University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

1-28-2012

Giving Haskell a Promotion

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

Brent A. Yorgey
University of Pennsylvania

Julien Cretin
INRIA

Simon Peyton Jones
Microsoft Research--Cambridge

Dimitrios Vytiniotis
Microsoft Research--Cambridge

See next page for additional authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. TLDI'12, January 28, 2012, Philadelphia, PA, USA. Copyright © 2012 ACM 978-1-4503-1120-5/12/01

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/625
For more information, please contact repository@pobox.upenn.edu.

Giving Haskell a Promotion

Abstract
Static type systems strive to be richly expressive while still being simple enough for programmers to use. We describe an experiment that enriches Haskell's kind system with two features promoted from its type system: data types and polymorphism. The new system has a very good power-to-weight ratio: it offers a significant improvement in expressiveness, but, by re-using concepts that programmers are already familiar with, the system is easy to understand and implement.

Disciplines
Computer Sciences

Comments
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. TLDI'12, January 28, 2012, Philadelphia, PA, USA. Copyright © 2012 ACM 978-1-4503-1120-5/12/01

Author(s)
Stephanie Weirich, Brent A. Yorgey, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and Jose P. Magalhaes

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/625

Giving Haskell a Promotion

Brent A. Yorgey
Stephanie Weirich
University of Pennsylvania
{byorgey,sweirich}@cis.upenn.edu

Julien Cretin
INRIA
julien.cretin@inria.fr

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

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

Abstract

Static type systems strive to be richly expressive while still being simple enough for programmers to use. We describe an experiment that enriches Haskell's *kind* system with two features promoted from its *type* system: data types and polymorphism. The new system has a very good power-to-weight ratio: it offers a significant improvement in expressiveness, but, by re-using concepts that programmers are already familiar with, the system is easy to understand and implement.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures, Polymorphism; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Design

Keywords Haskell, promotion, kinds, polymorphism

1. Introduction

Static type systems are the world's most successful application of formal methods. Types are simple enough to make sense to programmers; they are tractable enough to be machine-checked on every compilation; they carry no run-time overhead; and they pluck a harvest of low-hanging fruit. It makes sense, therefore, to seek to build on this success by making the type system more expressive without giving up the good properties we have mentioned.

Every static type system embodies a compromise: it rejects some "good" programs and accepts some "bad" ones. As the dependently-typed programming community knows well, the ability to express computation at the type level can improve the "fit"; for example, we might be able to ensure that an alleged red-black tree really has the red-black property. Recent innovations in Haskell have been moving in exactly this direction. Notably, GADTs [24] and type families [25] turn the type system into a (modest) programming language in its own right.

But, embarrassingly, type-level programming in Haskell is almost entirely *untyped*, because the kind system has too few kinds ($\star$, $\star \to \star$, and so on). Not only does this prevent the programmer from properly expressing her intent, but stupid errors in type-level programs simply cause type-level evaluation to get stuck rather than properly generating an error (see §2). In addition to being too *permissive* (by having too few kinds), the kind system is also too *restrictive*, because it lacks polymorphism. The lack of kind polymorphism is a well-known wart; see, for example, Haskell's family of Typeable classes (§2.5), with a separate (virtually identical) class for each kind.

In this paper we describe a design that fixes these problems, and we sketch its implementation in GHC, our Haskell compiler. Our design is inspired by Conor McBride's Strathclyde Haskell Enhancement (SHE) preprocessor [19], which automatically promotes datatypes to be *datakinds*. Our work goes well beyond SHE by also introducing *kind polymorphism*, and integrating these two new features with convenient source syntax and full type (and kind) inference.

From a type-theoretic point of view, this paper has little new to say: full-spectrum dependently typed languages like Coq [31] or Agda [23] are more expressive still. But, as we discuss in more detail in §7, these languages are in some ways *too* powerful: they have a high barrier to entry both for programmers (who want to write programs in natural and straightforward ways) and implementors (who have to worry about efficiently executing these programs). Instead, we start from the other end: we carefully extend a state-of-the-art functional programming language with features that appear in dependently-typed languages. Our primary audience is the community of Designers and Implementors of Typed Languages, to whom we offer a big increase in expressive power for a very modest cost in terms of intellectual and implementation complexity.

Specifically, our contributions are:

- We extend Haskell with a rich kind system, including kind polymorphism. Suitable value and type constructors are automatically promoted to become type and kind constructors, respectively (we explain precisely which constructors can be lifted in §3.3). We show, by example, that these modest extensions offer a large gain in expressiveness (§2).
- We formalize an explicitly-typed intermediate language, System $F_C^{\uparrow}$ (pronounced "FC-pro"), that embodies the new kind system (§3). $F_C^{\uparrow}$ is no toy: we have implemented it in GHC as a modest extension of GHC's existing intermediate language, System F_C. Our extension to this intermediate language guides

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TLDI'12, January 28, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1120-5/12/01…$10.00

our overall design—going further would require much more significant effort.

- GHC uses $F_C^{\uparrow}$ throughout its optimization phases, and each optimization must ensure that the transformed program is well-typed. So the metatheory of $F_C^{\uparrow}$ is of practical importance, especially subject reduction (§4). In addition to subject reduction we prove progress, which ensures that coercions can be erased at runtime without compromising type safety.
- We describe the modifications to GHC's type inference engine (§5) and the source language extensions (§6) to support the new features. (Haskell is a large language, so these sections are necessarily informal.)

Finally, we discuss related work (§7) as well as directions for future research (§8).

Our goal throughout is to provide maximum gain for minimum pain. One could go further in terms of supported features, but we believe we have found a sweet spot. The new extensions are fully implemented, and will soon be released as part of GHC. The experience of GHC's users will then inform our understanding of possible future developments in the design space of dependently-typed programming languages.

2. Typed type-level programming

We begin with an informal motivation for the extensions of our system.

2.1 Promoting datatypes

Consider the following standard implementation of length-indexed vectors in Haskell:

```
data Zero
data Succ n
data Vec :: ⋆ → ⋆ → ⋆ where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Succ n)
```

We declare two empty datatypes, Zero and Succ, to serve as "type-level values", and use a generalized algebraic datatype (GADT) [24] to define a type Vec whose first argument is the type of its elements and whose second argument is a type-level natural number reflecting the length of its values.

However, for a language like Haskell with a strong, static type system, this example is rather embarrassing: it is *untyped*! The type parameter to Succ has kind $\star$, which gives us no indication that we intend for it to always be a type-level representation of a natural number. The same is true of the second parameter to Vec. Essentially, $\star$ is serving as the single type in a "uni-typed" type system.[1] We are not prevented from writing nonsensical types such as Succ Bool or Vec Zero Int.

It would be much better to be able to write

```
data Nat = Zero | Succ Nat
data Vec :: ⋆ → Nat → ⋆ where
  VNil  :: Vec a Zero
  VCons :: a → Vec a n → Vec a (Succ n)
```

This is the sort of thing we could write in a language with a full-spectrum dependent type system, such as Agda [23]. We have declared a normal datatype Nat, and Vec takes as its second argument a *value* of type Nat. Our intention for the arguments of Vec is now clear, and writing Vec Zero Int will rightly yield a type error.

[1] Well, not quite uni-typed, because we have arrow kinds, but close.

In our new system *the above example is now valid Haskell*! In the declaration of Vec we can see that Nat is used as a kind. Following SHE, we achieve this effect by automatically *promoting* (suitable) datatypes to become kinds; and their data constructors (Zero and Succ in this case) to become type-level data. In effect this gives the Haskell programmer the ability to declare their own kinds, by declaring datatypes and promoting them up a level.

These datakinds may also be used in GHC's indexed type families (*i.e.* type functions). For example,

```
type family     Plus (a :: Nat) (b :: Nat) :: Nat
type instance Plus Zero      b          = b
type instance Plus (Succ a)  b          = Succ (Plus a b)
```

In general, a suitable (§3.3) datatype declaration

```
data T = C1 T1 | C2 T2 | ...
```

not only defines a type constructor T and data constructors C1, C2, ..., but *also* a kind T, a type constructor C1 of kind T1 → T (using the promoted kinds T and T1), and similarly for the other constructors.

2.2 Resolving namespaces

Although in principle a simple idea, in practice this approach leads to an awkward naming problem for source Haskell programs, because Haskell allows type constructors and data constructors to have the same name:

```
data T = T Int
```

If we see "T" in a type, does it refer to the type T (of kind $\star$) or the promoted data constructor T (of kind Int → T)? In such cases, we adopt the following notation: plain T means the type constructor, while 'T, with a prefixed single quote, denotes the promoted data constructor. So the fully explicit version of the foregoing example looks like this:

```
data Nat = Zero | Succ Nat
data Vec :: ⋆ → Nat → ⋆ where
  VNil  :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)
```

Where the promotion is unambiguous, the quote may be omitted.

We do not require (or allow) the quote notation in kinds, because there is no ambiguity to resolve. We do not allow promoting types which are themselves indexed by a promoted data constructor (§3.3), so anything in a kind other than $\star$ can only be a promoted type.

2.3 Kind polymorphism for promoted types

We also allow promoting *parameterized* datatypes, such as lists. For example:

```
data HList :: [⋆] → ⋆ where
  HNil  :: HList '[]
  HCons :: a → HList as → HList (a : as)
```

Here we have declared HList, the type of *heterogeneous* singly-linked lists. The index of HList is a list of types which records the types of the elements (created by promoting the list datatype). For example,

```
HCons "Hi" (HCons True HNil) :: HList (String : Bool : '[])
```

is a heterogeneous list containing two values of different types. Haskell allows the syntactic sugar $[a, b]$ for the explicit list $(a : b : [\,])$, and we support the same sugar in types, thus:

```
HCons "Hi" (HCons True HNil) :: HList '[String, Bool]
```

The prefix quote serves, as before, to distinguish the type-level list from the list *type* in, say $reverse :: [a] \to [a]$.

If this promotion is to be allowed, what is the kind of the promoted data constructor '(:)? Since the data constructor (:) is type-polymorphic, the promoted constructor '(:) must be *kind*-polymorphic:

$$'(:) :: \forall \mathcal{X}.\ \mathcal{X} \to [\mathcal{X}] \to [\mathcal{X}]$$

where $\mathcal{X}$ is a kind variable.

We have by now seen the most significant modifications to the kind language. Haskell kinds now include the kind of types of values, written $\star$, other base kinds formed from promoted datatypes, arrow kinds, and polymorphic kinds. This means that we can only promote data constructors with types analogous to one of these kinds. In particular, we will not promote data constructors with constrained types (including GADTs) or higher-order type polymorphism. We return to this issue in §3.3.

2.4 Kind polymorphism for datatypes

Kind polymorphism is useful independently of promotion. Consider the following datatype declaration:

$$\mathbf{data}\ \mathsf{TApp}\ f\ a = \mathsf{MkTApp}\ (f\ a)$$

This code has always been legal Haskell, but, lacking kind polymorphism, the kind of TApp was *defaulted* to $(\star \to \star) \to \star \to \star$ [18, Section 4.6]. However, TApp is naturally kind-polymorphic; just as we do type generalization for term definitions, we now also do kind generalization for type definitions. The kind of TApp is thus inferred as $\forall \mathcal{X}.\ (\mathcal{X} \to \star) \to \mathcal{X} \to \star$. (Like all datatypes, the result of TApp must still be $\star$.)

A less contrived example is the following GADT, used to reify proofs of type equality:

$$\begin{array}{l}\mathbf{data}\ \mathsf{EqRefl}\ a\ b\ \mathbf{where}\\ \quad \mathsf{Refl} :: \mathsf{EqRefl}\ a\ a\end{array}$$

The kind of EqRefl also used to default to $(\star \to \star \to \star)$; that is, EqRefl could only express equality between types of values. To express equality between type constructors, such as Maybe, required tediously declaring a separate EqRefl-like datatype for each kind, with the only difference being kind annotations on EqRefl's type parameters. However, EqRefl is now inferred to have the polymorphic kind $\forall \mathcal{X}.\ \mathcal{X} \to \mathcal{X} \to \star$, so it can be used equally well on any two types of the same kind. We also allow the programmer to write

$$\begin{array}{l}\mathbf{data}\ \mathsf{EqRefl}\ (a :: \mathcal{X})\ (b :: \mathcal{X})\ \mathbf{where}\\ \quad \mathsf{Refl} :: \forall \mathcal{X}.\ \forall (a :: \mathcal{X}).\ \mathsf{EqRefl}\ a\ a\end{array}$$

if they wish to indicate the kind polymorphism explicitly.

A final example of a datatype definition that benefits from kind polymorphism is a higher-kinded fixpoint operator [21]

$$\mathbf{data}\ \mathsf{Mu}\ f\ a = \mathsf{Roll}\ (f\ (\mathsf{Mu}\ f)\ a).$$

Mu can be used to explicitly construct polymorphic recursive types from their underlying functors[2]; for instance:

$$\begin{array}{l}\mathbf{data}\ \mathsf{ListF}\ f\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (f\ a)\\ \mathbf{type}\ \mathsf{List}\ a = \mathsf{Mu}\ \mathsf{ListF}\ a\end{array}$$

Previously, the kind of Mu would have been defaulted to $((\star \to \star) \to (\star \to \star)) \to (\star \to \star)$; with the addition of kind polymorphism, Mu is given the polymorphic kind

$$\mathsf{Mu} :: \forall \mathcal{X}.\ ((\mathcal{X} \to \star) \to (\mathcal{X} \to \star)) \to (\mathcal{X} \to \star)$$

and can now be used to construct recursive types indexed on kinds other than $\star$. For example, here is an explicit construction of the Vec datatype from §2.2:

$$\begin{array}{l}\mathbf{data}\ \mathsf{VecF}\ (a :: \star)\ (f :: \mathsf{Nat} \to \star)\ (n :: \mathsf{Nat})\ \mathbf{where}\\ \quad \mathsf{VFNil}\ \ :: \mathsf{VecF}\ a\ f\ \mathsf{Zero}\\ \quad \mathsf{VFCons} :: a \to f\ n \to \mathsf{VecF}\ a\ f\ (\mathsf{Succ}\ n)\\ \mathbf{type}\ \mathsf{Vec}\ a\ n = \mathsf{Mu}\ (\mathsf{VecF}\ a)\ n\end{array}$$

This time, Mu is instantiated to $((\mathsf{Nat} \to \star) \to (\mathsf{Nat} \to \star)) \to (\mathsf{Nat} \to \star)$.

2.5 Kind polymorphism for classes

Classes can also usefully be kind-polymorphic. This allows us, for example, to clean up Haskell's family of Typeable classes, which currently look like this:

$$\begin{array}{l}\mathbf{class}\ \mathsf{Typeable}\ (a :: \star)\ \mathbf{where}\\ \quad \mathit{typeOf} :: a \to \mathsf{TypeRep}\\ \mathbf{class}\ \mathsf{Typeable1}\ (a :: \star \to \star)\ \mathbf{where}\\ \quad \mathit{typeOf1} :: \forall b.\ a\ b \to \mathsf{TypeRep}\\ \ldots\ \mathit{and\ so\ on}\ \ldots\end{array}$$

The lack of kind polymorphism is particularly unfortunate here; the library has only a fixed, *ad hoc*, collection of Typeable classes. With kind polymorphism we can write

$$\begin{array}{l}\mathbf{class}\ \mathsf{Typeable}\ a\ \mathbf{where}\\ \quad \mathit{typeOf} :: \mathsf{Proxy}\ a \to \mathsf{TypeRep}\end{array}$$

We have generalized in two ways here. First, Typeable gets a polymorphic kind: $\mathsf{Typeable} :: \forall \mathcal{X}.\ \mathcal{X} \to \mathsf{Constraint}$[3], so that it can be used for types of any kind. Second, we need some way to generalize the argument of typeOf, which we have done via a poly-kinded data type Proxy:

$$\mathbf{data}\ \mathsf{Proxy}\ a = \mathsf{Proxy}$$

The idea is that, say $\mathit{typeOf}\ (\mathsf{Proxy} :: \mathsf{Proxy}\ \mathsf{Int})$ will return the type representation for Int, while $\mathit{typeOf}\ (\mathsf{Proxy} :: \mathsf{Proxy}\ \mathsf{Maybe})$ will do the same for Maybe. The proxy argument carries no information—the type has only one, nullary constructor—and is only present so that the programmer can express the type at which to invoke typeOf. Because there are no constraints on the kind of a, it is safe to assign Proxy the polymorphic kind $\forall \mathcal{X}.\ \mathcal{X} \to \star$.

The user is allowed to provide instances of the class Typeable for specific kind and type instantiations, such as:

$$\begin{array}{l}\mathbf{instance}\ \mathsf{Typeable}\ \mathsf{Int}\ \mathbf{where}\\ \quad \mathit{typeOf}\ _ = \ldots \quad \text{-- some representation for Int}\\ \mathbf{instance}\ \mathsf{Typeable}\ \mathsf{Maybe}\ \mathbf{where}\\ \quad \mathit{typeOf}\ _ = \ldots \quad \text{-- some representation for Maybe}\end{array}$$

Even though the class *declaration* is kind polymorphic, the instances need not be.

2.6 Kind polymorphism for terms

So far, we have given examples of *types* with polymorphic kinds. It is also useful to have *terms* whose types involve kind polymorphism. We have already seen several, since data constructors of kind-polymorphic data types have kind-polymorphic types, thus:

$$\begin{array}{l}\mathsf{Proxy} :: \forall \mathcal{X}.\ \forall (a :: \mathcal{X}).\ \mathsf{Proxy}\ a\\ \mathsf{Refl}\ \ :: \forall \mathcal{X}.\ \forall (a :: \mathcal{X}).\ \mathsf{EqRefl}\ a\ a\end{array}$$

Here is a more substantial example, taken from McBride [20]. Consider, first, the type constructor $(:\!\to)$ defined as follows:

[2] The cognoscenti will know that lists can be expressed with a simpler, first-order fixpoint operator. We use a higher-kinded one here because it can also handle an indexed type such as Vec.

[3] The Constraint kind is a new base kind introduced to classify the types of evidence terms, such as type class dictionaries—more about this feature in §3.

$$\textbf{type}\ s :\rightarrow t = \forall i.\ s\ i \rightarrow t\ i$$

If $s, t :: \kappa \rightarrow \star$ are type constructors indexed by types of kind κ, then $s :\rightarrow t$ is the type of *index-preserving functions* from s to t.

For example, consider the function $vdup$ which duplicates each element of a length-indexed vector, guaranteeing to preserve the length of the vector:

$$\begin{array}{lll} vdup :: \mathsf{Vec}\ a\ n & \rightarrow \mathsf{Vec}\ (a, a)\ n \\ vdup\ \ \mathsf{VNil} & = \mathsf{VNil} \\ vdup\ \ (\mathsf{VCons}\ a\ as) & = \mathsf{VCons}\ (a, a)\ (vdup\ as) \end{array}$$

Using the type synonym above, $vdup$'s type can be rewritten as

$$vdup :: \mathsf{Vec}\ a :\rightarrow \mathsf{Vec}\ (a, a).$$

McBride observes that it is possible (and useful) to define functors that lift index-preserving functions (like $vdup$) from one indexed set (such as $\mathsf{Vec}\ a\ n$) to another (such as square matrices $\mathsf{Vec}\ (\mathsf{Vec}\ a\ n)\ n$). He introduces the notion of an *indexed functor*

$$\begin{array}{l} \textbf{class}\ \mathsf{IFunctor}\ f\ \textbf{where} \\ \quad imap :: (s :\rightarrow t) \rightarrow (f\ s :\rightarrow f\ t) \end{array}$$

and goes on to present several interesting instances of this class.

Now, what are the kinds of s, t, and f? A bit of thinking (or simply running GHC's type checker) reveals that there must be kinds $\mathcal{X}_1$ and $\mathcal{X}_2$ such that s, t, and f have the kinds

$$\begin{array}{l} s, t :: \mathcal{X}_1 \rightarrow \star \\ f :: (\mathcal{X}_1 \rightarrow \star) \rightarrow (\mathcal{X}_2 \rightarrow \star) \end{array}$$

All three are mentioned in the type of $imap$, which ought to be polymorphic over $\mathcal{X}_1$ and $\mathcal{X}_2$. The full type of $imap$ must therefore be

$$\begin{array}{r} \forall \mathcal{X}_1\ \mathcal{X}_2.\ \forall f :: (\mathcal{X}_1 \rightarrow \star) \rightarrow (\mathcal{X}_2 \rightarrow \star).\ \forall s, t :: \mathcal{X}_1 \rightarrow \star. \\ \mathsf{IFunctor}\ f \Rightarrow (s :\rightarrow t) \rightarrow (f\ s :\rightarrow f\ t). \end{array}$$

Note that this type involves two different sorts of $\forall$ abstractions: the first abstracts over the kinds $\mathcal{X}_1$ and $\mathcal{X}_2$, whereas the others abstract over the type constructors s, t, and f.

As an aside, this example highlights an interesting difference between $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ and SHE [19], which restricts the kind of f to

$$f :: ('a \rightarrow \star) \rightarrow ('b \rightarrow \star).$$

That is, using SHE, f can only transform types indexed by some *promoted* kinds, whereas in our implementation f can transform types indexed by arbitrary kinds. This is certainly no failing of SHE, which places this restriction on kind polymorphism for the sensible reason that promoted kinds can all be "erased" to $\star$, and as a textual preprocessor SHE cannot be expected to do much else. However, this does show one of the advantages of a natively implemented, strongly typed implementation over a textual preprocessor.

2.7 Kind-indexed type families

In GHC type families are type-indexed. With the addition of kind-polymorphism we may now write kind- and type-indexed families, as the example below demonstrates:

$$\begin{array}{lll} \textbf{type family} & \mathsf{Shape}\ (a :: \mathcal{X}) & :: \star \\ \textbf{type instance} & \mathsf{Shape}\ (a :: \star) & = a \\ \textbf{type instance} & \mathsf{Shape}\ (a :: \star \rightarrow \mathcal{X}) & = \mathsf{Shape}\ (a\ ()) \end{array}$$

The above type family is an example of arity-generic programming, where $\mathsf{Shape}\ t$ denotes the application of the type constructor t to as many copies of the unit type as its kind allows. This form of kind indexing allows for code reuse, since without it the programmer would have to declare separate type families at each kind, in a manner similar to the current treatment of the Typeable class.

2.8 Summary

So far, we have demonstrated the nature of programs that can be written with our two new features:

- Automatic promotion of datatypes to be kinds and data constructors to be types.
- Kind polymorphism, for kinds, types (including kind-indexed type families and type classes), and terms.

The former allows us to give informative kinds to types, thereby excluding bad programs that were previously accepted. The latter accepts a wider class of good programs, and increases re-use, just like type polymorphism. Both features are integrated with type and kind inference.

Our extensions not only enable programmers to write new and interesting programs, but also help clean up existing libraries. For instance, the HList library [13] may be entirely rewritten without code duplication and extra type classes to track well-kindedness. Similarly, the Scrap Your Boilerplate library [14, 15] can benefit from the kind-polymorphic Typeable class described previously.

3. System $\mathrm{F}_{\mathrm{C}}^{\uparrow}$

In this section we present the extensions to GHC's intermediate language, System F_{C} [29], that are necessary to support these new source language features. System F_{C} is an explicitly typed intermediate language which has simple syntax-directed typing rules and is robust to program transformation. Through the mechanism of type (and now kind) inference, source Haskell programs are elaborated to well-typed System F_{C} programs, a process that guarantees the soundness of type inference.

This section presents the technical details of System F_{C} and its extensions, to provide a semantics for the new features. Because F_{C} is a small language, it allows us to make precise exactly what our new design does and does not support. Moreover, System F_{C} has a straightforward metatheory, so we can justify our design decisions by demonstrating that our additions do not complicate reasoning about the properties of the system (§4).

For clarity, we call the extended language of this paper System $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ (pronounced "FC-pro"), reserving the name System F_{C} for the prior version of the language.

3.1 System $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ overview

The expression syntax of System $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ is given in Figure 1, with the differences from System F_{C} highlighted. As the language is explicitly typed, this syntax references kinds (κ) and types (τ), which appear in Figure 3 and Figure 4.

The syntax also mentions *coercions* (γ), which explicitly encode type equalities arising from type family instance declarations and GADT constructors (plus a Haskell-specific form of type generativity, **newtype** declarations). For example, each type family instance declaration gives rise (through elaboration) to an equality axiom that equates the left and the right-hand side of the instance declaration. Such axioms can be composed and transformed to form coercions between more complex types. The role of coercions is not central to this paper, so we refer the reader to related work for more details on motivation and uses of coercions [29, 33], or the foundations of type equalities in type theory [16].

This abstract syntax of $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ makes no mention of the quotemarks of §2.2 because the quotes are required only to resolve ambiguity in the *concrete* syntax of Haskell. As a convention, we use overline notation to stand for a list of syntactic elements, such as the branches of a case expression $\overline{p \rightarrow u}$. Multi-substitution, such as $\tau\ [\overline{\sigma}/\overline{a}]$, is only well-defined when the lists have the same length.

$e,\ u$	$::=$	Expressions
	$\mid x$	Variables
	$\mid \lambda x\colon \tau.\ e \mid e_1\ e_2$	Abstraction/application
	$\mid \Lambda a\colon \kappa.\ e \mid e\ \tau$	Type abstraction/application
	$\mid \lambda c\colon \tau.\ e \mid e\ \gamma$	Coercion abstraction/application
	$\mid \Lambda \mathcal{X}.\ e$	Kind abstraction
	$\mid e\ \kappa$	Kind application
	$\mid K$	Data constructors
	$\mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Case analysis
	$\mid e \rhd \gamma$	Casting
p	$::=$	Patterns
	$\mid K\ \overline{\mathcal{Y}}\ \overline{b\colon \kappa}\ \overline{c\colon \sigma}\ \overline{x\colon \tau}$	Data constructor pattern
q	$::=$	Term-level names
	$\mid x$	Term variables
	$\mid K$	Data constructors
	$\mid c$	Coercion variables and axioms
w	$::=$	Type-level names
	$\mid a$	Type variables
	$\mid H$	Type constants
	$\mid F$	Type functions
bnd	$::= q\colon \tau \mid w\colon \kappa \mid \mathcal{X}\colon \Box$	Bindings
Γ	$::= \varnothing \mid \Gamma,\ bnd$	Contexts

Figure 1. Syntax of expressions and contexts

System F_C has three forms of abstraction, over expressions $\lambda x\colon \tau.\ e$, types $\Lambda a\colon \kappa.\ e$, and coercions $\lambda c\colon \tau.\ e$. Figure 1 shows that System $F_C^{\uparrow}$ adds a fourth abstraction form, $\Lambda \mathcal{X}.\ e$, to abstract over kinds. Correspondingly, the term language of $F_C^{\uparrow}$ includes four forms of application, for expressions, coercions, types, and kinds. In order to make sure that all subtleties of the semantics are clearly exposed, we choose not to merge the four abstraction forms into one (and similarly applications), as is customary in pure type systems [2]; they are, however, combined in our implementation.

$\boxed{\Gamma \vdash_{\mathsf{tm}} e\ :\ \tau}$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e\ :\ \tau_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{tm}} e \rhd \gamma\ :\ \tau_2}\ \text{T_CAST}$$

$$\frac{\Gamma,\ \mathcal{X}\colon \Box \vdash_{\mathsf{tm}} e\ :\ \tau}{\Gamma \vdash_{\mathsf{tm}} \Lambda \mathcal{X}.\ e\ :\ \forall \mathcal{X}.\ \tau}\ \text{T_KABS}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e\ :\ \forall \mathcal{X}.\ \tau \quad \Gamma \vdash_{\mathsf{k}} \kappa\colon \Box}{\Gamma \vdash_{\mathsf{tm}} e\ \kappa\ :\ \tau[\kappa/\mathcal{X}]}\ \text{T_KAPP}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e\ :\ T\ \overline{\kappa}\ \overline{\sigma} \\ \text{for each} \\ K_j\colon \forall \overline{\mathcal{X}}.\ \forall \overline{a\colon \kappa_j}.\ \forall \overline{\mathcal{Y}_j}.\ \forall \overline{b_j\colon \eta_j}.\ \overline{\psi_j} \to \overline{\varphi_j} \to (T\ \overline{\mathcal{X}}\ \overline{a}) \in \Gamma_0 \\ \quad \overline{\eta'_j = \eta_j[\overline{\kappa}/\overline{\mathcal{X}}]} \\ \quad \overline{\psi'_j = \psi_j\ [\overline{\kappa}/\overline{\mathcal{X}}]\ [\overline{\sigma}/\overline{a}]} \\ \quad \overline{\varphi'_j = \varphi_j\ [\overline{\kappa}/\overline{\mathcal{X}}]\ [\overline{\sigma}/\overline{a}]} \\ \quad \Gamma, \overline{\mathcal{X}_j\colon \Box}, \overline{b_j\colon \eta_j}, \overline{c_j\colon \psi'_j}, \overline{x_j\colon \varphi'_j} \vdash_{\mathsf{tm}} u_j\ :\ \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{K_j\ \overline{\mathcal{Y}_j}\ \overline{b_j\colon \eta'_j}\ \overline{c_j\colon \psi'_j}\ \overline{x_j\colon \varphi'_j} \to u_j}\ :\ \tau}\ \text{T_CASE}$$

Figure 2. Typing rules (selected)

Expression typing The typing judgement for terms is syntax-directed and largely conventional; Figure 2 gives the typing rules for some of the novel syntactic forms. Explicit type coercions of the form $e \rhd \gamma$ are used to cast the type of an expression e from τ_1 to τ_2, given γ, a proof that the two types are equal. Note that although these coercions are explicit in the intermediate language, they have no runtime significance and are erased by GHC in a later compilation stage.

Rule T_CASE is used to typecheck pattern matching expressions and will be discussed in more detail in §3.2, where we address datatypes.

ι	$::=$	Base kinds
	$\mid \star$	Star
	$\mid \mathsf{Constraint}$	Constraint kind
$\kappa,\ \eta$	$::=$	Kinds
	$\mid \mathcal{X}$	Kind variables
	$\mid \iota$	Base kinds
	$\mid \kappa_1 \to \kappa_2$	Arrow kinds
	$\mid \forall \mathcal{X}.\ \kappa$	Kind polymorphism
	$\mid T\ \overline{\kappa}$	Promoted type constant

Figure 3. Syntax of kinds

H	$::=$	Type constants
	$\mid T$	Datatypes
	$\mid (\to)$	Arrow
	$\mid (\sim)$	Equality
$\sigma,\ \tau,\ \varphi,\ \psi$	$::=$	Types
	$\mid a$	Variables
	$\mid H$	Constants
	$\mid F$	Type functions
	$\mid K$	Promoted data constructors
	$\mid \forall\ a\colon \kappa.\ \tau$	Polymorphic types
	$\mid \forall \mathcal{X}.\ \tau$	Kind-polymorphic types
	$\mid \tau_1\ \tau_2$	Type application
	$\mid \tau\ \kappa$	Kind application
$\gamma,\ \delta$	$::=$	Coercions
	$\mid c\ \overline{\kappa}\ \overline{\gamma}$	Variables and Axioms
	$\mid \langle \tau \rangle$	Reflexivity
	$\mid \mathbf{sym}\ \gamma$	Symmetry
	$\mid \gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid \forall\ a\colon \kappa.\ \gamma$	Type polytype cong
	$\mid \forall \mathcal{X}.\ \gamma$	Kind polytype cong
	$\mid \gamma_1\ \gamma_2$	Type app cong
	$\mid \gamma\ \kappa$	Kind app cong
	$\mid \gamma[\tau]$	Type instantiation
	$\mid \gamma[\kappa]$	Kind instantiation
	$\mid \mathbf{nth}^i\ \gamma$	Nth argument projection

Figure 4. Syntax of types and coercions

Kinds The syntax of kinds of $F_C^{\uparrow}$ is given in Figure 3. In addition to the familiar $\star$ and $\kappa_1 \to \kappa_2$ kinds, the syntax introduces kind variables $\mathcal{X}$ and polymorphic kinds $\forall \mathcal{X}.\ \kappa$. The kind well-formedness rules are given in Figure 5. Kinds are uni-typed in the sense that there exists only one sort of kinds: $\Box$. Two more notable syntactic forms of kinds are included in $F_C^{\uparrow}$:

- Applications of promoted datatypes (§2), of the form $T\ \overline{\kappa}$. We discuss the details of this mechanism in §3.3.

$\boxed{\Gamma \vdash_{\mathsf{k}} \kappa : \Box}$ Kind validity

$$\frac{\mathcal{X} : \Box \in \Gamma}{\Gamma \vdash_{\mathsf{k}} \mathcal{X} : \Box} \quad \text{KV_VAR}$$

$$\frac{}{\Gamma \vdash_{\mathsf{k}} \iota : \Box} \quad \text{KV_BASE}$$

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa_1 : \Box \quad \Gamma \vdash_{\mathsf{k}} \kappa_2 : \Box}{\Gamma \vdash_{\mathsf{k}} \kappa_1 \to \kappa_2 : \Box} \quad \text{KV_ARR}$$

$$\frac{\Gamma, \mathcal{X} : \Box \vdash_{\mathsf{k}} \kappa : \Box}{\Gamma \vdash_{\mathsf{k}} \forall \mathcal{X}. \kappa : \Box} \quad \text{KV_ABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{k}} \kappa_1 : \Box \quad .. \quad \Gamma \vdash_{\mathsf{k}} \kappa_n : \Box \\ \varnothing \vdash_{\mathsf{ty}} T : \star^n \to \star\end{array}}{\Gamma \vdash_{\mathsf{k}} T\,\overline{\kappa} : \Box} \quad \text{KV_LIFT}$$

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$ Kinding

$$\frac{\vdash \Gamma \quad w : \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} w : \kappa} \quad \text{K_VAR}$$

$$\frac{\vdash \Gamma \quad K : \tau \in \Gamma \quad \varnothing \vdash \tau \rightsquigarrow \kappa}{\Gamma \vdash_{\mathsf{ty}} K : \kappa} \quad \text{K_LIFT}$$

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa : \Box \quad \Gamma, a : \kappa \vdash_{\mathsf{ty}} \tau : \iota}{\Gamma \vdash_{\mathsf{ty}} \forall a : \kappa. \tau : \iota} \quad \text{K_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\,\tau_2 : \kappa_2} \quad \text{K_APP}$$

$$\frac{\Gamma, \mathcal{X} : \Box \vdash_{\mathsf{ty}} \tau : \iota}{\Gamma \vdash_{\mathsf{ty}} \forall \mathcal{X}. \tau : \iota} \quad \text{K_KABS}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \forall \mathcal{X}. \kappa \quad \Gamma \vdash_{\mathsf{k}} \kappa_1 : \Box}{\Gamma \vdash_{\mathsf{ty}} \tau\,\kappa_1 : \kappa[\kappa_1/\mathcal{X}]} \quad \text{K_KAPP}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \iota_1 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \iota_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2 : \iota_2} \quad \text{K_ARRT}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\sim) : \forall \mathcal{X}. \mathcal{X} \to \mathcal{X} \to \mathsf{Constraint}} \quad \text{K_EQ}$$

Figure 5. Formation rules for kinds and types

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \tau}$ Coercion typing

$$\frac{\begin{array}{c}c : \forall \overline{\mathcal{X}}. \forall \overline{a : \eta}. (\tau_1 \sim \tau_2) \in \Gamma \\ \text{for each } \gamma_i \in \overline{\gamma}, \\ \Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \varphi_i \\ \Gamma \vdash \sigma_i, \varphi_i : (\eta_i[\overline{\kappa}/\overline{\mathcal{X}}])\end{array}}{\Gamma \vdash_{\mathsf{co}} c\,\overline{\kappa}\,\overline{\gamma} : (\tau_1\,[\overline{\kappa}/\overline{\mathcal{X}}]\,[\overline{\sigma}/\overline{a}]) \sim (\tau_2\,[\overline{\kappa}/\overline{\mathcal{X}}]\,[\overline{\varphi}/\overline{a}])} \quad \text{C_VARAX}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle : \tau \sim \tau} \quad \text{C_REFL}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,\gamma : \tau_2 \sim \tau_1} \quad \text{C_SYM}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \quad \text{C_TRANS}$$

$$\frac{\begin{array}{c}\Gamma, a : \kappa \vdash \tau_1, \tau_2 : \iota \\ \Gamma, a : \kappa \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall a : \kappa. \gamma : \forall a : \kappa. \tau_1 \sim \forall a : \kappa. \tau_2} \quad \text{C_TABS}$$

$$\frac{\begin{array}{c}\Gamma, \mathcal{X} : \Box \vdash \tau_1, \tau_2 : \iota \\ \Gamma, \mathcal{X} : \Box \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall \mathcal{X}. \gamma : \forall \mathcal{X}. \tau_1 \sim \forall \mathcal{X}. \tau_2} \quad \text{C_KABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_1, \sigma_2 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash \tau_1, \tau_2 : \kappa_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1 \sim \sigma_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\,\gamma_2 : \sigma_1\,\tau_1 \sim \sigma_2\,\tau_2} \quad \text{C_APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1, \tau_2 : \forall \mathcal{X}. \kappa' \\ \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{k}} \kappa : \Box\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma\,\kappa : \tau_1\,\kappa \sim \tau_2\,\kappa} \quad \text{C_KAPP}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \forall a : \kappa. \tau_1 \sim \forall a : \kappa. \tau_2 \quad \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa}{\Gamma \vdash_{\mathsf{co}} \gamma[\sigma] : \tau_1[\sigma/a] \sim \tau_2[\sigma/a]} \quad \text{C_TINST}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \forall \mathcal{X}. \tau_1 \sim \forall \mathcal{X}. \tau_2 \quad \Gamma \vdash_{\mathsf{k}} \kappa : \Box}{\Gamma \vdash_{\mathsf{co}} \gamma[\kappa] : \tau_1[\kappa/\mathcal{X}] \sim \tau_2[\kappa/\mathcal{X}]} \quad \text{C_KINST}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : H\,\overline{\kappa}\,\overline{\tau} \sim H\,\overline{\kappa}\,\overline{\tau'}}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^j\,\gamma : \tau_j \sim \tau'_j} \quad \text{C_NTH}$$

Figure 6. Formation rules for coercions

- Another small extension of System $F_C^{\uparrow}$ is a special base kind Constraint [4], which classifies types representing evidence, such as type class dictionaries and type equalities. While in source Haskell such evidence is *implicit*, the elaboration of a source program into $F_C^{\uparrow}$ constructs *explicit* evidence terms whose types have kind Constraint. We use the metavariable ι to refer to an arbitrary base kind ($\star$ or Constraint).

The kind well-formedness rules in Figure 5 ensure that kinds are *first-order*, in the sense that we do not include any kind operators. In other words, Maybe by itself is not a kind, although Maybe $\star$ is, and there are no classifiers for kind variables other than $\Box$.

Types The types of $F_C^{\uparrow}$ are given in Figure 4, and their kinding rules appear in Figure 5. The new constructs include (i) kind-polymorphic types $\forall \overline{\mathcal{X}}. \tau$, (ii) kind application $\tau\,\kappa$, and (iii) promoted data constructors. Kind-polymorphic types classify kind-abstractions in the expression language. The syntax $\tau\,\kappa$ applies a type constructor with a polymorphic kind to a kind argument. There is no explicit kind abstraction form in the type language for the same reason that there is no explicit type abstraction form—type inference in the presence of anonymous abstractions would require higher-order unification.

The syntax of constants includes the equality constructor $(\sim)$, which has the polymorphic kind $\forall \mathcal{X}. \mathcal{X} \to \mathcal{X} \to \mathsf{Constraint}$ by rule K_EQ. This means that equality constraints written $\tau_1 \sim \tau_2$ are actually formed from the application of the constant $(\sim)$ to a kind and two type arguments of that kind. In F_C, this equality constraint was a special form, but the addition of polymorphic kinds means that it can be internalized and treated just like any other type constructor. To keep the syntactic overhead low, we continue to use the notation $\tau_1 \sim \tau_2$ to stand for the application $(\sim)\,\kappa\,\tau_1\,\tau_2$ when the kind is unimportant or clear from the context.

Another minor point is the use of ι to classify the base kinds in rules K_ABS and K_KABS (Figure 5). A conventional system would require that a type $\forall a.\tau$ has kind $\star$, but in our system it can have either kind $\star$ or Constraint, because both classify types inhabited by values.

Coercions Coercions, also shown in Figure 4, are "proofs" that provide explicit evidence of the equality between types. The judgement $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ expresses the fact that the coercion γ is a proof of equality between the types τ_1 and τ_2. If such a derivation is possible, it is an invariant of the relation that τ_1 and τ_2 have the same kind, and that kind instantiates the $(\sim)$ constructor in the conclusion of the relation.

The coercion forms are best understood by looking at their formation rules, shown in Figure 6. Coercion variables and uses of primitive axioms are typed by rule C_VARAX. Recall that primitive coercion axioms may be kind-polymorphic since type family instance declarations in source Haskell may be kind-polymorphic, such as the last Shape instance from §2.7. When such axioms are

$\boxed{\vdash \Gamma}$ Context well-formedness

$$\frac{}{\vdash \varnothing}\ \text{GWF_EMPTY}$$

$$\frac{\vdash \Gamma \quad \mathcal{X} \# \Gamma}{\vdash \Gamma,\ \mathcal{X}\colon \square}\ \text{GWF_SORT}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash_{\mathsf{k}} \kappa\colon \square \quad a \# \Gamma}{\vdash \Gamma,\ a\colon \kappa}\ \text{GWF_TYVAR}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash_{\mathsf{k}} \kappa\colon \square \quad F \# \Gamma}{\vdash \Gamma,\ F\colon \kappa}\ \text{GWF_TYFUN}$$

$$\frac{\begin{array}{l}\kappa = \forall \overline{\mathcal{X}}.\overline{\kappa} \to \star \\ \vdash \Gamma \quad \Gamma \vdash_{\mathsf{k}} \kappa\colon \square \quad T \# \Gamma\end{array}}{\vdash \Gamma,\ T\colon \kappa}\ \text{GWF_TYDATA}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash_{\mathsf{ty}} \tau : \kappa \quad x \# \Gamma}{\vdash \Gamma,\ x\colon \tau}\ \text{GWF_VAR}$$

$$\frac{\begin{array}{l}\tau = \forall \overline{\mathcal{X}}.\, \forall \overline{a{:}\kappa}.\, \forall \overline{\mathcal{Y}}.\, \forall \overline{b{:}\eta}.\, (\overline{\sigma} \to (T\ \overline{\mathcal{X}}\ \overline{a})) \\ \vdash \Gamma \quad \Gamma \vdash_{\mathsf{ty}} \tau : \star \quad K \# \Gamma\end{array}}{\vdash \Gamma,\ K\colon \tau}\ \text{GWF_CON}$$

$$\frac{\begin{array}{l}\tau = \forall \overline{\mathcal{X}}.\, \forall \overline{a{:}\kappa}.\, (\tau_1 \sim \tau_2) \\ \vdash \Gamma \quad \Gamma \vdash_{\mathsf{ty}} \tau : \mathsf{Constraint} \quad c \# \Gamma\end{array}}{\vdash \Gamma,\ c\colon \tau}\ \text{GWF_AX}$$

Figure 7. Context formation rules

used, they must be applied to kind and coercion arguments. In this rule and elsewhere, the notation $\Gamma \vdash \sigma, \varphi : \kappa$ ensures that both types σ and φ have the same kind in the same context.

Coercions include rules for reflexivity, symmetry, transitivity and congruence (rules C_TABS through C_KAPP). Coercions can be destructed, through instantiation (C_TINST and C_KINST) as well as by appealing to the injectivity of type constructors (C_NTH). Notice that in rule C_NTH the kinds of the two applications are required to be the same—this syntactically ensures that coercions are always between types of exactly the same kind.

Contexts System $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ allows top-level definitions for datatypes T, type functions F, and equality axioms c. Rather than give concrete syntax for declarations of these three constants, we instead give formation rules for the initial context in which terms are type-checked, shown in Figure 7. The notation $x \# \Gamma$ indicates that x is fresh for Γ.

Operational semantics Selected rules of the operational semantics of $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ appear in Figure 8, including the β-reduction rules for abstraction forms and for case expressions. The operational semantics includes crucial "push" rules, inherited from $\mathrm{F_C}$, which make sure that coercions do not interfere with evaluation. For example, rule S_PUSH illustrates a situation where a coercion may interfere with a β-reduction. In that case γ must be a coercion between two function types, $(\tau_1 \to \tau_2) \sim (\sigma_1 \to \sigma_2)$. The rule decomposes γ into two simpler coercions and rewrites the term to expose opportunities for reduction.

3.2 Kind polymorphism and datatypes

We allow datatype definitions to have polymorphic kinds. However, rule GWF_TYDATA requires all type constants to have prenex kind quantification, [4]

$$T\colon \forall \overline{\mathcal{X}}.\overline{\kappa} \to \star.$$

[4] We use the notation $\forall \overline{\mathcal{X}}.\, \tau$ as an abbreviation for $\forall \mathcal{X}_1.\forall \mathcal{X}_2. \ldots \tau$. Likewise, $\overline{\kappa} \to \star$ abbreviates $\kappa_1 \to \kappa_2 \to \ldots \to \star$.

$\boxed{e \longrightarrow e'}$ One-step reduction

$$\frac{}{(\lambda x\colon \tau.\, e)\ e' \longrightarrow e[e'/x]}\ \text{S_BETA}$$

$$\frac{}{(\lambda c\colon \tau.\, e)\ \gamma \longrightarrow e[\gamma/c]}\ \text{S_CBETA}$$

$$\frac{}{(\Lambda a\colon \kappa.\, e)\ \tau \longrightarrow e[\tau/a]}\ \text{S_TBETA}$$

$$\frac{}{(\Lambda \mathcal{X}.\, e)\ \kappa \longrightarrow e[\kappa/\mathcal{X}]}\ \text{S_KBETA}$$

$$\frac{K_i\ \overline{\mathcal{Y}}\ \overline{b{:}\eta}\ \overline{c{:}\psi}\ \overline{x{:}\varphi} \to u_i \in \overline{p \to u}}{\begin{array}{l}\mathbf{case}\ K_i\ \overline{\kappa}\ \overline{\sigma}\ \overline{\kappa'}\ \overline{\tau}\ \overline{\gamma}\ \overline{e}\ \mathbf{of}\ \overline{p \to u} \longrightarrow \\ \quad u_i\ [\overline{e}/\overline{x}]\ [\overline{\gamma}/\overline{c}]\ \left[\overline{\tau}/\overline{b}\right]\ \left[\overline{\kappa'}/\overline{\mathcal{Y}}\right]\end{array}}\ \text{S_CASE}$$

$$\frac{}{(v \triangleright \gamma)\ e \longrightarrow (v\ (e \triangleright \mathbf{sym}\ (\mathbf{nth}^1\ \gamma))) \triangleright \mathbf{nth}^2\ \gamma}\ \text{S_PUSH}$$

$$\frac{}{(v \triangleright \gamma)\ \tau \longrightarrow v\ \tau \triangleright \gamma[\tau]}\ \text{S_TPUSH}$$

$$\frac{}{(v \triangleright \gamma)\ \kappa \longrightarrow v\ \kappa \triangleright \gamma[\kappa]}\ \text{S_KPUSH}$$

$$\frac{\begin{array}{l}\varnothing \vdash_{\mathsf{co}} \gamma' : T\ \overline{\kappa}\ \overline{\sigma_1} \sim T\ \overline{\kappa}\ \overline{\sigma_2} \\ K\colon \forall \overline{\mathcal{X}}.\, \forall \overline{a{:}\kappa}.\, \forall \overline{\mathcal{Y}}.\, \forall \overline{b{:}\eta}.\, (\overline{\psi_1 \sim \psi_2} \to \overline{\varphi} \to (T\ \overline{\mathcal{X}}\ \overline{a})) \in \Gamma \\ \psi_1' = \psi_1 \left[\overline{\tau}/\overline{b}\right] \left[\overline{\kappa'}/\overline{\mathcal{Y}}\right] [\overline{\kappa}/\overline{\mathcal{X}}] \\ \psi_2' = \psi_2 \left[\overline{\tau}/\overline{b}\right] \left[\overline{\kappa'}/\overline{\mathcal{Y}}\right] [\overline{\kappa}/\overline{\mathcal{X}}] \\ \varphi' = \varphi \left[\overline{\tau}/\overline{b}\right] \left[\overline{\kappa'}/\overline{\mathcal{Y}}\right] [\overline{\kappa}/\overline{\mathcal{X}}] \\ \text{for each } \gamma_j \in \overline{\gamma}, \\ \quad \gamma_j' = \mathbf{sym}\ ([\overline{a} \mapsto \overline{\mathbf{nth}\ \gamma'}]\psi_{1\,j}) \fatsemi \gamma_j \fatsemi [\overline{a} \mapsto \overline{\mathbf{nth}\ \gamma'}]\psi_{2\,j} \\ \text{for each } e_j \in \overline{e}, \\ \quad e_j' = e_j \triangleright [\overline{a} \mapsto \overline{\mathbf{nth}\ \gamma'}]\varphi_j'\end{array}}{\begin{array}{l}\mathbf{case}\ (K\ \overline{\kappa}\ \overline{\sigma_1}\ \overline{\kappa'}\ \overline{\tau}\ \overline{\gamma}\ \overline{e}) \triangleright \gamma'\ \mathbf{of}\ \overline{p \to u} \longrightarrow \\ \qquad \mathbf{case}\ K\ \overline{\kappa}\ \overline{\sigma_2}\ \overline{\kappa'}\ \overline{\tau}\ \overline{\gamma'}\ \overline{e'}\ \mathbf{of}\ \overline{p \to u}\end{array}}\ \text{S_CPUSH}$$

$$\frac{}{(v \triangleright \gamma_1) \triangleright \gamma_2 \longrightarrow v \triangleright (\gamma_1 \fatsemi \gamma_2)}\ \text{S_COMB}$$

Figure 8. Operational semantics of $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ (selected rules)

In other words, type constants must take all of their kind arguments before any of their type arguments. This restriction simplifies their semantics. For example, because type constants are injective, equations between them may be decomposed into equations between their type arguments, as in the rule C_NTH. By not allowing kind and type arguments to mix, we can write this rule succinctly.

Data constructors for kind-polymorphic datatypes must themselves be kind polymorphic (the alternative would be to allow *kind equalities*, just as type equalities are currently allowed for GADTs, but we do not: see §3.4). Rule GWF_CON requires data constructors to have types of the following form:

$$K\colon \forall \overline{\mathcal{X}}.\, \forall \overline{a{:}\kappa}.\, \forall \overline{\mathcal{Y}}.\, \forall \overline{b{:}\eta}.\, \overline{\varphi} \to (T\ \overline{\mathcal{X}}\ \overline{a})$$

Data constructors can be polymorphic over kinds and types, all of which must show up as parameters to the datatypes that they construct. Furthermore, data constructors can take additional kind and type arguments that do not appear in the result type, as well as term arguments whose types may include constraints on any of the quantified type variables. As a result, data constructor values carry six lists of arguments. (Above, $\overline{\varphi}$ includes both the types of the coercion and expression arguments.)

The treatment of these six arguments shows up in the formation rule for case expressions (T_CASE, from Figure 2), and the two reduction rules for case expressions (S_CASE and S_CPUSH, from

Figure 8). These rules were already quite involved in System F_C—adding kind polymorphism is a straightforward modification.

The rule T_CASE typechecks a case expression. In this rule, the lists $\overline{\kappa}$ and $\overline{\sigma}$ are the kind and type parameters to some datatype T. These arguments replace the kind and type variables $\overline{\mathcal{X}}$ and $\overline{a}$, wherever they appear in the case expression. The rule then typechecks each branch of the case expression in a context extended with the "existential" kind and type variables, as well as the coercion assumptions and constructor arguments of that branch.

The rule S_CASE describes the normal reduction of a case expression. If the scrutinee is a data constructor value, the appropriate branch is selected, after substitution of the last four arguments carried by the data constructor value—the "existential" kinds and types, coercions, and expression arguments.

If there is a coercion around the data constructor value, rule S_CPUSH pushes the coercion into the arguments of the data constructor. Again, the complexity of the semantics of this rule is inherited from System F_C. The coercion $\overline{\gamma'}$ and expression arguments $\overline{e}$ are each transformed by types "lifted" to coercions, using the operation $[\overline{a}{\mapsto}\overline{\mathbf{nth}\ \gamma}]\tau$. This operation replaces each type variable a_i appearing in the type τ with a coercion $\mathbf{nth}^i\ \gamma$. We discuss this operation in more detail in §4. For more on the operation of this rule, we refer readers to previous work [29, 33].

3.3 Promotion

We allow the "promotion" of certain type constructors into kinds, and the promotion of certain data constructors to become types. But exactly *which* type and data constructors are promoted, and what kinds do the promoted data constructors have? The answers may be found in Figure 5:

- *Type constructors*. Rule KV_LIFT states that $T\,\overline{\kappa}$ is a valid kind only if T is a fully applied type constructor of kind $\star^n \to \star$.
- *Data constructors*. Rule K_LIFT states that a data constructor K may be treated as a type if K's type τ can be promoted to a kind κ, via the judgement $\varnothing \vdash \tau \rightsquigarrow \kappa$, whose rules are given in Figure 9. The latter judgement merely replaces type polymorphism in τ with kind polymorphism in κ, and checks that type constructors mentioned in τ can themselves be promoted.

The rule for promoting type constructors is deliberately restrictive. There are many Haskell type constructors that do not have kinds of the form $\star^n \to \star$.

- We do not promote higher-order types of kind, say, $(\star \to \star) \to \star$. If we did so, we would need a richer classification of kinds to ensure that such promoted higher-order types were applied to appropriate arguments.
- We do not promote a type whose kind itself involves promoted types, such as $\mathsf{Vec} : \star \to \mathsf{Nat} \to \star$. If we did so, the second argument to Vec would have to be a kind classified by Nat. In order to make this possible we would either have to allow "double promotion" (such as promoting a natural number value all the way to the kind level), or make the type and kind levels fully dependent. Either approach would complicate matters, so we rule them out for now.
- We do not promote type constructors with polymorphic kinds. If we did, we would need a system of polymorphic sorts to accommodate the promoted kinds. At present, this seems like extra complication for little benefit.

The guiding principle here is that *kinds are not classified*, or, to put it another way, there is only one sort $\Box$ in the kind validity judgement $\Gamma \vdash_{\mathsf{k}} \kappa : \Box$.

$\boxed{\Theta \vdash \tau \rightsquigarrow \kappa}$ Type lifting

$$\frac{a \mapsto \mathcal{X} \in \Theta}{\Theta \vdash a \rightsquigarrow \mathcal{X}}\ \text{L_VAR}$$

$$\frac{\Theta \vdash \tau_1 \rightsquigarrow \kappa_1 \quad \Theta \vdash \tau_2 \rightsquigarrow \kappa_2}{\Theta \vdash \tau_1 \to \tau_2 \rightsquigarrow \kappa_1 \to \kappa_2}\ \text{L_ARR}$$

$$\frac{\Theta, a \mapsto \mathcal{X} \vdash \tau \rightsquigarrow \kappa}{\Theta \vdash \forall\, a\colon \star\,.\,\tau \rightsquigarrow \forall\, \mathcal{X}.\,\kappa}\ \text{L_ABS}$$

$$\frac{\varnothing \vdash_{\mathsf{ty}} T : \star^n \to \star \quad \overline{\Theta \vdash \tau_i \rightsquigarrow \kappa_i}^{\,n}}{\Theta \vdash T\,\overline{\tau} \rightsquigarrow T\,\overline{\kappa}}\ \text{L_APP}$$

Figure 9. Type to kind translation

3.4 Design principle: no kind equalities!

There is one other major restriction on promoted types: we do not promote GADTs. This limitation derives from an important design principle of $F_C^{\uparrow}$: *we do not allow equality constraints between kinds, nor kind coercions*. Since GADT data constructors take coercions between types as arguments, their promotions would necessarily require coercions between kinds. Disallowing kind equality constraints and coercions means that α-equivalence is the only meaningful equivalence for kinds, which dramatically simplifies the system.

The difficulty with kind constraints lies with type equivalence. In $F_C^{\uparrow}$, all nontrivial type conversions must be made explicit in the code, through the use of coercions. Coercions simplify the metatheory of the F_C language, by rendering type checking trivially decidable and separating the type soundness proof from the consistency of the type equational theory. In the jargon of dependent type theory, F_C has a trivial *definitional equality* (types are equal only when α-equivalent) and an expressive *propositional* equality (coercions are proofs of a much coarser equality between types.)

The presence of nontrivial kind equivalences muddies the definition of propositional equality between types. We would need to generalize the coercion judgement to mention two types as well as their kinds. But what should the interpretation of this judgement be? That the types and their kinds are equal? Should we be able to extract a proof of kind equality from a proof of type equality? Such a system would lead to bloated proofs and many additional rules.

We plan to address these issues in future work, using ideas from Observational Type Theory [1]. But even if we are able to design a sensible core language, there is still the issue of the complexity of the *source* language. Heterogeneous equality in Coq and Agda is notoriously difficult for users to understand and use. By only allowing one-level indexing, we have defined a simple, well-behaved language for users to get started with dependently-typed programming.

4. Metatheory

We now turn to the formal properties of $F_C^{\uparrow}$, and ultimately show that the system enjoys subject reduction and progress *under some consistency requirements* about the initial environment. We begin with some scaffolding lemmas.

Lemma 4.1 (Kind substitution)**.**

1. *If* $\vdash \Gamma_1, (\mathcal{X}\colon\Box), \Gamma_2$ *and* $\Gamma_1 \vdash_{\mathsf{k}} \kappa\colon \Box$ *then* $\vdash \Gamma_1, \Gamma_2[\kappa/\mathcal{X}]$.
2. *If* $\Gamma_1, (\mathcal{X}\colon\Box), \Gamma_2 \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Gamma_1 \vdash_{\mathsf{k}} \kappa' : \Box$ *then* $\Gamma_1, \Gamma_2[\kappa'/\mathcal{X}] \vdash_{\mathsf{ty}} \tau[\kappa'/\mathcal{X}] : \kappa$.
3. *If* $\Gamma_1, (\mathcal{X}\colon\Box), \Gamma_2 \vdash_{\mathsf{k}} \eta\colon \Box$ *and* $\Gamma_1 \vdash_{\mathsf{k}} \kappa\colon \Box$ *then* $\Gamma_1, \Gamma_2[\kappa/\mathcal{X}] \vdash_{\mathsf{k}} \eta[\kappa/\mathcal{X}]\colon \Box$.

The lemmas above are proved simultaneously by induction on the height of the derivation. In each case, the induction hypothesis asserts that all lemmas hold for derivations of smaller height. They have to be proved simultaneously, as each of the three judgements appeals to the other two.

Lemma 4.2 (Type substitution)**.**

1. *If* $\vdash \Gamma_1, (a\colon \eta), \Gamma_2$ *and* $\Gamma_1 \vdash_{\mathsf{ty}} \tau : \eta$ *then* $\vdash \Gamma_1, \Gamma_2[\tau/a]$.
2. *If* $\Gamma_1, (a\colon \eta), \Gamma_2 \vdash_{\mathsf{ty}} \sigma : \kappa$ *and* $\Gamma_1 \vdash_{\mathsf{ty}} \tau : \eta$ *then* $\Gamma_1, \Gamma_2[\tau/a] \vdash_{\mathsf{ty}} \sigma[\tau/a] : \kappa$.
3. *If* $\Gamma_1, (a\colon \eta), \Gamma_2 \vdash_{\mathsf{k}} \kappa\colon \Box$ *and* $\Gamma_1 \vdash_{\mathsf{ty}} \tau : \eta$ *then* $\Gamma_1, \Gamma_2[\tau/a] \vdash_{\mathsf{k}} \kappa\colon \Box$.

With these two lemmas, we can prove that the derived coercions of our system are homogeneous.

Lemma 4.3 (Coercion homogeneity)**.** *If* $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *then* $\Gamma \vdash \tau_1, \tau_2 : \kappa$ *for some kind* κ.

As a corollary, if $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ then $\Gamma \vdash_{\mathsf{ty}} \tau_1 \sim \tau_2 : \mathsf{Constraint}$, since the two types have the same kind and the application of the $(\sim)$ constructor is possible by rule K_EQ.

Moreover, coercion derivations (like type and kind derivations) are preserved by type and kind substitution.

Lemma 4.4 (Kind substitution in coercions)**.** *If* $\Gamma_1, (\mathcal{X}\colon \Box), \Gamma_2 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *and* $\Gamma_1 \vdash_{\mathsf{k}} \kappa\colon \Box$ *then* $\Gamma_1, \Gamma_2[\kappa/\mathcal{X}] \vdash_{\mathsf{co}} \gamma[\kappa/\mathcal{X}] : \tau_1[\kappa/\mathcal{X}] \sim \tau_2[\kappa/\mathcal{X}]$.

Lemma 4.5 (Type substitution in coercions)**.** *If* $\Gamma_1, (a\colon \eta), \Gamma_2 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *and* $\Gamma_1 \vdash_{\mathsf{ty}} \tau : \eta$ *then* $\Gamma_1, \Gamma_2[\tau/a] \vdash_{\mathsf{co}} \gamma[\tau/a] : \tau_1[\tau/a] \sim \tau_2[\tau/a]$.

We may also substitute coercions inside other coercions:

Lemma 4.6 (Coercion substitution in coercion)**.**

1. *If* $\Gamma_1, (c\colon \tau_1 \sim \tau_2), \Gamma_2 \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$ *and* $\Gamma_1 \vdash_{\mathsf{co}} \gamma' : \tau_1 \sim \tau_2$ *then* $\Gamma_1, \Gamma_2[\gamma'/c] \vdash_{\mathsf{co}} \gamma[\gamma'/c] : \sigma_1 \sim \sigma_2$.
2. *If* $\vdash \Gamma_1, (c\colon \tau_1 \sim \tau_2), \Gamma_2$ *and* $\Gamma_1 \vdash_{\mathsf{co}} \gamma' : \tau_1 \sim \tau_2$ *then* $\vdash \Gamma_1, \Gamma_2[\gamma'/c]$.
3. *If* $\Gamma_1, (c\colon \tau_1 \sim \tau_2), \Gamma_2 \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Gamma_1 \vdash_{\mathsf{co}} \gamma' : \tau_1 \sim \tau_2$ *then* $\Gamma_1, \Gamma_2[\gamma'/c] \vdash_{\mathsf{ty}} \tau : \kappa$.
4. *If* $\Gamma_1, (c\colon \tau_1 \sim \tau_2), \Gamma_2 \vdash_{\mathsf{k}} \kappa\colon \Box$ *and* $\Gamma_1, \Gamma_2[\gamma'/c] \vdash_{\mathsf{k}} \kappa\colon \Box$.

4.1 Subject reduction

To show subject reduction, we first prove a substitution theorem for terms.

Theorem 4.7 (Substitution)**.**

1. *If* $\Gamma_1, (\mathcal{X}\colon \Box), \Gamma_2 \vdash_{\mathsf{tm}} e : \tau$ *and* $\Gamma_1 \vdash_{\mathsf{k}} \kappa\colon \Box$ *then* $\Gamma_1, \Gamma_2[\kappa/\mathcal{X}] \vdash_{\mathsf{tm}} e[\kappa/\mathcal{X}] : \tau[\kappa/\mathcal{X}]$.
2. *If* $\Gamma_1, (a\colon \eta), \Gamma_2 \vdash_{\mathsf{tm}} e : \tau$ *and* $\Gamma_1 \vdash_{\mathsf{ty}} \sigma : \eta$ *then* $\Gamma_1, \Gamma_2[\sigma/a] \vdash_{\mathsf{tm}} e[\sigma/a] : \tau[\sigma/a]$.
3. *If* $\Gamma_1, (c\colon \sigma_1 \sim \sigma_2), \Gamma_2 \vdash_{\mathsf{tm}} e : \tau$ *and* $\Gamma_1 \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$ *then* $\Gamma_1, \Gamma_2[\gamma/c] \vdash_{\mathsf{tm}} e[\gamma/c] : \tau$.

We also need a substitution lemma for terms:

Lemma 4.8 (Expression substitution)**.** *If* $\Gamma_1, (x\colon \sigma), \Gamma_2 \vdash_{\mathsf{tm}} e : \tau$ *and* $\Gamma_1 \vdash_{\mathsf{tm}} u : \sigma$ *then* $\Gamma_1, \Gamma_2 \vdash_{\mathsf{tm}} e[u/x] : \tau$.

Finally, the crux of subject reduction is the so-called *lifting lemma*, which also has the same key role for proving subject reduction in previous work on $\mathrm{F_C}$. Recall the *lifting* operation from §3.2, denoted $[\overline{a} \mapsto \overline{\gamma}]\tau$. Lifting provides a way to convert a type to a coercion by substituting its free type variables with coercions between types of the appropriate kind. Its formal definition is straightforward.

Lemma 4.9 (Lifting)**.** *If* $\Gamma, (a\colon \kappa) \vdash_{\mathsf{ty}} \tau : \eta$ *and* $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *with* $\Gamma \vdash \tau_1, \tau_2 : \kappa$ *then* $\Gamma \vdash_{\mathsf{co}} [\overline{a} \mapsto \overline{\gamma}]\tau : \tau[\tau_1/a] \sim \tau[\tau_2/a]$.

With all the scaffolding in place, we can show that evaluation preserves types.

Theorem 4.10 (Subject reduction)**.** *Let* Γ_0 *be the initial environment that does not contain any term, type, or kind variable bindings (we will refer to this as the* top-level *environment in what follows). The following is true: if* $\Gamma_0 \vdash_{\mathsf{tm}} e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $\Gamma_0 \vdash_{\mathsf{tm}} e_2 : \tau$.

4.2 Progress

Despite the tedious scaffolding lemmas, subject reduction is not hard conceptually, since each evaluation rule constructs proof terms which justify the typeability of the reduct.

Progress, on the other hand, is more interesting: the formal operational semantics of $\mathrm{F_C^{\uparrow}}$ involves coercions and pushing coercions in terms, and we must make sure that these coercions do not stand in the way of ordinary β-reductions. If coercions could prevent some reductions, that would contradict our claim that coercions can be safely and entirely erased at runtime. The coercion erasure of a "safe" stuck $\mathrm{F_C^{\uparrow}}$ term could lead to a crashing program.

Intuitively, for progress to hold we must impose the restriction that if a coercion coerces the type of a *value* to another value type then the two value types have the same runtime representation. We formalize this condition below.

Definition 4.11 (Value type)**.** *A type* τ *is a* value type *in an environment* Γ *if* $\Gamma \vdash_{\mathsf{ty}} \tau : \iota$ *and moreover it is of the form* $H\ \overline{\kappa}\ \overline{\sigma}$*, or* $\forall\, a\colon \kappa.\, \sigma$*, or* $\forall\, \mathcal{X}.\, \sigma$.

To prevent unsound type equalities, the initial environment containing axioms and datatypes (but no term, type, and coercion variables) should be *consistent*, a notion that we directly adapt from previous work on $\mathrm{F_C}$ [29, 33].

Definition 4.12 (Consistency)**.** *A context* Γ *is* consistent *iff*

- *If* $\Gamma \vdash_{\mathsf{co}} \gamma : H\ \overline{\kappa}\ \overline{\sigma} \sim \tau$*, and* τ *is a value type, then* $\tau = H\ \overline{\kappa}\ \overline{\varphi}$.
- *If* $\Gamma \vdash_{\mathsf{co}} \gamma : (\forall\, a\colon \kappa.\, \sigma) \sim \tau$*, and* τ *is a value type, then* $\tau = \forall\, a\colon \kappa.\, \varphi$.
- *If* $\Gamma \vdash_{\mathsf{co}} \gamma : (\forall\, \mathcal{X}.\, \sigma) \sim \tau$*, and* τ *is a value type, then* $\tau = \forall\, \mathcal{X}.\, \varphi$.

Under the assumption of consistency for the top-level definitions we can state and prove progress.

Theorem 4.13 (Progress)**.** *If* $\Gamma_0 \vdash_{\mathsf{tm}} e : \tau$ *and* Γ_0 *is a top-level consistent environment, then either* e *is a value possibly wrapped in casts, or* $e \longrightarrow e'$ *for some* e'.

As defined above, consistency is a property not only of the initial environment but also of the coercion formation judgement. It is therefore not easy to check. For this reason, previous work [33] gives sufficient conditions exclusively on the top-level environment which guarantee consistency (for a similar but simpler coercion language). These conditions can be adapted in a straightforward way here—the only interesting bit is that type family axioms now involve type and kind arguments and both forms of arguments have to be taken into account when checking the overlap of these axioms.

We finally remark that unsound equalities such as $\mathsf{Int}\sim\mathsf{Bool}$ can be derived *by terms*, since $\mathsf{Int}\sim\mathsf{Bool}$ is just a type, so one may wonder how type safety can possibly hold. For instance, the following is valid $\mathrm{F_C^{\uparrow}}$ code:

$$f :: (\mathsf{Int}\sim\mathsf{Bool})$$
$$f = \bot$$

Fortunately, such terms do not pose any danger to type safety, because the syntax prevents us from injecting ordinary terms into

the universe of coercions and using them to erroneously cast the types of other expressions.

5. Surface language and elaboration

When extending Haskell's surface syntax to expose the new features to users, we found two major issues that needed to be addressed. First, a new mechanism for namespace resolution was needed; second, we had to decide when and to what degree kind generalization should be performed.

5.1 Namespace resolution

As we saw in §2.2, type and data constructors have separate namespaces, so that type constructors and data constructors can be given the same name. This practice is common in Haskell programs, starting with Prelude declarations such as $(,)$ and $[]$. However, when data constructors can appear in types, it is no longer clear what namespace should be used to resolve type constants. SHE explored one solution to this problem: it requires all promoted data constructors and kinds to be surrounded by curly braces.

$$\begin{array}{l}\mathbf{data}\ \mathsf{Vec} :: \star \to \{\mathsf{Nat}\} \to \star\ \mathbf{where}\\ \quad \mathsf{VNil}\ \ :: \mathsf{Vec}\ a\{\mathsf{Z}\}\\ \quad \mathsf{VCons} :: a \to \mathsf{Vec}\ a\{n\} \to \mathsf{Vec}\ a\{\mathsf{S}\ n\}\end{array}$$

Note that curly braces can surround type expressions, not just data constructors, as in $\{\mathsf{S}\ n\}$ above. Such notation is useful for expressions with multiply lifted components. However, in SHE, type variables with promoted kinds must also be lifted, meaning that SHE is based on lifting *values* rather than just constructors.

In contrast, we separate the issue of namespace resolution from that of semantics by adopting the single quote mechanism. Each (ambiguous) data constructor must be marked. We find that this notation is easier to specify, as the treatment of type variables is completely standard. Furthermore, because quotes can often be omitted, it is also visually lighter.

5.2 Kind generalization

The second issue that we needed to address in the source language extension was the specification of where kind generalization should occur, and how it should interact with type and kind annotations and lexically scoped type (and now kind) variables.

In general, our extension tries to be consistent with the existing treatment of type polymorphism. Kind variables are generalized at the same places that type variables are, and kind variables can be brought into scope using annotations just like type variables. Furthermore, kind variables that appear in type argument signatures are quantified. For example, the declaration of T below is valid, and the kind of T is inferred to be $\forall\mathcal{X}.\ (\mathcal{X} \to \star) \to \mathcal{X} \to \star$.

$$\mathbf{data}\ \mathsf{T}\ (a :: \mathcal{X} \to \star)\ b = \mathsf{K}\ (a\ b)$$

In the same way, kind polymorphism can be explicitly specified in class declarations. For example, the definition of $\mathsf{IFunctor}$ in §2.6 is accepted just as it appears there, but the programmer may also write this more explicit definition:

$$\begin{array}{l}\mathbf{class}\ \mathsf{IFunctor}\ (f :: (\mathcal{X}_1 \to \star) \to \mathcal{X}_2 \to \star)\ \mathbf{where}\\ \quad imap :: \forall(s, t :: \mathcal{X}_1 \to \star).\ (s :\to t) \to (f\ s :\to f\ t)\end{array}$$

The first line brings the kind variables $\mathcal{X}_1$ and $\mathcal{X}_2$ into scope, along with the type variable f; all are then mentioned in the type of $imap$.

6. Implementation

We have a preliminary implementation of our system ready for release with GHC version 7.4. This initial release includes a solid implementation of $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ in the Core language; the source-language features, and the kind inference that supports them, mostly work but are less fully tested. The changes required to the compiler, although widespread, were surprisingly modest. We briefly summarize them here. To understand this explanation it may help to recall that GHC's type inference engine *elaborates* input terms by filling in implicit type and kind abstractions and applications, so that the terms can subsequently be desugared into $\mathrm{F}_{\mathrm{C}}^{\uparrow}$.

6.1 The Core language

We extended GHC's Core language to support kind polymorphism, which was mostly straightforward. The only awkward point was that we often abstract a term over a set of type variables whose order used to be immaterial. Now we need to abstract over type *and kind* variables, and we must ensure that the kind variables come first, because they may appear in the kinds of the type variables.

6.2 Type inference for terms

The type inference engine needed to be extended in two main ways:

- When instantiating the type of a kind-polymorphic function, say $f :: \forall\,\mathcal{X}.\,\forall\,a{:}\,\mathcal{X}.\,\tau$ we must instantiate a fresh unification kind variable for $\mathcal{X}$, and a fresh unification type variable for a, with an appropriate kind.
- When unifying a type variable with a type we must unify its kind with the kind of the type.

All kind unifications are solved on the fly by the usual side-effecting unification algorithm, and do not generate evidence. In contrast, type equalities are often gathered as constraints to be solved later, and are witnessed by evidence in the form of coercion terms [26]. Not having to do this step for kind equalities rests on the design principle discussed in §3.4.

6.3 Kind inference for types

GHC allows the user to write type signatures, such as

$$f :: \mathsf{Typeable}\ a \Rightarrow [\mathsf{Proxy}\ a] \to \mathsf{TypeRep}$$

When elaborating this type signature, the inference engine performs kind generalization of the type, yielding the $\mathrm{F}_{\mathrm{C}}^{\uparrow}$ type

$$f : \forall\mathcal{X}.\ \forall a : \mathcal{X}.\ \mathsf{Typeable}\ \mathcal{X}\ a \to [\mathsf{Proxy}\ \mathcal{X}\ a] \to \mathsf{TypeRep}$$

Type declarations themselves are a little more complex. Consider

$$\begin{array}{l}\mathbf{data}\ \mathsf{T}\ f\ a = \mathsf{MkT}\ (\mathsf{S}\ a\ f) \mid \mathsf{TL}\ (f\ a)\\ \mathbf{data}\ \mathsf{S}\ b\ g = \mathsf{MkS}\ (\mathsf{T}\ g\ b)\end{array}$$

These two mutually recursive definitions must be kind checked together. Indeed the situation is precisely analogous to the well-studied problem of type inference for mutually recursive term definitions.

- We sort the type declarations into strongly connected components, and kind check them in dependency order.
- A type constructor can be used only monomorphically within its mutually recursive group.
- Type inference assigns each type constructor a kind variable, then walks over the definitions, generating and solving kind constraints.
- Finally, the kind of each constructor in the group is generalized.

Class declarations are handled in the same way. There is nothing new here—it all works in precisely the same way as for terms—so we omit a precise account.

7. Related work

Promoting data types The starting point for this work has been Conor McBride's *Strathclyde Haskell Enhancement* preproces-

sor [19], which, among other features, allows users to promote datatypes to be used in the type language. Throughout this paper, we have already discussed several points of both similarity and difference between our work and SHE.

The language extensions we describe here are most closely related to the LX calculus [6] and the Ωmega language [27, 28]. These languages also allow the definition of datakinds and type-level functions over datatypes. Like both of these systems, GHC maintains a phase separation: terms can depend on types, but not vice versa. In terms of expressiveness, $F_C^{\uparrow}$ is very similar to LX: $F_C^{\uparrow}$ includes kind polymorphism and does not require termination proofs for type-level computation, but lacks anonymous functions at the type-level. However, $F_C^{\uparrow}$ is less expressive than Ωmega, which allows GADT programming at the type level. Because datakinds may be indexed, Ωmega also allows the definition of datatypes in the kind language. In fact, Ωmega has a hierarchy of levels (types, kinds, superkinds, *etc.*) where each level includes datatypes that can be indexed by elements from any of the higher levels. All datatype definitions are "level-polymorphic", meaning that they can be used at any level. On the other hand, Ωmega does not compile to an explicitly typed language with evidence terms, and LX does not include explicit evidence for type equalities. This makes it harder in those languages to reason about type safety, robustness under transformations, and the soundness of type and kind inference. In $F_C^{\uparrow}$ many of the restrictions in our design (such as no evidence-based kind-equalities) are motivated by our desire to keep the explicitly typed intermediate language simple.

Our extension does not add full-spectrum dependency to Haskell. It merely makes the phase distinction less distinct. Full-spectrum languages actually allow program expressions to invade the type system, but in the presence of nontermination or other effects, often limit those expressions to a subset with a trivial definition of equality. For example, the Aura programming language [12] and Licata and Harper's Positive Dependent Types [17] only allow dependency on *values* of positive types (such as datatypes formed from strict products and sums). Alternatively, the F^* language [30] allows dependency for all *values*, but mandates a trivial definition of equality for functions. In contrast, promotion is more similar to the duplication described above. Although Haskell datatypes are nonstrict, we do not promote values, only the constructors used to form them.

Agda takes the promotion idea one step further with its experimental support for *universe polymorphism*[5]. In Agda, all definitions (including datatypes) may be polymorphic over their level. Functions that work with such datatypes may also be level polymorphic, so the same function can be used both at runtime and for type-level computation. This extension is much more sophisticated than the simple form of promotion that we describe here, and its interactions with the rest of the type system are not yet well understood.

Kind polymorphism Kind polymorphism is an often requested feature for Haskell in its own right. In fact, the Utrecht Haskell Compiler (UHC)[6] [7] already supports kind polymorphism. In particular, unknown kinds of type constructors do not default to kind $\star$, but are instead generalized. The particular motivation for this addition to UHC seems to be the ability to define a kind-polymorphic *Leibniz* equality datatype:

$$\mathbf{data}\ \mathsf{Eq}\ a\ b = \mathsf{Eq}\ (\forall f.\, f\ a \to f\ b)$$

However, UHC does not compile to a typed intermediate language like System $F_C^{\uparrow}$.

[5] `http://wiki.portal.chalmers.se/agda/agda.php?n=Main.UniversePolymorphism`

[6] `http://www.cs.uu.nl/wiki/UHC`

Kind polymorphism has also been added to typed intermediate languages, even when the source language did not include kind polymorphism. Trifonov *et al.* [32] found that the addition of kind polymorphism to the FLINT/ML compiler allowed it to be fully reflexive. In other words, they were able to extend their type-analyzing operations so that they are applicable to the type of any runtime value in the language.

More formally, the properties of a polymorphic lambda calculus with kind polymorphism were studied by Girard [8]. Girard showed that this calculus can encode a variant of the Burali-Forti paradox and is thus inappropriate for use as a constructive logic. This proof of "Girard's paradox" is described in detail by Barendregt [2]. Hurkens later simplified this paradox [11]. It is not clear whether this result holds for our system, since we do not directly allow function abstraction at the type or kind levels. However, even if it did, it would result in a non-normalizing *term*, which Haskell already has in abundance.

Kind-indexed type families The idea of allowing type functions to dispatch on kinds is a fairly novel component of this extension. Work by Morrisett *et al.* on intensional type analysis [9] included an operator that dispatched on types, but not kinds. Indeed, the extension of that work by Trifonov *et al.* [32] relied on the fact that kind polymorphism was parametric. In System F_C, we have separated the soundness of the language from the decidability of type checking [33].

Although we do not yet have many motivating examples of the use of kind-indexed type families, we still believe that they have great potential. For example, we can use them to define the *polykinded-types* [10] found in Generic Haskell. System $F_C^{\uparrow}$ cannot yet write the polytypic functions that inhabit such types, but we plan to further extend the language with such support (see §8.5).

Why not full-spectrum dependent types? One might well ask: why not just use, say, Agda [23] or Coq [3]? What benefit is there in adding dependent type features to Haskell without going all the way to a full-spectrum dependently typed language? We see a number of advantages of our evolutionary approach:

- **Type inference**. Haskell enjoys robust type inference, even for top-level terms. Adding full dependent types would severely complicate matters at best. Promotion, however, requires only slight enhancements to existing inference mechanisms (§5).
- **Phase distinction**. Since Haskell types never actually depend on terms and vice versa, types can be safely erased before runtime. This is still true for promoted values. Erasure analysis for full-spectrum languages, on the other hand, is an active area of research.
- **Explicit, strongly typed evidence**. This allows us to be certain that type and kind inference is *sound* and does not accept programs that may crash at runtime.
- **Simple type checking**. System F_C, into which GHC desugars Haskell source code, enjoys a simple, linear-time type checking algorithm, guided by the presence of explicit coercion terms. This remains true for our promoted variant. Type checking the core language for a full-spectrum dependently typed intermediate language, on the other hand, must take into account a more complex equivalence on types which includes β-equivalence.
- **Optimization**. Thanks again to the presence of explicit coercion terms, Haskell's core language can be aggressively optimized while remaining statically typed.
- **Familiarity**. Many programmers are already familiar with Haskell, and it enjoys a large collection of existing libraries and tools. Modest, backwards-compatible extensions will be adopted in ways that major, breaking changes cannot.

8. Future work

One of the main driving forces behind the current design of $F_C^{\uparrow}$ has been *simplicity*. We wanted to make sure we have all the details solidly in place before introducing any complications. There are, however, several further directions we would like to explore.

8.1 Promoting functions

If we can promote (some) term-level data constructors to type constructors, why not promote (some) term-level *functions* to type-level functions? We have not done so yet because we already have a carefully-limited way to define type-level functions, and it seems awkward to specify exactly which term-level functions should be promoted. However, the resulting code duplication is irksome, and there is no difficulty in principle, so we expect to revisit this topic in the light of experience.

8.2 Generating singleton types

When using promoted types as indices, one quickly comes to desire the ability to do case analysis on them. Of course, this is not directly possible, since types are erased before runtime. A well-known technique for overcoming this limitation is the use of *singleton types*, which allow doing case analysis on value-level representatives in lieu of types (the survey article on programming in Ωmega [28] offers an excellent introduction to this technique). We would like to avoid this additional source of code duplication by automatically generating singleton types from datatype declarations.

For instance, consider implementing a function *replicate*, which creates a Vec of a certain (statically determined) length. This can be accomplished with the help of singleton type SNat:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{SNat} :: \mathsf{Nat} \to \star\ \textbf{where} \\
\quad \mathsf{SZero} :: \mathsf{SNat}\ \mathsf{Zero} \\
\quad \mathsf{SSucc} :: \mathsf{SNat}\ n \to \mathsf{SNat}\ (\mathsf{Succ}\ n) \\
\mathit{replicate} :: \mathsf{SNat}\ n \to a \to \mathsf{Vec}\ a\ n \\
\mathit{replicate}\quad \mathsf{SZero}\qquad\ _ = \mathsf{VNil} \\
\mathit{replicate}\quad (\mathsf{SSucc}\ n)\ x = \mathsf{VCons}\ x\ (\mathit{replicate}\ n\ x)
\end{array}
$$

Automatically generating singleton types is not hard [19, 22]. More interesting would be the design of convenient surface syntax to completely hide their use, so users could just "pattern match on types".

8.3 Promoting primitive datatypes

Another useful extension would be the promotion of primitive built-in datatypes, such as integers, characters, and strings. For instance, having promoted string literals as types would allow us to go well beyond heterogeneous type-level lists to type-level *records*, as in the Ur programming language [5].

There are no implementation or theoretical difficulties with simply promoting primitive datatypes. However, our implementation does not yet promote primitive datatypes because, to make this feature truly usable, we would also like to promote primitive *operations* on these datatypes (such as integer addition or string concatenation) and integrate these operations with type inference and evidence generation. Some promising work in this direction is Diatchki's recent addition of type-level naturals to GHC.[7]

8.4 Closed type families

Type families, as currently implemented in GHC, are *open* [25]: it is always possible to add more clauses to a type family definition at any time, as long as they do not overlap with existing clauses. In a world where the only base kind is $\star$, this makes sense, since $\star$ itself is open. However, with the ability to promote closed datatypes to kinds, we now have type families which are "naturally" closed. For example, given a type family Foo with two clauses, $\mathsf{Foo}\ \mathsf{Zero} = \ldots$ and $\mathsf{Foo}\ (\mathsf{Succ}\ n) = \ldots$, it is impossible to add any more well-kinded clauses without overlap. Could the compiler somehow take advantage of this knowledge?

[7] http://hackage.haskell.org/trac/ghc/wiki/TypeNats

8.5 Kind-indexed GADTs

Recall the definition of the Typeable class with a kind-polymorphic type from §2.5:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Typeable}\ a\ \textbf{where} \\
\quad \mathit{typeOf} :: \mathsf{Proxy}\ a \to \mathsf{TypeRep}
\end{array}
$$

Notice that *typeOf*'s return type, TypeRep, does not mention a. It would be nicer to have *typed* TypeReps by using GADTs whose constructors refine types *and kinds*, for instance:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{TypeRep}\ (a :: \kappa)\ \textbf{where} \\
\quad \mathsf{RepInt}\ :: \mathsf{TypeRep}\ \mathsf{Int} \\
\quad \mathsf{RepList} :: (\forall a.\ \mathsf{TypeRep}\ a \to \mathsf{TypeRep}\ [a]) \to \mathsf{TypeRep}\ []
\end{array}
$$

Notice that RepInt must have type $\mathsf{TypeRep} \star \mathsf{Int}$ in $F_C^{\uparrow}$ whereas RepList returns $\mathsf{TypeRep}\ (\star \to \star)\ []$. However, supporting kind-indexed GADTs is tricky. Just as GADTs currently introduce type equalities, these kind-indexed GADTs could introduce *kind* equalities, an avenue that we are not prepared to take (see the discussion in §3.4).

On the other hand, there may be just enough machinery already in $F_C^{\uparrow}$ to deal with kind-indexed GADTs in a more primitive fashion than ordinary GADTs, by employing *unifiers* for kind equalities, in the same way that GADTs used to be implemented in pre-F_C times [24]. We plan to explore this direction in the future.

Acknowledgments

Thanks to Conor McBride, Tim Sheard, the members of the Penn PL Club, and the anonymous reviewers for many helpful and inspiring suggestions. This work was partially supported by the National Science Foundation (NSF grants 0910500 and 1116620).

References

[1] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM.

[2] H. P. Barendregt. *Lambda calculi with types*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 1 edition, June 2004.

[4] M. Bolingbroke. Constraint Kinds for GHC. 2011. URL http://blog.omega-prime.co.uk/?p=127.

[5] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45:122–133, June 2010. ISSN 0362-1340.

[6] K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 233–248, 1999.

[7] A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM.

[8] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM.

[10] R. Hinze. Polytypic Values Possess Polykinded Types. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, MPC '00, pages 2–27, London, UK, 2000. Springer-Verlag.

[11] A. J. C. Hurkens. A Simplification of Girard's Paradox. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 266–278, London, UK, 1995. Springer-Verlag.

[12] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 27–38, 2008.

[13] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

[15] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 204–215, New York, NY, USA, 2005. ACM.

[16] D. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *POPL '12*. To appear.

[17] D. R. Licata and R. Harper. Positively dependent types. In *PLPV '09: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-330-3.

[18] S. Marlow, editor. *Haskell 2010 Language Report*. 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`.

[19] C. McBride. The Strathclyde Haskell Enhancement. URL `http://personal.cis.strath.ac.uk/~conor/pub/she/`.

[20] C. McBride. Kleisli arrows of outrageous fortune. 2011. URL `http://personal.cis.strath.ac.uk/~conor/Kleisli.pdf`.

[21] N. P. Mendler. Predicative type universes and primitive recursion. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science, 15-18 July, 1991, Amsterdam, The Netherlands*, pages 173–184. IEEE Computer Society, 1991.

[22] S. Monnier and D. Haguenauer. Singleton types here Singleton types there Singleton Types Everywhere, 2009.

[23] U. Norell. *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007.

[24] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61. ACM Press, 2006.

[25] T. Schrijvers, S. Peyton Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 51–62. ACM Press, 2008.

[26] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM.

[27] T. Sheard. Type-level Computation Using Narrowing in Omega. *Electr. Notes Theor. Comput. Sci.*, 174(7):105–128, 2007.

[28] T. Sheard and N. Linger. Programming in Omega. In *CEFP*, pages 158–227, 2007.

[29] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66. ACM Press, 2007.

[30] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. pages 266–278.

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

[32] V. Trifonov, B. Saha, and Z. Shao. Fully Reflexive Intensional Type Analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93. ACM Press, 2000.

[33] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. 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.