Löh added Type-Indexed Datatypes [Hinze et al. 2002] to the Generic Haskell front end. In later work, Chakravarty et al. [2005b] introduced associated data families in GHC, which are type-indexed datatypes associated with type class instances. Extending this work, they later introduced associated type synonyms [Chakravarty et al. 2005a], which are proper type-level functions with instances associated with type class instances. Currently, the source language of GHC also supports standalone type-level type functions, often referred to as indexed type families [Kiselyov et al. 2010; Schrijvers et al. 2008], a feature that we have extensively used in our presentation.

Codes, types, and interpretations Our distinction between codes and types—and our terminology—is inspired by similar notions in intuitionistic type theory [Benke et al. 2003; Dybjer 2000; Martin-Löf 1975]. There, types (sets) are constructed as the recursive interpretation of codes, which inhabit inductively constructed code universes. A newtype definition can be viewed as giving rise to a new code, inhabiting an open universe of codes, and whose interpretation coincides with the interpretation of its definition.

Languages based on dependent type theory, such as Agda [Bove et al. 2009] or Coq [The Coq Team], naturally offer type-level computation to construct types, but they allow elimination of codes only, not types. Therefore, they do not exhibit the same soundness problem, as the expressiveness of these languages can readily enforce the distinction between types and codes. The disadvantage is the extra programming verbosity of explicit definitions and interpretations of codes. To better support generic programming, the dependently-typed language Epigram [Chapman et al. 2010] identifies types with their code universes.

The LX language [Crary and Weirich 1999] also uses universe constructions to solve problems with type-directed compilation. When the type translation in a compiler pass is not the identity then type dispatch must be compiled to code dispatch (so the generated code can dispatch on source types instead of target types). The interpretation of codes is then the type translation. To support universes, LX includes datakinds (for codes) and primitive recursive functions over datakinds (for their interpretation at types). In LX, source types Age and Int would be mapped to definable codes AgeCode and IntCode, and would be accompanied by an interpretation function such that interp(AgeCode) equals Int and interp(IntCode) equals Int. Therefore, the problem with generativity would not show up in that context. If one wanted to solve the problem in this paper along the LX lines, one would have to translate source Haskell types to void types that stand in as codes and handle the interp() function as any other type function. This function, as well as interpreting the codes as types, would have to be accompanied with suitable congruence axioms, like interp(T t) ~ T (interp(t)). Explicitly introducing these axioms means that coercions would be significantly more verbose. Our system dispenses with an explicit interp() function by conveniently using the roles in the judgements to determine whether we wish to derive an equality between codes or between their interpretations.

8. Conclusions and future work

In this paper we have identified a problem for the safe interaction of flexible type generativity and type-level computation. We have proposed a solution that distinguishes between indexed and parametric type contexts, by extending the language of kinds, and formalized the solution in the FC2 language. We have several avenues for future research in mind, which we outline below.

Source language technology We would like to work on ways to expose the FC2 expressive features to programmers. Specific directions are: enriching the kind declarations with the ability to declare parametric or indexed type-level constructs, introducing type family injectivity annotations, extending kind inference with roles, and extending type inference to support unsaturated functions using the more sophisticated kinds.

Enriching the universes of codes with terms We are currently working on enriching the universe of codes with constants or functions drawn from the term syntax, such as data constructors, in order to enable direct dependently-typed programming in Haskell.

More roles Lemma 6 asserts that the equivalence classes induced by T-equality are refined by C-equality. However, our approach readily extends to arbitrary lattices of roles with gradually more refined equivalence classes as we move down the ⪯ relation.

Consider, for example, a lattice with new roles C1 and C2, each lying between C and T, but incomparable with each other. These two roles model partial knowledge of newtype equalities. For example, perhaps C1 identifies Age and Int but distinguishes...
a different newtype $\text{Moo}$ from its definition $\text{Bool}$. Conversely, $C_2$ can identify $\text{Moo}$ and $\text{Bool}$ and distinguish $\text{Age}$ and $\text{Int}$. These more precise roles could be used to give more precise types to nonparametric functions by identifying exactly which newtype equivalences they do and do not respect. However, we have not yet explored the practical implications of this precision.

Furthermore, one could explore adding a role above $T$. Type functions with kind $\star \to \star$, are not necessarily parametric in $F$. For example, because $G$ below does not match newtypes it may be assigned either kind $\star \to \star$ or $\star / C \to \star$.

$axG1 : G \to \text{Bool} / C$

$axG2 : G \to \text{Char} / C$

To distinguish truly parametric functions from those like $G$, we could add a role $P$, induced by the relation that equates all types of the same kind. As this equivalence is coarser than $T$, it could not (and should not) participate in coercions. However, this new role would provide a yet more descriptive kind for type functions.

Acknowledgements

We thank Brent Yorgey and the attendees of the Type System Wrestling sessions at MSR Cambridge for many useful discussions. We wrote this paper using the Ott tool. This work was partially supported by grants from DARPA (CSSG) and NSF (0910786) and a Sloan Foundation fellowship.

References


