System F with Type Equality Coercions

Martin Sulzmann
School of Computing
National University of Singapore
sulzmann@comp.nus.edu.sg

Manuel Chakravarty
Programming Languages and Systems
Computer Science & Engineering
chak@cse.unsw.edu.au

Simon Peyton Jones
Microsoft Research Ltd
Cambridge, England
simonpj@microsoft.com

Abstract

We introduce a variant of System F that uses a single mechanism to enable the *type preserving* translation of generalised abstract data types (GADTs), type classes with associated types and functional dependencies, as well as closed type functions. The core idea is to pass around explicit evidence for type equalities, just like System F passes types explicitly. We use this evidence to justify type casts encoding non-syntactic type equalities induced by the mentioned source language features. In particular, we don't need special typing rules for pattern matching on GADTs, we can easily combine GADTs with type classes, and we can relax restrictions on programs involving associated types or functional dependencies.

1. Introduction

GHC (the Glasgow Haskell Compiler) has a problem. It uses System $F\omega$, extended with algebraic data types and case-expressions, as its typed intermediate language. But Haskell has incubated a variety of extensions that are hard to translate into this language: in particular, functional dependencies [14], generalised algebraic data types (GADTs) [30, 21], and associated types [4, 3]. When we added GADTs to GHC, we found we had to extend GHC's intermediate language with GADTs as well, although GADTs are arguably an over-sophisticated addition to a typed intermediate language. But when it came to associated types, it was clear that we could not throw them in too without making the intermediate language unbearably baroque.

In this paper we present a solution to this dilemma. There is no need to add GADTs to System F. Instead, we add explicit type-equality coercions, and use them as witnesses to justify explicit type-cast operations. These coercions are treated just like types: terms have coercion abstraction and application, just like type abstraction and application, and we have an erasure result that allows us to discard all coercions and casts before execution.

This single mechanism allows a very direct encoding of associated types and GADTs, and allows us to deal with some exotic functional-dependency programs that GHC currently rejects on the grounds that they have no System-F translation. Our specific contributions are these:

Submitted to ICFP'06.

- We articulate the problem in Section 2, and give an informal description of our our solution.
- We give a formal description of System $\mathrm{F_C}$, our new intermediate language, including its type system, operational semantics, soundness result, and erasure properties (Section 3).
- The system is very general, but we are able to identify *consistency* as a necessary and sufficient property for soundness (Section 3.4). Conditions identified in earlier work on GADTs, associated types, and functional dependencies, are special cases of consistency.
- As one application, we give a type-preserving translation of a source language supporting GADTs into System $\mathrm{F_C}$ (Section 4). As a second, we sketch a similar translation for associated types (Section 5). The latter, and the corresponding translation for functional dependencies, are more general than all previous type-preserving translations for these features.

System $\mathrm{F_C}$ has no new foundational content: rather, it is an intriguing and practically-useful application of techniques that have been well studied in the type-theory community. We discuss this and other related work in Section 6.

Although we have written this introduction as if GHC were the only compiler in the world, there is nothing GHC-specific about System $\mathrm{F_C}$. Indeed, it occupies an interesting place in the design space. On the one hand, we know of no language smaller than $\mathrm{F_C}$ that is sufficient to serve as a target for GADTs, associated types, and functional dependencies (see also Appendix A); while on the other, $\mathrm{F_C}$ seems much more general than any of these specific source-language constructs, much as System F is far more general than ML. In fact, we believe that $\mathrm{F_C}$ may have much wider application as a typed target for sophisticated HOT (higher-order typed) source languages — a hypothesis that remains to be tested.

2. The key ideas

System F (aka the polymorphic lambda calculus), or its more general cousin $F\omega$, is popular as a highly-expressive typed intermediate language in compilers for functional languages, such as the Glasgow Haskell Compiler. Many source-language features can be desugared into System F with little difficulty, but not all. The first such feature is the algebraic data types of Haskell or ML, which are made more complicated in Haskell because algebraic data types can capture existential type variables. Whilst these can be encoded in System F, the encoding is heavy, and compilers invariably extend System F by adding algebraic data types, data constructors, and **case** expressions. We will use $\mathrm{F_A}$ to describe System F extended in this way, where the data constructors are allowed to have existential components [17].

Over the last few years, source languages (notably Haskell) have started to explore language features that embody *non-syntactic type equality*. These features include functional dependencies [10], generalised algebraic data types (GADTs) [30, 26], and associated types [4, 3]. All three are difficult or impossible to translate into System F — and yet the alternative of simply extending System F by adding functional dependencies, GADTs, and associated types, seems wildly unattractive. Where would one stop?

The paper presents System $\mathrm{F_C}$, an extension of System F that resolves the dilemma. In the rest of this section we present $\mathrm{F_C}$ informally, showing how it can serve as a target for each of the three examples, and we sketch further possibilities (such as closed type functions) that $\mathrm{F_C}$ opens up. The formal details are presented in Section 3. Throughout we use `typewriter font` for source-code fragments, and *italics* for $\mathrm{F_C}$ fragments.

2.1 GADTs

Consider the following simple type-safe evaluator, often used as the poster child of GADTs, written in the GADT extension of Haskell supported by GHC:

```
data Exp a where
  Zero :: Exp Int
  Succ :: Exp Int -> Exp Int
  Pair :: Exp b -> Exp c -> Exp (b, c)

eval :: Exp a -> a
eval Zero       = 0
eval (Succ e)   = eval e + 1
eval (Pair x y) = (eval x, eval y)

main = eval (Pair (Succ Zero) Zero)
```

The key point about this program, and the one that is hard to express in System F, is that in the `Zero` branch of `eval`, the type variable `a` is the same as `Int`, even though the two are syntactically quite different. That is why the `0` in the `Zero` branch is well-typed in a context expecting a result of type `a`.

Rather than extend the intermediate language with GADTs themselves — GHC's current "solution", incidentally — we instead propose a general mechanism of parameterising functions with *type equalities* witnessed by *coercions types,* which we call *type equality coercions.* Coercions types are passed around using System F's existing type passing facilities and enable representing GADTs by ordinary algebraic data types encapsulating such *type equality coercions.*

Specifically, we translate the GADT `Exp` to an ordinary algebraic data type, where each variant is parametrised by a coercion:

$$\begin{array}{l}\mathbf{data}\ \mathit{Exp}\ :\ \star \to \star\ \mathbf{where}\\ \quad \mathit{Zero}\ :\ \forall a.\,(a \doteq \mathit{Int})\ \Rightarrow\ \mathit{Exp}\ a\\ \quad \mathit{Succ}\ :\ \forall a.\,(a \doteq \mathit{Int})\ \Rightarrow\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ a\\ \quad \mathit{Pair}\ :\ \forall abc.\,(a \doteq (b,c))\ \Rightarrow\ \mathit{Exp}\ b\ \to\ \mathit{Exp}\ c\ \to\ \mathit{Exp}\ a\end{array}$$

So far, this is quite standard; indeed, several authors present GADTs in the source language using a syntax involving explicit equality constraints, similar to that above [30, 8]. However, for us the equality constraints are extra type arguments to the constructor, which must be given when the constructor is applied, and which are brought into scope by pattern matching. The "⇒" is syntactic sugar, and we sloppily omitted the kind of the quantified type variables, so the type of *Zero* is really this:

$$\mathit{Zero}\ :\ \forall\ a{:}\star.\ \forall(c{:}\,a \doteq \mathit{Int}).\ \mathit{Exp}\ a$$

Here a ranges over *types*, of kind $\star$, while c ranges over *coercions*, of kind $a \doteq Int$. An important property of our approach is that *coercions are types*, and hence, *equalities* $\tau_1 \doteq \tau_2$ *are kinds.* An equality kind $\tau_1 \doteq \tau_2$ categorises all coercion types that witness the equality of the two types t_1 and t_2.

Coercion types may be formed from a set of elementary coercions that correspond to the rules of equational logic; for example, $\mathrm{refl}\ \mathit{Int}\ :\ (\mathit{Int} \doteq \mathit{Int})$ is an instance of the reflexivity of equality. A call of the constructor $Zero$ must be given a type (to instantiate a) and a coercion (to instantiate c), thus for example:

$$\mathit{Zero}\ \mathit{Int}\ (\mathrm{refl}\ \mathit{Int})\ :\ \mathit{Exp}\ \mathit{Int}$$

Just like value arguments, the coercions passed to a constructor when it is built are made available again by pattern matching. Here, then is the code of `eval` in $\mathrm{F_C}$:

$$\begin{array}{l}\mathit{eval}\ =\ \Lambda a{:}\star.\lambda e{:}\mathit{Exp}\ a.\\ \quad \mathbf{case}\ e\ \mathbf{of}\\ \qquad \mathit{Zero}\ (co : a \doteq \mathit{Int})\ \to\\ \qquad\quad 0 \blacktriangleright \mathrm{sym}\ co\\ \qquad \mathit{Succ}\ (co : a \doteq \mathit{Int})\ (e' : \mathit{Exp}\ a)\ \to\\ \qquad\quad (\mathit{eval}\ \mathit{Int}\ e'\ +\ 1) \blacktriangleright \mathrm{sym}\ co\\ \qquad \mathit{Pair}\ (b{:}\star)\ (c{:}\star)\ (co : a \doteq (b,\ c))\\ \qquad\ \ (e_1 : \mathit{Exp}\ b)\ (e_2 : \mathit{Exp}\ c)\ \to\\ \qquad\quad (\mathit{eval}\ b\ e_1,\ \mathit{eval}\ c\ e_2) \blacktriangleright \mathrm{sym}\ co\end{array}$$

The form $\Lambda a{:}\star.e$ abstracts over manifest types, as usual. In the first alternative of the **case** expression, the pattern binds the coercion type argument of $Zero$ to co. We use the symmetry of equality in $(\mathrm{sym}\ co)$ to get a coercion from Int to a and use that to cast the type of 0 to a, using the *cast expression* $0 \blacktriangleright \mathrm{sym}\ co$. Cast expressions have no *operational* effect, but they serve to explain to the type system when a value of one type (here Int) should be treated as another (here a), and provide evidence for this equivalence. In general, the form $e \blacktriangleright g$ has type t_2 if $e\ :\ t_1$ and $g\ :\ (t_1 \doteq t_2)$. So, $\mathit{eval}\ \mathit{Int}\ (\mathit{Zero}\ \mathit{Int}(\mathrm{refl}\ \mathit{Int}))$ is of type Int as required by $eval$'s signature. We shall discuss coercion types and their kinds in more detail in Section 3.2.

2.2 Associated types

Associated types are a recently-proposed extension to Haskell's type-class mechanism [4, 3]. They offer open, type-indexed types that are associated with a type class. Here is a standard example:

```
class Collects c where
  type Elem c        -- associated type synonym
  empty  :: c
  insert :: Elem c -> c -> c
```

The class `Collects` abstracts over a family of containers, where the representation type of the container, `c`, defines (or constrains) the type of its elements `Elem c`. That is, `Elem` is a type-level function that transforms the collection type to the element type. Just as `insert` is non-parametric – its implementation varies depending on `c` – so is `Elem`. For example, a list container can contain elements of any type supporting equality, and a bit-set container might represent a collection of characters:

```
instance Eq e => Collects [e] where
    {type Elem [e] = e; ...}
instance Collects BitSet where
    {type Elem BitSet = Char; ...}
```

Generally, type classes are translated into System F [11] by (1) turning each class into a record type, called a dictionary, containing the class methods, (2) converting each instance into a dictionary value, and (3) passing such dictionaries to whichever function mentions a class in its signature. For example, a function of type `negate :: Num a => a -> a` will translate to $\mathit{negate}\ :\ \mathit{NumDict}\ a\ \to\ a\ \to\ a$, where $NumDict$ is the record generated from the class `Num`.

A record only encapsulates values, so what to do about associated types, such as `Elem` in the example? The system given in [4] translates each associated type into an additional type parameter of the class's dictionary type, provided the class and instance declarations abide by some moderate constraints [4]. For example, the class `Collects` translates to dictionary type $\mathit{CollectsDict}\ c\ e$, where e represents $\mathit{Elem}\ c$ and where all occurrences of $\mathit{Elem}\ c$ of the method signatures have been replaced by the new type parameter e. So, the (System F) type for insert would now be $\mathit{CollectDict}\ c\ e \rightarrow e \rightarrow c \rightarrow c$. The required type transformations become more complex when associated types occur in data types; the data types have to be rewritten substantially during translation, which can be a considerable burden in a compiler.

Type equality coercions enable a far more direct translation. Here is the translation of `Collects` into $\mathrm{F_C}$:

$$\begin{array}{l}\mathbf{type}\ \mathit{Elem}\ :\ \star \rightarrow \star\\ \mathbf{data}\ \mathit{CollectsDict}\ c\ =\\ \quad \mathit{Collects}\ \{\mathit{empty}\ :\ c;\ \mathit{insert}\ :\ \mathit{Elem}\ e\ \rightarrow\ c\ \rightarrow\ c\}\end{array}$$

as in a translation without associated types. The **type** declaration in $\mathrm{F_C}$ introduces a new *abstract type constructor* — abstract in the sense that its representation is not given. An instance declaration for `Collects` is translated to (a) a dictionary transformer for the values and (b) a coercion constant that describes (part) of the representation for `Elem`. For example, here is the translation into $\mathrm{F_C}$ of the `Collects [e]` instance:

$$\begin{array}{l}\mathbf{coercion}\ \mathit{collectsList}\ :\ \Pi e{:}\star.\mathit{Elem}\ [e] \doteq e\\ \mathit{dCollectsList}\ :\ \forall e{:}\star.\mathit{EqDict}\ e\ \rightarrow\ \mathit{CollectsDict}\ [e]\\ \mathit{dCollectsList}\ =\ \ldots\end{array}$$

The top-level **coercion** declaration defines a new, named *coercion constant, collectList*, which serves as a witness that we can cast types of the form $\mathit{Elem}\ [e]$ to e and vice versa. Using this coercion, we can insert the character 'b' into a list of characters ['a'] by applying the instantiated coercion $\mathit{collectsList}\ \mathit{Char}$ backwards to 'b'; i.e., in $\mathit{insert}(\text{'b'} \blacktriangleright \mathrm{sym}\ (\mathit{collectsList}\ \mathit{Char}))\ [\text{'a'}]$, we have $(\text{'b'} \blacktriangleright \mathrm{sym}\ (\mathit{collectsList}\ \mathit{Char}))\ :\ \mathit{Elem}\ [\mathit{Char}]$, which fits the signature of insert.

In short, System $\mathrm{F_C}$ supports a very direct translation of associated types, in contrast to the clumsy one described in [4]. What is more, there are several obvious extensions to the latter paper that cannot be translated at all, even clumsily, and $\mathrm{F_C}$ supports them too, as we sketch in Section 5.

2.3 Functional dependencies

Functional dependencies are another popular extension of Haskell's type-class mechanism [14]. With functional dependencies, we can encode a function over types `F` as a relation; i.e., the dependent type parameters are not unlike associated types:

```
class F a b | a -> b
instance F Int Bool
```

A useful idiom in type-level programming is to abstract over the co-domain of a type function by way of an existential type, the `b` in this example:

```
data T a = forall b. F a b => MkT (b -> b)
```

Then one might hope that the following function would type-check:

```
combine :: T a -> T a -> T a
combine (MkT f) (MkT f') = MkT (f . f')
```

After all, since the type `a` functionally determines `b`, `f` and `f'` must have the same type. Yet GHC rejects this program, because it cannot be translated into System $\mathrm{F_A}$, because `f` and `f'` each have distinct, existentially-quantified types, and there is no way to express their (non-syntactic) identity in $\mathrm{F_A}$.

It is easy to translate this example into $\mathrm{F_C}$, however:

$$\begin{array}{l}\mathbf{type}\ F1\ :\ \star \rightarrow \star\\ \mathbf{data}\ \mathit{FDict}\ :\ \star \rightarrow \star \rightarrow \star\ \mathbf{where}\\ \quad F\ :\ \forall a\ b.\,(b \doteq F1\ a)\ \Rightarrow\ \mathit{FDict}\ a\ b\\ \mathbf{coercion}\ \mathit{fIntBool}\ :\ F\ \mathit{Int} \doteq \mathit{Bool}\end{array}$$

$$\begin{array}{l}\mathbf{data}\ T : \star \rightarrow \star\ \mathbf{where}\\ \quad \mathit{MkT}\ :\ \forall a\ b.\mathit{FDict}\ a\ b\ \rightarrow\ (b\ \rightarrow\ b)\ \rightarrow\ T\ a\\ \\ \mathit{combine}\ :\ T\ a\ \rightarrow\ T\ a\ \rightarrow\ T\ a\\ \mathit{combine}\ (\mathit{MkT}\ b\ \ (F\ (c\ :\ b \doteq F1\ a))\ f)\\ \qquad\quad (\mathit{MkT}\ b'\ \ (F\ (c' : b' \doteq F1\ a))\ f')\\ \quad =\ \mathit{MkT}\ b\ (F\ c)\ (f\ .\ (f' \blacktriangleright d_2))\\ \quad \mathbf{where}\\ \qquad d_1\ :\ (b \doteq b') \qquad\qquad\quad =\ c \rhd \mathrm{sym}\ c'\\ \qquad d_2\ :\ (b\ \rightarrow\ b \doteq b'\ \rightarrow\ b') =\ (\rightarrow) \odot d_1 \odot d_1\end{array}$$

The functional dependency is expressed as an abstract type function $F1$ and leads to one top-level coercion declaration per instance. (In general there might be many functional dependencies for a single class.) The dictionary for class `F` includes a witness that indeed b is equal to $F1\ a$, as you can see from the declaration of constructor F. When pattern matching in $\mathit{combine}$ we gain access to these witnesses, and can use them to cast f' so that it has the same type as f. (When constructing the witnesses d_1 and d_2 we use the coercion combinators $\mathrm{sym}\ \cdot$, $\cdot \rhd \cdot$, and $\cdot \odot \cdot$, which represent the symmetry, transitivity, and ability to compose under equality, respectively.)

Even in the absence of existential types, there are reasonable source programs involving functional dependencies that have no System F translation, and hence are rejected by GHC. We have encountered this problem in real programs, but here is a boiled-down example, using the same class `F` as before:

```
class D a where { op :: F a b => a -> b }
instance D Int where { op _ = True }
```

The crucial point is that the context `F a b` of the signature of `op` constrains the parameter of the enclosing type class `D`. This becomes a problem when typing the definition of `op` in the instance `D Int`. In `D`'s dictionary DDict, we have $op\ :\ \forall b.C\ a\ b\ \rightarrow\ a\ \rightarrow b$ with b universally quantified, but in the instance declaration, we would need to instantiate b with Bool. The instance declaration for `D` cannot be translated into System F. Using $\mathrm{F_C}$, this problem is easily solved. The coercion in the dictionary for `F` enables the result of `op` to be cast to type b as required.

To summarise, a compiler that uses translation into System F (or $\mathrm{F_A}$) must reject some reasonable (albeit slightly exotic) programs involving functional dependencies, and also similar programs involving associated types. The extra expressiveness of System $\mathrm{F_C}$ solves the problem neatly.

2.4 Closed type functions

Associated types constitute *open,* that is extensible, type functions. Especially in conjunction with GADTs, *closed* type functions are also desirable, as argued in the proposal for Ωmega [26]. To review the basic idea, consider the following definition of bounded sequences whose length is encoded in their type:

```
kind Nat = Z | S Nat

data Seq a (n :: Nat) where
  Nil  :: Seq a Z
  Cons :: a -> Seq a n -> Seq a (S n)
```

We define bounded sequences with a GADT parametrised by a kind `Nat` encoding Peano numerals with two new type constructors `Z :: Nat` and `S :: Nat -> Nat`. Now, what is the signature of a function appending one such sequence at another? The length of the resulting sequence is clearly the sum of the lengths of the two component sequences. Hence, we want to write something along the following lines:

```
app :: Seq a n -> Seq a m -> Seq a (Plus n m)
app Nil         ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

Here `Plus` is a closed type function:

```
type Plus Z     b = b
     Plus (S a) b = S (Plus a b)
```

All of this has a very direct translation into $\mathrm{F_C}$. The type function `Plus` becomes an abstract type constructor; the equations for `Plus` become coercion constants; and the GADT for `Seq` is encoded as in Section 2.1. $\mathrm{F_C}$ does not support the declaration of algebraic data kinds, such as `Nat`, but that is a orthogonal extension that could easily be added. (Or, perhaps better, one could re-use types as kinds, but that is another story.) In a similar manner, programs including the recently proposed extended algebraic data types [28], which add equality and predicate constraints to GADTs, can be translated to $\mathrm{F_C}$.

2.5 Summary

In this section we have shown that System F is inadequate as a typed intermediate language for source languages that embody non-syntactic type equality — and Haskell has developed several such extensions. We have sketchily introduced System $\mathrm{F_C}$ as a solution to these difficulties. In the next section we describe $\mathrm{F_C}$ in detail.

3. System $\mathrm{F_C}$

This section introduces System $\mathrm{F_C}$ formally. The main idea is that we pass around explicit evidence for type equalities, in just the same way that System F passes types explicitly. Indeed, in $\mathrm{F_C}$ the evidence γ for a type equality *is* a type; we use type abstraction for evidence abstraction, and type application for evidence application. Ultimately we erase all types before running the program, and doing so thereby erase all type-equality evidence as well.

Figure 1 defines the syntax of System $\mathrm{F_C}$, while Figure 2 gives its static semantics. The notation $\overline{a}^n$ means the sequence $a_1 \cdots a_n$; the "n" may be omitted when it is unimportant. Moreover, we use comma to mean sequence extension as follows: $\overline{a}^n, a_{n+1} \triangleq \overline{a}^{n+1}$. We use $fv(x)$ to denote the free variables of a structure x, which maybe an expression, type term, or environment.

3.1 Conventional features

System $\mathrm{F_C}$, as presented here, is based on the predicative fragment of System $F\omega$. The syntax of types and kinds is given in Figure 1. To keep matters simple, the system of Figure 1 stratifies types into polytypes, σ, and monotypes, τ; and insists that all type applications (both in terms and types) involve only monotypes. We are confident that this restriction can readily be lifted (allowing us to collapse the distinction between τ and σ) — but doing so involves a few extra constructs and type rules that distract from our main contribution.[1] However, like $F\omega$ (and Haskell), our system does allows types of higher kind; hence the type application form $\tau_1\ \tau_2$.

Value type constructors T range over (a) the built-in function type constructor, (b) any other built-in types, such as Int, or (c) algebraic data types. We regard a function type $\sigma_1 \to \sigma_2$ as the curried application of a built-in function type constructor to two arguments, thus $(\to)\,\sigma_1\,\sigma_2$. Furthermore, although we give the syntax of arrow types and quantified types in an uncurried way, we also sometimes use equivalent curried notation, thus:

$$\begin{array}{rcl} \overline{\tau}^n \to \upsilon & \equiv & \tau_1 \to \cdots \to \tau_n \to \upsilon \\ \forall \overline{\alpha}^n.\tau & \equiv & \forall \alpha_1 \cdots \forall \alpha_n.\tau \end{array}$$

An *algebraic data type* T is introduced by a top-level $\mathbf{data}$ declaration, which also introduces its *data constructors*. The type of a data constructor K takes the form

$$K : \forall \overline{a{:}\kappa}^n . \forall \overline{b{:}\iota}.\ \overline{\sigma} \to T\ \overline{a}^n$$

The first n quantified type variables $\overline{a}$ appear in the same order in the return type $T\ \overline{a}$. The remaining quantified type variables bind either existentially quantified type variables, or (as we shall see) coercions.

Types are classified by *kinds* κ, using the $\vdash_{ty}$ judgement in Figure 2. Temporarily ignoring the kind $\tau_1 \doteq \tau_2$, the structure of kinds is conventional: $\star$ is the kind of manifest types (that is, the types that a term can have), while the arrow kind $\kappa_1 \to \kappa_2$ is syntactic sugar for the dependent kind $\Pi\_ : \kappa_1 . \kappa_2$. The general case of dependent kinds will be introduced in Section 3.3; meanwhile, its

[1] The standard translation of type classes produces rank-2 types and GHC currently even supports rank-n types, so practical applications of $\mathrm{F_C}$ certainly require the full version.

Symbol Classes

$$\begin{array}{lcl}
a, b, c & \to & \langle \text{type variable} \rangle \\
x, f & \to & \langle \text{term variable} \rangle \\
k & \to & \langle \text{coercion constant} \rangle \\
T & \to & \langle \text{value type constructor} \rangle \\
S_n & \to & \langle n\text{-ary abstract type constructor} \rangle \\
K & \to & \langle \text{data constructor} \rangle
\end{array}$$

Declarations

$$\begin{array}{lcl}
pgm & \to & \overline{decl};\ e \\
decl & \to & \mathbf{data}\ T : \overline{\kappa} \to \star\ \mathbf{where}\ \overline{K : \forall \overline{a{:}\kappa}. \forall \overline{b{:}\iota}.\ \overline{\tau} \to T\ \overline{a}} \\
 & \mid & \mathbf{type}\ S_n : \overline{\kappa}^n \to \iota \\
 & \mid & \mathbf{coercion}\ k : \kappa
\end{array}$$

Kinds

$$\begin{array}{lcll}
\delta & \to & \Diamond \mid \Box & \text{Super-kinds} \\
\kappa, \iota & \to & \star & \text{Manifest type} \\
 & \mid & \Pi a{:}\kappa_1 .\, \kappa_2 & \text{Dependent kinds} \\
 & \mid & \tau_1 \doteq \tau_2 & \text{Coercion kind}
\end{array}$$

Types

$$\begin{array}{llcl}
\text{Polytypes} & \sigma & \to & \forall \overline{a{:}\kappa}.\, \tau \\
\text{Monotypes} & \tau, \upsilon, \rho, \varepsilon & \to & a \mid \gamma \mid T \mid \tau_1\ \tau_2 \mid S_n\ \overline{\tau}^n
\end{array}$$

Coercions

$$\begin{array}{lcll}
d & \to & a \mid k & \text{Variables and constants} \\
\gamma & \to & d \mid \mathrm{refl}\ \tau \mid \mathrm{sym}\ \gamma \mid \gamma_1 \rhd \gamma_2 & \\
 & \mid & \gamma_1 \odot \gamma_2 \mid S_n \odot \overline{\gamma}^n \mid \mathrm{left}\ \gamma \mid \mathrm{right}\ \gamma &
\end{array}$$

Syntactic sugar

$$\begin{array}{llcl}
\text{Kinds} & \kappa_1 \to \kappa_2 & \equiv & \Pi_{:}\kappa_1 .\, \kappa_2 \\
\text{Types} & \kappa \Rightarrow \sigma & \equiv & \forall_{:}\kappa .\, \sigma
\end{array}$$

Terms

$$\begin{array}{lcll}
u & \to & x \mid K & \text{Variables and data constructors} \\
e & \to & u & \\
 & \mid & \Lambda a{:}\kappa.\, e \mid e\ \tau & \text{Type abstraction/application} \\
 & \mid & \lambda x{:}\tau.\, e \mid e_1\ e_2 & \text{Term abstraction/application} \\
 & \mid & \mathbf{let}\ x{:}\sigma = e_1\ \mathbf{in}\ e_2 & \\
 & \mid & \mathbf{case}\ e_1\ \mathbf{of}\ \overline{p \to e_2} & \\
 & \mid & e \blacktriangleright \gamma & \text{Cast} \\
 & & & \\
p & \to & K\ \overline{b{:}\iota}\ \overline{x{:}\tau} & \text{Pattern}
\end{array}$$

Environments

$$\Gamma \ \to\ \epsilon \mid \Gamma, u{:}\sigma \mid \Gamma, d{:}\kappa \mid \Gamma, T{:}\kappa \mid \Gamma, S_n{:}\kappa$$

Figure 1: Syntax of System $\mathrm{F_C}$

$$\boxed{\Gamma \vdash_k \kappa : \delta}$$

$$\text{(Star)}\ \frac{}{\Gamma \vdash_k \star : \Box} \qquad \text{(EqK)}\ \frac{\Gamma \vdash_k \kappa : \Box \quad \Gamma \vdash_{ty} \tau_1 : \kappa \quad \Gamma \vdash_{ty} \tau_2 : \kappa}{\Gamma \vdash_k \tau_1 \doteq \tau_2 : \Diamond} \qquad \text{(ForallK)}\ \frac{\Gamma \vdash_k \kappa_1 : \Box \quad \Gamma, a{:}\kappa_1 \vdash_k \kappa_2 : \delta_2}{\Gamma \vdash_k \Pi a{:}\kappa_1.\,\kappa_2 : \delta_2}$$

$$\boxed{\Gamma \vdash_{ty} \sigma : \kappa}$$

$$\text{(TVar)}\ \frac{d : \kappa \in \Gamma}{\Gamma \vdash_{ty} d : \kappa} \qquad \text{(TCon)}\ \frac{T : \kappa \in \Gamma}{\Gamma \vdash_{ty} T : \kappa} \qquad \text{(SCon)}\ \frac{S_n : \overline{\kappa}^n \to \iota \in \Gamma \quad \Gamma \vdash_{ty} \overline{\tau : \kappa}^n}{\Gamma \vdash_{ty} S_n\, \overline{\tau}^n : \iota}$$

$$\text{(AllT)}\ \frac{\Gamma \vdash_k \kappa_1 : \delta_1 \quad \Gamma, a : \kappa_1 \vdash_{ty} \sigma : \kappa_2 \quad a \notin fv(\Gamma)}{\Gamma \vdash_{ty} \forall a{:}\kappa_1.\,\sigma : \kappa_2} \qquad \text{(AppT)}\ \frac{\Gamma \vdash_{ty} \tau_1 : \Pi a{:}\kappa_1.\,\kappa_2 \quad \Gamma \vdash_{ty} \tau_2 : \kappa_1}{\Gamma \vdash_{ty} \tau_1\ \tau_2 : \kappa_2[\tau_2/a]}$$

$$\text{(Refl)}\ \frac{\Gamma \vdash_{ty} \tau : \kappa}{\Gamma \vdash_{ty} \text{refl}\ \tau : \tau \doteq \tau} \qquad \text{(Sym)}\ \frac{\Gamma \vdash_{ty} \gamma : \tau_1 \doteq \tau_2}{\Gamma \vdash_{ty} \text{sym}\ \gamma : \tau_2 \doteq \tau_1} \qquad \text{(Trans)}\ \frac{\Gamma \vdash_{ty} \gamma_1 : \tau_1 \doteq \tau_2 \quad \Gamma \vdash_{ty} \gamma_2 : \tau_2 \doteq \tau_3}{\Gamma \vdash_{ty} \gamma_1 \rhd \gamma_2 : \tau_1 \doteq \tau_3}$$

$$\text{(Comp)}\ \frac{\Gamma \vdash_{ty} \gamma_1 : \tau_1 \doteq \upsilon_1 \quad \Gamma \vdash_{ty} \gamma_2 : \tau_2 \doteq \upsilon_2}{\Gamma \vdash_{ty} \gamma_1 \odot \gamma_2 : \tau_1\ \tau_2 \doteq \upsilon_1\ \upsilon_2} \qquad \text{(SComp)}\ \frac{\Gamma \vdash_{ty} \overline{\gamma_i : \tau_i \doteq \upsilon_i}^n}{\Gamma \vdash_{ty} S_n \odot \overline{\gamma}^n : S_n\ \overline{\tau}^n \doteq S_n\ \overline{\upsilon}^n}$$

$$\text{(DLeft)}\ \frac{\Gamma \vdash_{ty} \gamma : \tau_1\ \tau_2 \doteq \upsilon_1\ \upsilon_2}{\Gamma \vdash_{ty} \text{left}\ \gamma : \tau_1 \doteq \upsilon_1} \qquad \text{(DRight)}\ \frac{\Gamma \vdash_{ty} \gamma : \tau_1\ \tau_2 \doteq \upsilon_1\ \upsilon_2}{\Gamma \vdash_{ty} \text{right}\ \gamma : \tau_2 \doteq \upsilon_2}$$

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

$$\text{(Var)}\ \frac{u : \sigma \in \Gamma}{\Gamma \vdash_e u : \sigma} \qquad \text{(Case)}\ \frac{\Gamma \vdash_e e : \tau \quad \overline{\Gamma \vdash_p p \to e : \tau \to \sigma}}{\Gamma \vdash_e \textbf{case}\ e\ \textbf{of}\ \overline{p \to e} : \sigma} \qquad \text{(Let)}\ \frac{\Gamma \vdash_e e_1 : \sigma_x \quad \Gamma, x{:}\sigma \vdash_e e_2 : \sigma}{\Gamma \vdash_e \textbf{let}\ x{:}\sigma_x = e_1\ \textbf{in}\ e_2 : \sigma}$$

$$\text{(Cast)}\ \frac{\Gamma \vdash_e e : \tau_1 \quad \Gamma \vdash_{ty} \gamma : \tau_1 \doteq \tau_2}{\Gamma \vdash_e e \blacktriangleright \gamma : \tau_2}$$

$$\text{(Abs)}\ \frac{\Gamma \vdash_{ty} \tau_1 : \star \quad \Gamma, x : \tau_1 \vdash_e e : \tau_2}{\Gamma \vdash_e \lambda x{:}\tau_1.\,e : \tau_1 \to \tau_2} \qquad \text{(App)}\ \frac{\Gamma \vdash_e e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_e e_2 : \tau_1}{\Gamma \vdash_e e_1\ e_2 : \tau_2}$$

$$\text{(TAbs)}\ \frac{\Gamma \vdash_k \kappa : \delta \quad \Gamma, a : \kappa \vdash_e e : \sigma \quad a \notin fv(\Gamma)}{\Gamma \vdash_e \Lambda a{:}\kappa.\,e : \forall a.\sigma} \qquad \text{(TApp)}\ \frac{\Gamma \vdash_e e : \forall a : \kappa.\sigma_1 \quad \Gamma \vdash_{ty} \tau_2 : \kappa}{\Gamma \vdash_e e\ \tau_2 : \sigma_1[\tau_2/a]}$$

$$\boxed{\Gamma \vdash_p p \to e : \tau \to \sigma_r}$$

$$\text{(Alt)}\ \frac{K : \forall \overline{a{:}\kappa}.\forall \overline{b{:}\iota}.\overline{\tau} \to T\ \overline{a} \in \Gamma \quad \theta = [\overline{\upsilon/a}] \quad \Gamma, \overline{b{:}\theta(\iota)}, \overline{x{:}\theta(\tau)} \vdash_e e : \sigma_r}{\Gamma \vdash_p K\ \overline{b{:}\theta(\iota)}\ \overline{x{:}\theta(\tau)} \to e : T\ \overline{\upsilon} \to \sigma_r}$$

$$\boxed{\Gamma \vdash decl : \Gamma'}$$

$$\text{(Data)}\ \frac{\overline{\Gamma \vdash_{ty} \phi : \star} \quad \Gamma \vdash_k \kappa : \Box}{\Gamma \vdash (\textbf{data}\ T{:}\kappa\ \textbf{where}\ \overline{K{:}\phi}) : (T{:}\kappa, \overline{K{:}\phi})}$$

$$\text{(Type)}\ \frac{\Gamma \vdash_k \kappa : \Box}{\Gamma \vdash (\textbf{type}\ S : \kappa) : (S{:}\kappa)} \qquad \text{(Coerce)}\ \frac{\Gamma \vdash_k \kappa : \Diamond}{\Gamma \vdash (\textbf{coercion}\ k : \kappa) : (k{:}\kappa)}$$

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

$$\text{(Pgm)}\ \frac{\overline{\Gamma \vdash decl : \Gamma_d} \quad \Gamma = \Gamma_0, \overline{\Gamma_d} \quad \Gamma \text{ is consistent} \quad \Gamma \vdash e : \sigma}{\Gamma_0 \vdash \overline{decl};\ e : \sigma}$$

Figure 2: Typing rules for System $\mathrm{F_C}$

form is conventional, as is its use in the kinding rules for types, Rule (AppT). Finally, the rules for judgements of the form $\Gamma \vdash_k \kappa$ ensure the well-formedness of kinds.

The syntax of terms is largely conventional, as are their type rules which take the form $\Gamma \vdash_e e : \sigma$. As in $F\omega$, every binder has an explicit type annotation, and type abstraction and application are also explicit. As mentioned above, the system is predicative; that is, the argument of a type application is a monotype.

There is a built-in **case** expression to take apart values built with data constructors. The patterns of a case expression are flat — there are no nested patterns — and bind only the existential type variables and coercions. For example, suppose

$$K : \forall a{:}\star. \forall b{:}\star.\, a \to b \to (b \to Int) \to T\ a$$

Then a **case** expression that deconstructs K would have the form

$$\mathbf{case}\ e\ \mathbf{of}\ K\ (b{:}\star)\ (v{:}a)\ (x{:}b)\ (f{:}b \to Int) \to e'$$

Notice that only the existential type variable b is bound in the pattern, because a is implicit in the type of e.

3.2 Type Equality Coercions

We now describe the unconventional features of our system. To begin with, consider the fragment of System F_C that completely omits abstract type constructors S, and top-level **type** and **coercion** declarations. This fragment is sufficient to serve as a target for translating GADTs, and so is of interest in its own right. We return to abstract type constructors in Section 3.3.

The essential addition to plain System $F\omega$ (beyond algebraic data types) is an infrastructure to declare, pass, and apply *type-equality constraints,* also called *coercions*. In System F_C, a coercion, γ, is a special sort of type whose kind takes the unusual form $\tau_1 \doteq \tau_2$. This kind indicates that the coercion takes an expression of type τ_1 to an expression of type τ_2. More concretely, we can use a coercion $\gamma : (\tau_1 \doteq \tau_2)$, to cast an expression $e : \tau_1$ to type τ_2 using the *cast expression* $(e \blacktriangleright \gamma)$ (see Rule (Cast) in Figure 2).

Since coercions are types, type abstraction and application also serve for coercion abstraction and application. Type abstractions and applications have no computational meaning, and may be erased before run-time, as can cast expressions (Section 3.6). There is absolutely no run-time penalty for coercions; like types, they simply provide a statically-checkable guarantee that there will be no run-time crash.

This approach gives F_C a slightly unusual type structure, because a coercion γ (that is, evidence for a type equality) is a *type* whose *kind* describes the types it connects. Hence kinds mention types, as the syntax in Figure 1 shows. The type of a term often has the form $\forall g{:}(\tau_1 \doteq \tau_2).\sigma$, where σ does not mention g — indeed, the kind system would reject any mention of g in σ. We allow syntactic sugar for this case, writing it thus: $(\tau_1 \doteq \tau_2) \Rightarrow \sigma$ (Figure 1).

More complex coercions can be built by combining or transforming other coercions. Each variant in the syntax of coercion expressions γ (Figure 1) corresponds to an inference rule of equational logic. We have the reflexivity of equality for a given type σ (i.e., $\mathrm{refl}\ \sigma : (\sigma \doteq \sigma)$), symmetry '$\mathrm{sym}\ \gamma$', transitivity '$\gamma_1 \rhd \gamma_2$', type composition '$\gamma_1 \odot \gamma_2$', and decomposition '$\mathrm{left}\ \gamma$' and '$\mathrm{right}\ \gamma$'. The typing rules for these coercion expressions are given in Figure 2.

Here is an example, taken from Section 2. Suppose a GADT $Expr$ has a constructor $Succ$ with type

$$Succ : \forall a{:}\star.(a \doteq Int) \Rightarrow Exp\ Int \to Exp\ a$$

(notice the use of the syntactic sugar $\kappa \Rightarrow \sigma$). Then we can construct a value of type $Exp\ Int$ thus:

$$Succ\ Int\ (\mathrm{refl}\ Int)\ 3$$

The coercion argument ($\mathrm{refl}\ Int$) has kind $(Int \doteq Int)$ (rule (Refl)), and serves as a witness that indeed Int is the same type as Int. More interestingly, here is a function that decomposes a value of type $Exp\ a$:

$$\begin{aligned}
&foo\ :\ \forall\, a{:}\star.\ Exp\ a\ \to\ a\ \to\ a \\
&\ \ =\ \Lambda a{:}\star.\ \lambda e{:}Exp\ a.\ \lambda\, x{:}a. \\
&\quad \mathbf{case}\ e\ \mathbf{of} \\
&\qquad Succ\ (g : a \doteq Int)\ (e' : Exp\ Int)\ \to \\
&\qquad\quad (foo\ Int\ e'\ 0\ +\ (x \blacktriangleright g)) \blacktriangleright \mathrm{sym}\ g
\end{aligned}$$

The **case** pattern binds the coercion g, which provides evidence that a and Int are the same type. This evidence is needed twice, once to cast $x : a$ to Int, and once to coerce the Int result back to a, via the coercion ($\mathrm{sym}\ g$).

Coercion composition and decomposition allows us to move coercions in and out of algebraic data types freely. For example, if we have a coercion $g : \tau_1 \doteq \tau_2$ then the coercion $((\mathrm{refl}\ Tree) \odot g)$ is evidence that $Tree\ \tau_1 \doteq Tree\ \tau_2$. Dually, if we have $h : Tree\ \tau_1 \doteq Tree\ \tau_2$, then ($\mathrm{right}\ h$) is evidence that $\tau_1 \doteq \tau_2$. The soundness of this latter coercion relies, of course, on the fact that algebraic type are injective, so that $Tree\ \tau_1 = Tree\ \tau_2$ iff $\tau_1 = \tau_2$. Notice, too, that $\mathrm{refl}\ Tree$ is a coercion that relates two types of higher kind.

3.3 Abstract types

Our last step extends the power of F_C, by adding *abstract types*. An abstract type S_n is introduced by a top-level **type** declaration, which specifies its *kind* $\overline{\kappa}^n \to \iota$, but says nothing about its *representation*. The index n indicates the *arity* of S. The syntax of types requires that S_n always appears applied to its full complement of n arguments (Section 3.4 explains why).

We use the term "abstract type" because our intent is that $S\ \overline{\sigma}$ stands for some concrete representation type, but is nevertheless a distinct type. For example, the type $Elem\ \tau$, from Section 2,is the type of elements of arrays of type τ.

An abstract type is given its representation by means of one or more top-level **coercion** declarations. Each coercion declaration introduces a named (type) constant, whose kind specifies the types equated by the coercion. For example,

$$\mathbf{coercion}\ elemBitSet\ :\ Elem\ BitSet \doteq Char$$

introduces the named coercion $elemBitSet$. Given an expression $e : Elem\ BitSet$, we can use the coercion as in the cast $e \blacktriangleright elemBitSet$, which is of type $Char$.

One often wants to write polymorphic coercions, thus:

$$\mathbf{coercion}\ elemList\ :\ \Pi a{:}\star.\ Elem\ [a] \doteq a$$

and that is the reason that F_C includes dependent kinds $\Pi a{:}\kappa_1.\,\kappa_2$. To use this coercion, say, for lists of integers, we need to apply it to a type argument:

$$elemList\ Int : (Elem\ [Int] \doteq Int)$$

As usual, dependent products $\Pi\_{:}\kappa_1.\,\kappa_2$ that do not mention the bound type variable in the second component, degenerate to arrow kinds $\kappa_1 \to \kappa_2$ (see the syntactic sugar in Figure 1).

3.4 Consistency and saturation

In System F_C, one simply declares top-level coercion constants, without supplying any constructive evidence for them. *So what is to prevent us declaring an unsound coercion constant?* For example, one could easily write a program that would crash, using the following constant:

$$\textbf{coercion } \mathit{utterlybogus} \ : \ \mathit{Int} \doteq \mathit{Bool}$$

(where Int and $Bool$ are both algebraic data types). There are many *ad hoc* ways to restrict the system to exclude such cases, the most general one is this: we require that the top-level environment, formed from the top-level declarations, is *consistent*

DEFINITION 1 (Consistency). Γ *is* consistent *iff*

$$\Gamma \vdash_{ty} \gamma : T_1\ \overline{\tau_1} \doteq T_2\ \overline{\tau_2} \quad \textit{implies} \quad T_1 = T_2$$

That is, if there is a coercion connecting two *value* types — algebraic data types, built-in types, or functions — then the outermost type constructors must be the same.[2] For example, there can be no coercion of type $Bool \doteq Int$. It is clear that the condition is necessary for soundness, and it turns out (the proof appears below) that it is also sufficient. Rule (Pgm) in Figure 2 enforces consistency.

We remarked earlier that applications of abstract type constructors S_n are required to be saturated. The reason for this insistence is, again, consistency. We definitely want to allow abstract types to be non-injective; for example:

$$\begin{array}{l}\textbf{coercion } c1 \ : \ S\ \mathit{Int} \doteq \mathit{Bool} \\ \textbf{coercion } c2 \ : \ S\ \mathit{Bool} \doteq \mathit{Bool}\end{array}$$

Here, both $S\ Int$ and $S\ Char$ are represented by the $Bool$ type. But now we can form the coercion $(c1 \odot (\mathrm{sym}\ c2))$ which has type $S\ \mathit{Int} \doteq S\ \mathit{Bool}$, and from that we must not be able to deduce (via right) that $\mathit{Int} \doteq \mathit{Bool}$, because that would violate consistency! Applications of abstract types are syntactically distinguished so that right and left apply only to ordinary type application (rules (DLeft) and (DRight) in Figure 2), and not to applications of abstract types. The same restriction prevents a partial application of an abstract type appearing in a type application, thereby instantiating a type variable with a partial application — in effect, type variables of higher kind range only over injective type constructors.

However, it perfectly acceptable for an abstract type to have an arity of 1, say, but a higher kind of $\star \to \star \to \star$. For example:

$$\begin{array}{l}\textbf{type } S \ : \ \star \ \to \ \star \ \to \ \star \\ \textbf{coercion } c1 \ : \ \mathit{HS}\ \mathit{Int} \doteq \mathit{List} \\ \textbf{coercion } c2 \ : \ \mathit{HS}\ \mathit{Bool} \doteq \mathit{Maybe}\end{array}$$

An application of HS to one type is saturated, although it has kind $\star \to \star$ and can be applied (via ordinary type application) to another type.

3.5 Dynamic semantics and soundness

The operational semantics of $\mathrm{F_C}$ is shown in Figure 3. In the expression reductions we omit the type annotations on binders to save clutter, but that is mere abbreviation.

An unusual feature of our system is that values are stratified into *cvalues* and *plain values*; their syntax is in Figure 3. Evaluation reduces a closed term to a *cvalue*, or diverges. A cvalue is either a *plain value* v (an abstraction or saturated constructor application), or it is a value wrapped in a single cast, thus $v \blacktriangleright \gamma$ (Figure 3). The latter form is needed because we cannot reduce a term to a plain value without losing type preservation; for example, we cannot reduce $(\mathit{True} \blacktriangleright g)$, where $g : \mathit{Bool} \doteq S$ any further without changing its type from S to $Bool$.

However, there are two situations when a cvalue will not do, namely as the function part of a function application, and as the scrutinee of a **case** expression. Rules (PushL) and (PushC) deal with those two situations, by pushing the coercion inside the term, turning the cast into a plain value. Notice that both rules leave the *context* (the application or case expression) unchanged; they rewrite the function or case scrutinee respectively. Nevertheless, the context is necessary to guarantee that the type of the rewritten term is a function or data type respectively.

The (PushL) rule is quite straightforward, but (PushC) is more complicated. Here is an example, stripped of the **case** context, supposing that $Cons : \forall a.a \to [a] \to [a]$, and $\gamma : [\mathit{Int}] \doteq [S\ \mathit{Bool}]$:

$$\begin{array}{l} \mathit{Cons}\ \mathit{Int}\ e_1\ e_2 \blacktriangleright \gamma \\ \longrightarrow \mathit{Cons}\ (S\ \mathit{Bool}) \ \ (e_1 \blacktriangleright \mathrm{right}\ \gamma) \\ \qquad\qquad\qquad\qquad\ (e_2 \blacktriangleright (\mathrm{refl}\ \mathit{List}) \odot (\mathrm{right}\ \gamma)) \end{array}$$

Here the coercion wrapped around the application of $Cons$ has now been pushed inside to wrap each of its components. (Of course, an implementation does none of this, because types and coercions are erased.)

The reader may like to check that if the left-hand side of each rule is well-typed (in the top-level context) then so is the right-hand side.

Notice that evaluation affects *expressions* only, not types. Since coercions are types, it follows that coercions are not evaluated either.

THEOREM 1 (Progress). *Suppose that* Γ *is consistent, and* $\Gamma \vdash_e e : \sigma$*. Then either* e *is a cvalue, or either* $e \longrightarrow e'$ *and* $\Gamma \vdash_e e' : \sigma$ *for some term* e'*.*

COROLLARY 1 (Syntactic Soundness). *Let* Γ *be consistent and* $\Gamma \vdash_e e : \sigma$*. Then either either* $e \longrightarrow^* cv$ *and* $\Gamma \vdash_e cv : \sigma$ *for some cvalue* cv*, or the evaluation diverges.*

PROOF. By structural induction on e. The interesting case is for application. Suppose $\Gamma \vdash_e e_1\ e_2 : t$. Then $\Gamma \vdash_e e_1 : t_2 \to t$ and $\Gamma \vdash_e e_2 : t_2$. Then there are three well-typed possibilities for e:

1. e_1 is not a cvalue. Then by the induction hypothesis, e_1 can take a (type-preserving) step.
2. e_1 is is a plain value which, to be well typed, must be of form $\lambda x.e_3$. Hence we can take a (Beta) step.
3. e_1 is $v \blacktriangleright \gamma$. Then, since coercions equate only monotypes, v's type must be a monotype; and since v is a value, it must therefore be of form $\lambda x.e_3$ or $K\ \overline{e}$; and hence v's type must be of form $T\ \overline{\tau}$, for some value type constructor T. Now, since e_1 has a function type, γ meets the precondition in the definition of consistency for Γ, and hence v must also have a function type. Hence v must be of form $\lambda x.e_3$, so we can take a type-preserving step using (PushL).

□

We give a call-by-name semantics here, but a call-by-value semantics would be equally easy: simply add extend the syntax of evaluation contexts with the form $v\ E$, and restrict the argument of rule (Beta) to be a cvalue.

3.6 Type Erasure

System $\mathrm{F_C}$ permits syntactic type erasure much as plain System $F\omega$ does. Since coercions are types, standard erasure captures them. Following Pierce [22, Section 23.7] we erase a type abstraction to a trivial term abstraction, and type application to term application to the unit value; this standard device preserves termination behaviour in the presence of `seq`, or with call-by-value semantics.

[2] In the definition of consistency, the equality $T_1 = T_2$ is true mathematical equality, and not an equality kind $T_1 \doteq T_2$!

Values:

$$\begin{array}{lcl} \text{Plain values} \quad v & ::= & \Lambda a.e \mid \lambda x.e \mid K\ \overline{e} \\ \text{Cvalues} \quad cv & ::= & v \blacktriangleright \gamma \mid v \end{array}$$

Evaluation contexts:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \quad ::= \quad [] \mid E\ e \mid E\ \tau \mid E \blacktriangleright \gamma$$

Expression reductions:

$$\begin{array}{lll} (TBeta) & (\Lambda a.e)\ e' & \longrightarrow \quad e[e'/a] \\ (Beta) & (\lambda x.e)\ e' & \longrightarrow \quad e[e'/x] \\ (Case) & \textbf{case}\ (K\ \overline{\phi}\ \overline{\sigma}\ \overline{e})\ \textbf{of}\ \ldots\ K\ \overline{b}\ \overline{x} \to e'\ \ldots & \longrightarrow \quad e'[\overline{\sigma/b}, \overline{e/x}] \\ (Comb) & (v \blacktriangleright \gamma_1) \blacktriangleright \gamma_2 & \longrightarrow \quad v \blacktriangleright (\gamma_1 \rhd \gamma_2) \end{array}$$

$$\begin{array}{lll} (PushL) & (\lambda x.e \blacktriangleright \gamma)\ e' & \longrightarrow \quad (\lambda y.(e[(y \blacktriangleright \gamma_1)/x] \blacktriangleright \gamma_2))\ e' \\ & & \text{where} \ \ \gamma_1 = \text{sym}\ (\text{right}\ (\text{left}\ \gamma)) \\ & & \qquad\quad \gamma_2 = \text{right}\ \gamma \end{array}$$

$$\begin{array}{lll} (PushC) & \textbf{case}\ (K\ \overline{\tau}\ \overline{\varepsilon}\ \overline{e} \blacktriangleright \gamma)\ \textbf{of}\ \overline{p \to rhs} & \longrightarrow \quad \textbf{case}\ (K\ \overline{v}\ \overline{\varepsilon'}\ \overline{e'})\ \textbf{of}\ \overline{p \to rhs} \\ & & \text{where} \ \ \gamma : T\ \overline{\tau} \doteq T\ \overline{v} \\ & & \qquad\quad K : \forall \overline{a : \kappa}. \forall \overline{b : \iota}.\ \overline{\rho} \to T\ \overline{a} \\ & & \qquad\quad \varepsilon_i' = \begin{cases} (\text{sym}\ deep(\tau_1, \overline{a}, \gamma) \rhd \varepsilon_i \rhd deep(\tau_2, \overline{a}, \gamma))[\overline{\varepsilon/b}] \\ \qquad \text{if}\ \ b_i : \tau_1 \doteq \tau_2 \\ \varepsilon_i \quad \text{otherwise} \end{cases} \\ & & \qquad\quad e_i' = e_i \blacktriangleright deep(\rho_i, \overline{a}, \gamma)[\overline{\varepsilon/b}] \end{array}$$

Deep coercions: if $\gamma : T\ \overline{\tau} \doteq T\ \overline{v}$ then $deep(\tau, \overline{a}, \gamma) : \tau[\overline{\tau/a}] \doteq \tau[\overline{v/a}]$

$$\begin{array}{lcll} deep(\tau, \overline{a}, \gamma) & = & \text{refl}\ \tau, & \text{if}\ \overline{a} \notin fv(\tau) \\ deep(b, \overline{a}, \gamma) & = & sel(\overline{a}, b, \gamma), & \text{if}\ b \in \overline{a} \\ deep(\tau_1\ \tau_2, \overline{a}, \gamma) & = & deep(\tau_1, \overline{a}, \gamma) \odot deep(\tau_2, \overline{a}, \gamma) & \\ deep(S_n\ \overline{\tau}, \overline{a}, \gamma) & = & S_n \odot \overline{deep(\tau, \overline{a}, \gamma)} & \end{array} \qquad \begin{array}{lcl} sel((\overline{a}, b), b, \gamma) & = & \text{right}\ (\gamma) \\ sel((\overline{a}, b), c, \gamma) & = & sel(\overline{a}, c, \text{left}\ \gamma) \end{array}$$

Figure 3: Operational semantics

The only difference from plain $F\omega$ is that we also erase casts.

$$\begin{array}{rcl} x^\circ & = & x \\ K^\circ & = & K \\ (\Lambda a\!:\!\kappa.\, e)^\circ & = & \lambda a.e^\circ \\ (e\ \tau)^\circ & = & e^\circ() \end{array} \qquad \begin{array}{rcl} (\lambda x\!:\!\tau.\, e)^\circ & = & \lambda x.e^\circ \\ (e_1\ e_2)^\circ & = & {e_1}^\circ {e_2}^\circ \\ (e \blacktriangleright \gamma)^\circ & = & e^\circ \\ (K\ \overline{a\!:\!\kappa}\ \overline{x\!:\!\tau})^\circ & = & K\ \overline{a}\ \overline{x} \end{array}$$

$$\begin{array}{rcl} (\textbf{let}\ x\!:\!\sigma = e_1\ \textbf{in}\ e_2)^\circ & = & \textbf{let}\ x = {e_1}^\circ\ \textbf{in}\ {e_2}^\circ \\ (\textbf{case}\ e_1\ \textbf{of}\ \overline{p \to e_2})^\circ & = & \textbf{case}\ {e_1}^\circ\ \textbf{of}\ \overline{p^\circ \to {e_2}^\circ} \end{array}$$

THEOREM 2. *Given* $\Gamma \vdash_e e_1 : \sigma$, *(a) either* e_1 *is a cvalue and* ${e_1}^\circ$ *is a value or (b) we have* $e_1 \longrightarrow e_2$ *and either* ${e_1}^\circ \longrightarrow {e_2}^\circ$ *or* ${e_1}^\circ = {e_2}^\circ$.

PROOF. Proof by structural induction on e. The interesting case is application. Suppose we have $\Gamma \vdash_e e_1\ e_2 : \sigma$ and $e_1\ e_2 \longrightarrow e_3$ (in one step). Then either (a) e_1 can take a step (in which case the result follows by induction), or (b) e_1 is a cvalue. The latter has two sub-cases: either (b.1) e_1 is a plain value or (b.2) it is of form $(v_1 \blacktriangleright \gamma)$.

In case (b.1), since e can take a step, e_1 must be of form $\lambda x.e_1'$ so that $e_1\ e_2$ can take a (Beta) step. But then $(e_1\ e_2)^\circ$ can also take a (Beta) step. We need an auxiliary substitution lemma, that ${e_1'}^\circ[{e_2}^\circ/x] = e_1'[e_2/x]^\circ$, and then we are done.

In case (b.2), e_1 is of form $(v_1 \blacktriangleright \gamma)$, and by consistency v_1 must have a function type, and hence must be of the form $\lambda x.e_1'$. Hence $e_1\ e_2$ can take a (PushL) step. Taking a (PushL) step leaves the erasure of the term unchanged, modulo alpha conversion, which gives the result. □

COROLLARY 2. *For an well-typed System* $\mathrm{F_C}$ *term* e_1*, we have* $e_1 \longrightarrow^* e_2$ *iff* ${e_1}^\circ \longrightarrow^* {e_2}^\circ$.

3.7 Guaranteeing consistency

Consistency is both necessary and sufficient for soundness of $\mathrm{F_C}$, but it is not obvious how to check whether (the top level of) a particular program is consistent. For particular classes of programs, however, consistency is easy to guarantee. For example, to translate GADTs into $\mathrm{F_C}$ we need no coercion constants at all; and if there are no coercion constants, then well-typed coercion expressions equate identical types (only):

THEOREM 3 (GADT Consistency). *If* $dom(\Gamma)$ *contains no type variables or coercion constants, and* $\Gamma \vdash_{ty} \gamma : \tau_1 \doteq \tau_2$*, then* $\tau_1 = \tau_2$[3].

The proof is by induction on the structure of γ. Consistency is an immediate corollary of Theorem 3.

We turn our attention to type coercions resulting from AT type functions. For simplicity, we omit higher-kinded types and assume that type functions are unary. Hence, type coercions are of the form $c : \Pi\overline{a : *}.S\ \tau_1 \doteq \tau_2$. These type coercions imply a rewrite system among types. The key to obtain consistency is to guarantee that types have canonical normal forms. Confluence and termination of the rewrite systems are clearly essential properties. Additionally, we demand that type coercions are well-formed. We say that $c : \Pi\overline{a : *}.S\ \tau_1 \doteq \tau_2$ is *well-formed* iff τ_1 does not refer to any abstract type constructor, $fv(\tau_1) = \bar{a}$ and $fv(\tau_2) \subseteq \bar{a}$. This is an entirely

[3] As in the definition of consistency, this $\tau_1 = \tau_2$ is syntactic identity.

natural condition and can also be found in [3]. All three conditions guarantee that each type has a canonical normal form. Thus, we can decide type equality via a canonical normal form test.

Let Γ be a well-formed coercion environment. We define the one-step rewrite derivation from $\tau[S\ \tau_1']$ to $\tau[\phi(\tau_2)]$, written $\tau[S\ \tau_1'] \longrightarrow \tau[\phi(\tau_2)]$, if $(c : \Pi\overline{a : *}.S\ \tau_1 \doteq \tau_2) \in \Gamma$ and there exists a substitution ϕ with domain $\bar{a}$ such that $\phi(\tau_1) = \tau_1'$. We write $\tau[\tau']$ to refer to one occurrence of τ' in τ. We write $\tau_1 \longrightarrow^* \tau_2$ to denote a sequence of rewrite steps where no further steps are applicable on τ_2. We say Γ is *terminating* if for each τ_1 there exists τ_2 such that $\tau_1 \longrightarrow^* \tau_2$. We say Γ is *confluent* if different derivations starting from the same point can always be brought together again.

THEOREM 4 (Canonical Normal Forms). *Let Γ be well-formed, terminating and confluent. Then, $\Gamma \vdash_{ty} \gamma : \tau_1 \doteq \tau_2$ iff $\tau_1 \longrightarrow^* \tau_1$ and $\tau_2 \longrightarrow^* \tau_2$ such that $\tau_1 = \tau_2$.*

The 'only-if' direction follows from soundness of the rewrite steps. The 'if' direction can be proven by induction over the $\vdash_{ty}$ derivation.

From the above result, we immediately obtain consistency.

COROLLARY 3 (AT Consistency). *If Γ is well-formed, terminating and confluent, then Γ is consistent.*

Assume $\Gamma \vdash_{ty} \gamma : T_1\ \overline{\tau_1} \doteq T_2\ \overline{\tau_2}$. Then, we find $T_1\ \overline{\tau_1} \longrightarrow^* \tau_1$ and $T_2\ \overline{\tau_2} \longrightarrow^* \tau_2$ such that $\tau_1 = \tau_2$. None of the rewrite rules affect T_1 or T_2. Hence, τ_1 must have the shape $T_1\ \overline{\tau_1'}$ and τ_2 the shape $T_2\ \overline{\tau_2'}$. Immediately, we find that $T_1 = T_2$ and we are done.

We can state similar results for type coercions resulting from FD programs. Again, the canonical normal form property is the key to obtain consistency. While sufficient the canonical normal form property is not a necessary condition. Consider the non-confluent but consistent environment $\Gamma = \{c_1 : S_1\ [Int] \doteq S_2, c_2 : \Pi a : *.S_1\ [a] \doteq [S_1\ a]\}$. We find that $\Gamma \vdash_{ty} \gamma : S_1\ [Int] \doteq S_2$. But there exist derivations $S_1\ [Int] \longrightarrow^* [S_1\ Int]$ and $S_2 \longrightarrow^* S_2$ where $[S_1\ Int] \neq S_2$. Similar observations can be made for ill-formed, consistent environments.

3.8 Summary

With the technical presentation of F_C complete, we now move on to give some applications. Before doing so, a few points are worth noting.

First, it is well known that GADTs can be used to encode equality evidence [26, Section 9], via a GADT of the form

```
data Eq a b where { EQ :: Eq a a }
```

F_C turns this idea on its head, instead using equality evidence to encode GADTs. This is good for several reasons. First, F_C is simpler than System F plus GADTs. Second, F_C expresses equality evidence in *types*, which permit erasure; GADTs encode equality evidence as *values*, and these values cannot be erased. Why not? Because in the presence of recursion, the mere existence of an expression of type `Eq a b` is not enough to guarantee that `a` is the same as `b`, because bottom certainly has this type. In contrast, our language of types lacks recursion.

Second, to keep the presentation as simple as possible, we restricted F_C so that only let-bound variables can be polymorphic, and even they must have rank-1 types. We believe that there is no technical obstacle to allowing polytypes everywhere, giving a fully impredicative and higher-rank system. Doing so involves adding another couple of coercion forms, similar to composition and decomposition, to move in and out of quantified types. We refrain from doing so here only to focus attention on our main payload.

Lastly, the dynamic semantics of Figure 3 makes all the coercions in the program bigger and bigger. This is not a run-time concern, since types are erased by then, but it might be a concern for compiler transformations. Fortunately there are many type-preserving simplifications that can be performed on coercions, such as:

$$\begin{array}{lcl} (\text{refl } \tau) \odot (\text{refl } \upsilon) & = & \text{refl } (\tau\ \upsilon) \\ \text{sym } (\text{refl } \tau) & = & \text{refl } \tau \\ e \blacktriangleright \text{refl } \tau & = & e \end{array}$$

and so on. We do not give the complete set here, but they are easy to write down, and may be freely used by a compiler to keep the size of coercions under control.

In this context, it is interesting to note the connection of type-equality coercions to the notion of proof objects in machine-supported theorem proving. Coercion terms are essentially proof objects of equational logic and the above simplification rules, as well the manipulations performed by rules, such as (PushL), correspond to proof transformations.

4. Translating GADTs

With F_C in hand, we now sketch the translation of a source language supporting GADTs into F_C. As highlighted in Section 2.1, the key idea is to turn type equalities into coercion types. This approach strongly resembles the dictionary-passing translation scheme known from translating type classes [11]. The difference here is that type equalities are not turned into values, rather, we turn them into types.

We do not have space to present a full source language supporting GADTs, but instead sketch its main features; other papers give full details [30, 8]. We assume that the GADT source language has the following syntax of types:

$$\begin{array}{llcl} \text{Polytypes} & \pi & \to & \eta \mid \forall a.\pi \\ \text{Constrained types} & \eta & \to & t \mid t \doteq t \Rightarrow \eta \\ \text{Monotypes} & \tau & \to & a \mid \tau \to \tau \mid T\ \overline{\tau} \end{array}$$

We deliberately re-use F_C's syntax $\tau_1 \doteq \tau_2$ to describe GADT type equalities. These equality constraints are used in the source-language type of data constructors. For example, the `Succ` constructor from Section 2.1 would have type

$$\texttt{Succ} : \forall a.(a \doteq \texttt{Int}) \Rightarrow \texttt{Int} \to \texttt{Exp}\ a$$

Notice that this already *is* a F_C type, and so requires mere syntactic translation.

To keep the presentation simple, we use a non-syntax-directed translation scheme making use of judgement

$$C; \Gamma \vdash_{GADT} e : \pi \rightsquigarrow e'$$

This can be read "assuming constraint C and type environment Γ, the source-language expression e has type π, and translates to the F_C expression e'". It is straightforward make the translation scheme syntax-directed, e.g. along the lines of the MLGX system described in [24]. The constraint C consists of a set of named type equalities:

$$C \quad ::= \quad \cdot \mid C, c{:}\tau_1 \doteq \tau_2$$

The most interesting translation rules are shown in Figure 4. For simplicity, we assume that all quantified GADT variables are of kind $*$. Rule (Var) deals with variable look up.

The task of introducing and eliminating polymorphic and constrained types is divided among four rules. Rules ($\forall$-Intro) and ($\forall$-Elim) we know from translating Hindley/Milner to System F [13]. Rules (C-Intro) and (C-Elim) deal with introduction and elimination of constrained types, and closely correspond to the class constraint introduction and elimination rules of the standard type-class translation. However, where class constraints induce

$$\boxed{C; \Gamma \vdash_{GADT} e : \pi \leadsto e'}$$

$$\text{(Var)} \quad \frac{(x : \pi) \in \Gamma}{C; \Gamma \vdash_{GADT} x : \pi \leadsto x} \qquad \text{(Eq)} \quad \frac{C; \Gamma \vdash_{GADT} e : \tau \leadsto e' \quad C \vdash_{ty} \gamma : \tau \doteq \tau'}{C; \Gamma \vdash_{GADT} e : \tau' \leadsto e' \blacktriangleright \gamma}$$

$$\text{(}\forall\text{-Intro)} \quad \frac{C; \Gamma \vdash_{GADT} e : \pi \leadsto e' \quad a \notin fv(C, \Gamma)}{C; \Gamma \vdash_{GADT} e : \forall a.\pi \leadsto \Lambda a{:}{*}.\, e'} \qquad \text{(C-Intro)} \quad \frac{C, c{:}\tau_1 \doteq \tau_2;\ \Gamma \vdash_{GADT} e : \eta \leadsto e'}{C; \Gamma \vdash_{GADT} e : \tau_1 \doteq \tau_2 \Rightarrow \eta \leadsto \Lambda(c{:}\tau_1 \doteq \tau_2).\, e'}$$

$$\text{(}\forall\text{-Elim)} \quad \frac{C; \Gamma \vdash_{GADT} e : \forall a.\pi \leadsto e'}{C; \Gamma \vdash_{GADT} e : [\tau/a]\pi \leadsto e'\ \tau} \qquad \text{(C-Elim)} \quad \frac{C; \Gamma \vdash_{GADT} e : \tau_1 \doteq \tau_2 \Rightarrow \eta \leadsto e' \quad C \vdash_{ty} \gamma : \tau_1 \doteq \tau_2}{C; \Gamma \vdash_{GADT} e : \eta \leadsto e'\ \gamma}$$

$$\boxed{C; \Gamma \vdash_{GADT} p \to e : \pi \to \pi \leadsto p' \to e'}$$

$$\text{(Alt)} \quad \frac{\begin{array}{c} K :: \forall \bar{a}, \bar{b}.\overline{\tau' \doteq \tau''} \Rightarrow \overline{\tau} \to T\ \bar{a} \quad \bar{a} \cap \bar{b} = \emptyset \quad fv(\overline{\tau}, \overline{\tau'}, \overline{\tau''}) = fv(\bar{a}, \bar{b}) \quad \theta = [\overline{v/a}] \\ C, \overline{c{:}\theta(\tau') \doteq \theta(\tau'')};\ \Gamma, \overline{x{:}\theta(\tau)} \vdash_{GADT} e : \tau' \leadsto e' \quad \bar{c} \text{ fresh} \end{array}}{C; \Gamma \vdash_{GADT} K\ \overline{x} \to e : T\ \overline{v} \to \tau' \leadsto K\ (\overline{b{:}*})\ (\overline{c{:}\theta(\tau') \doteq \theta(\tau'')})\ (\overline{x{:}\theta(\tau)}) \to e'}$$

Figure 4: Type-Directed GADT to F_C Translation (interesting cases)

value abstraction and application, equality constraints induce type abstraction and application.

The translation of pattern clauses in rule (Case) should not contain any surprises. We replace each GADT constructor by an appropriate F_C constructor which additionally carries coercion types representing the GADT type equalities. For convenience, we assume that nested (source) patterns are de-sugared into a sequence of shallow patterns.

Rule (Eq) applies the cast construct to coerce types. To do so, we must find a coercion γ that witnesses the equality of the two types, and we simply re-use the F_C judgement $C \vdash_{ty} \gamma : \tau_1 \doteq \tau_2$ from Figure 2. In this context, γ is an "output" of the judgement, a type whose syntactic structure describes the proof of $\tau_1 \doteq \tau_2$. The important point to note is that the F_C statement $C \vdash_{ty} \gamma : \tau_1 \doteq \tau_2$ is equivalent to the (left-out) GADT condition that "C implies $\tau_1 \doteq \tau_2$".

Finding a γ is decidable, using an algorithm inspired by the unification algorithm [16]. The key observation is that the statement "C implies $\tau_1 \doteq \tau_2$" holds if $\phi(\tau_1) = \phi(\tau_2)$ where ϕ is the most general unifier of C. W.l.o.g., we neglect the case that C has no unifier, i.e. C is unsatisfiable. Program parts which make use of unsatisfiable constraints effectively represent dead-code.

Roughly, the type coercion construction procedure goes as follows. Given the assumption set C and our "goal" $\tau_1 \doteq \tau_2$ we perform the following calculations:

Step 1 : We normalise the constraint $C = \overline{c : \tau' \doteq \tau''}$ to the *solved* form $\overline{\gamma : a \doteq v}$ where $a_i < a_{i+1}$ and $fv(\bar{a}) \cap fv(\bar{v}) = \emptyset$ by applying the 'decomposition' rule (DRight) (we neglect higher-kinded types for simplicity), rule (Sym) and rule (Trans). $<$ is some suitable ordering among variables. Notice that we disallow recursive types.

Step 2 : Normalise $c' : \tau_1 \doteq \tau_2$ where c' is fresh to the solved form $\overline{\gamma' : a' \doteq v'}$ where $a'_j < a'_{j+1}$.

Step 3 : Match the resulting equations from Step 2 against equations from Step 1.

Step 4 : We obtain γ by reversing the normalisation steps in Step 2.

Failure in any of the steps implies that $C \vdash_{ty} \gamma : \tau_1 \doteq \tau_2$ does not hold for any γ. A constraint-based formulation of the above algorithm is given in [27].

To illustrate the workings of the algorithm we consider $C = \{c_1 : [a] \doteq [b], c_2 : b = c\}$ and $c_3 : [a] \doteq [c]$. We assume that $a < b < c$.
Step 1: Normalising C yields $\{\text{right } c_1 : a \doteq b, c_2 : b = c\}$ in an intermediate step. We apply rule (Trans) to obtain the solved form $\{(\text{right } c_1) \rhd c_2 : a \doteq c, c_2 : b = c\}$
Step 2: Normalising $c_3 : [a] \doteq [c]$ yields right $c_3 : a \doteq c$.
Step 3: We can match right $c_3 : a \doteq c$ against $(\text{right } c_1) \rhd c_2 : a \doteq c$.
Step 4: Reversing the normalisation steps in Step 2 yields $c_3 = (\text{refl } []) \odot ((\text{right } c_1) \rhd c_2)$. Notice that $\vdash_{ty} \text{refl } [] : [] \doteq []$.

The following result can be straightforwardly proven by induction over the derivation.

LEMMA 1 (Type Preservation). *Let $C; \emptyset \vdash_{GADT} e : t \leadsto e'$. Then, $C \vdash_e e' : t$.*

GADT programs make only use of syntactic (a.k.a. Herbrand) type equality. Results from Section 3.7 then guarantee that all F_C programs arising from the translation are sound by Theorem 3. From Corollary 2, we immediately can conclude that the semantics of GADT programs remains unchanged.

LEMMA 2. *Let $\emptyset; \emptyset \vdash_{GADT} e : t \leadsto e'$. Then, $e' \rightarrowtail^* v$ iff $e^\circ \rightarrowtail^* v$ where v is some base value, e.g. integer constants etc.*

5. Translating Associated Types

In Section 2.2 we claimed that F_C permits a more direct and more general type-preserving translation of associated types than the translation to plain System F described in [4]. In fact, the translation of associated types to F_C is almost embarrassingly simple, especially given the translation of GADTs to F_C from the previous section. In the following, we outline the additions required to the standard translation of type classes to System F [11] to support associated types.

5.1 Translating expressions

The translation of expressions requires three additional rules, which exactly match the Rules (Eq), (C-Intro), and (C-Elim) from Figure 4 of the GADT translation. Rule (Eq) permits casting types including associated types to equal types where associated types have been replaced by their definition. Strictly speaking, the Rules (C-Intro) and (C-Elim) are used in a more general setting during associated type translation than during GADT translation. Firstly, the set C contains not only equalities, but both equality and class constraints. Secondly, in the GADT translation only GADT data constructors carry equality constraints, whereas in the associated type translation, any function can carry equality constraints.

5.2 Translating class predicates

In the standard translation, predicates are translated to dictionaries by a judgement $C \Vdash_D D\, \tau \rightsquigarrow \nu$. In the presence of associated types, we have to handle the case where the type argument to a predicate contains an associated type. For example, given the class

```
class Collects c where
  type Elem c      -- associated type synonym
  empty  :: c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

we might want to define

```
sumColl :: (Collects c, Num (Elem c))
        => c -> Elem c
sumColl c = sum (toList c)
```

which sums the elements of a collection, provided theses elements are members of the `Num` class; i.e., provided we have `Num (Elem c)`. Here we have an associated type as a parameter to a class constraint. Wherever the function $sumColl$ is used, we will have to check the constraint `Num (Elem c)`, which will require a cast of the resulting dictionary if c is instantiated.

We achieve this by adding the following rule:

$$\text{(Subst)}\ \frac{C \Vdash_D D\, \tau_1 \rightsquigarrow w \qquad C \vdash_{ty} \gamma : D\, \tau_1 = D\, \tau_2}{C \Vdash_D D\, \tau_2 \rightsquigarrow w \blacktriangleright \gamma}$$

It permits to replace type class arguments by equal types, where the coercion γ witnessing the equality is used to adapt the type of the dictionary w, which in turn witnesses the type class instance. Interestingly, we need this rule also for the translation as soon as we admit qualified constructor signatures in GADT declarations.

5.3 Translating declarations

Strictly speaking, we also have to extend the translation rules for class and instance definitions, as these can now declare and define associated types. However, the extension is so small that we omitted the formal rules for space reasons. In summary, each declaration of an associated type in a type class turns into the declaration of an abstract type constructor in $\mathrm{F_C}$, and each definition of an associated type in an instance turns into a toplevel **coercion** definition in $\mathrm{F_C}$. We have seen examples of this in Section 2.2.

5.4 Observations

In the translation of associated types, it becomes clear why $\mathrm{F_C}$ includes coercions over type constructors of higher kind, instead of being restricted to coercions over manifest types. Consider the following class of monads with references:

```
class Monad m => RefMonad m where
  type Ref m :: * -> *
  newRef :: a -> m (Ref m a)
  readRef :: Ref m a -> m a
  writeRef :: Ref m a -> a -> m ()
```

This class may be instantiated for the `IO` monad and the `ST` monad. The associated type `Ref` is of higher-kind, which implies that the coercions generated from its definitions will also be higher kinded.

The translation of associated types to plain System F imposes two restrictions on the formation of well-formed programs [3, Section 5.1], namely (1) that equality constraints for an n parameter type class must have type variables as the first n arguments to its associated types and (2) that class method signatures cannot constrain type class parameters. Both constraints can be lifted in the translation to $\mathrm{F_C}$.

5.5 Functional dependencies and closed type functions

Due to space limitations, we will only discuss one important observation about the translation of functions dependencies. The current scheme of implementing multi-parameter type classes with functional dependencies in GHC translates these classes as if without dependencies; i.e., the dependencies are only used during type inference. However, this simple translation cannot deal with the examples from Section 2.3, which require the use of the full range of $\mathrm{F_C}$ facilities, just as the translation of associated types does.

In Section 2.4, we argued that type functions independent of classes can also be translated to $\mathrm{F_C}$. That translation proceeds in exactly the same way as the translation of associated types. In other words, the association of type functions with classes is useful to structure source programs, but doesn't influence the implementation essentially. The same is true for extended algebraic data types [28].

6. Related Work

System F with GADTs. Xi et al.[30] introduced the explicitly typed calculus $\lambda_{2,G\mu}$ together with a translation from an implicitly typed source language supporting GADTs. Their calculus has the typing rules for GADTs built in, just like Pottier & Régis-Gianas's MLGI [24]. This is the approach that GHC initially took. $\mathrm{F_C}$ is the result of a search for an alternative.

Encoding GADTs in plain System F and F_ω. There are several previous works [1, 7, 20, 29, 5, 27] which attempt an encoding of GADTs in plain System F with (boxed) existential types. We believe that these primitive encoding schemes are not practical and often non-trivial to achieve. We discuss this in more detail in Appendix A.

An encoding of a restricted subset of GADT programs in plain System F_ω can be found in [23], but this encoding only works for limited patterns of recursion.

Coercion-based subtyping. Mitchell [19] introduced the idea of inserting coercions during type inference for an ML-like languages. However, Mitchell's coercion are not identities, but perform coercions between different numeric types and so forth. A more recent proposal of the same idea was presented by Kießling and Luo [15]. Subsequently, Mitchell [18] also studied coercions that are operationally identities to model type refinement for type inference in systems that go beyond Hindley/Milner.

Much closer to $\mathrm{F_C}$ is the work by Breazu-Tannen et al. [2] who add a notion of coercions to System F to translate languages featuring inheritance polymorphism. In contrast to $\mathrm{F_C}$, their coercions model a subsumption relationship, not equalities, and their coercions are values, not types. Nevertheless, they introduce coercion combinators, as we do, but they don't consider decomposition, which is crucial to translate GADTs. Moreover, the focus of their paper is the translation of an extended version of Cardelli & Wegner's Fun, and in particular, the coherence properties of that translation.

Dependent types and theorem provers. The ideas from Mitchell's work [19, 18] have also been transferred to dependently typed calculi as they are used in theorem provers; see for example [6]. Generally, our coercion terms are a simple instance of the proof terms of logical frameworks, such as LF [12]. A connection that indicates several directions for extending the presented system.

Shao et al. [25] use equalities for type conversion in a lambda calculus with a type language based on the calculus of inductive constructions to denote certified binaries. Interestingly, already GADT equalities are useful to encode safety properties, as Sheard [26] demonstrated.

Translucency and singleton kinds. In the work on ML-style module systems, type equalities are represented as singleton kinds, which are essential to model translucent signatures [9]. Our use of equalities to represent programs that involve functional dependencies and existential types seems related.

7. Conclusions and further work

We showed that explicit evidence for type equalities is a convenient mechanism for the type-preserving translation of GADTs, associative types, functional dependencies, and closed type functions. To keep the presentation as simple as possible, we have restricted coercion kinds $\tau_1 \doteq \tau_2$ to constrain only monotypes. We believe that there is no technical obstacle to allowing them to equate polytypes, thus $\sigma_1 \doteq \sigma_2$, and such an extension would allow a fully-impredicative version of F_C. An interesting avenue for future work is to find good source language features to expose more of the power of F_C to programmers.

Acknowledgements. We thank Roman Leshchinskiy and Benjamin Pierce for their helpful comments on previous versions of this paper. We are also grateful to Lennart Augustsson for a discussion on encoding associated types in System F. We also thank Andreas Abel for pointing out the connection to [23].

References

[1] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.

[2] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[3] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM Press.

[4] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[5] C. Chen, D. Zhu, and H. Xi. Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proc. of PADL'04*, volume 3057 of *LNCS*, pages 239–254. Springer-Verlag, 2004.

[6] G. Chen. Coercive subtyping for the calculus of constructions. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 150–159, New York, NY, USA, 2003. ACM Press.

[7] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.

[8] J. Cheney and R. Hinze. First-class phantom types. TR 1901, Cornell University, 2003.

[9] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249, New York, NY, USA, 2003. ACM Press.

[10] G. J. Duck, S. P. Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *European Symposium on Programming 2004 (ESOP'04)*, number 2986 in LNCS, pages 49–63. Springer-Verlag, 2004.

[11] C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[12] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87*, pages 194–204. IEEE Computer Society Press, 1987.

[13] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[14] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[15] R. Kießling and Z. Luo. Coercions in hindley-milner systems. In *Types for Proofs and Programs: Third International Workshop, TYPES 2003, Revised Selected Papers*, number 3085 in LNCS, pages 259–275, 2004.

[16] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[17] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[18] J. Mitchell. Polymorphic type inference and containment. In *Logical Foundations of Functional Programming*, pages 153–193. Addison-Wesley, 1990.

[19] J. C. Mitchell. Coercion and type inference. In *Proc of POPL 84*, pages 175–185. ACM-Press, 1984.

[20] E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, September 2004.

[21] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. http://research.microsoft.com/ simonpj/papers/gadt/, Nov. 2005.

[22] B. Pierce. *Types and programming languages*. MIT Press, 2002.

[23] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical report, Carnegie Mellon University, 1989.

[24] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244. ACM Press, 2006.

[25] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

[26] T. Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

[27] M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.

[28] M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

[29] S. Weirich. Type-safe cast (functional pearl). In *Proc. of ICFP'00*, pages 58–67. ACM Press, 2000.

[30] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, 2003. ACM Press.

A. Primitive Translation of GADTs

We attempt a primitive translation (encoding) of GADTs to System F with (boxed) existential types (for convenience we will use Haskell extended with rank-n types and existentials). We provide evidence that such an encoding is sometimes hard to achieve.

The gist of the primitive encoding idea is to model type equality $a \doteq b$ via safe coercion functions. Effectively, a pair of embedding/projection functions. Each type cast $\gamma \blacktriangleright e$ is then turned into the function application $\gamma\ e$. To ensure correctness of this encoding scheme, we need to guarantee that at run-time each coercion γ evaluates to the identity.

There are two approaches known in the literature to encode such coercion functions. One approach, employed in [1, 7, 20, 29], uses "Leibniz" equality

```
newtype EQ a b =
   Proof { apply :: forall f . f a -> f b }
refl :: EQ a a
refl    =  Proof id
newtype Flip f a b =  Flip { unFlip :: f b a }
symm :: EQ a b -> EQ b a
symm p =  unFlip (apply p (Flip refl))
trans :: EQ a b -> EQ b c -> EQ a c
trans p q =  Proof (apply q . apply p)
newtype List f a =  List { unList :: f [a] }
list :: EQ a b  -> EQ [a] [b]
list p =  Proof (unList . apply p . List)
```

We also provide a few sample type coercion functions. As pointed out in [5], the trouble with this approach is that it seems impossible to define "decomposition" functions such as

```
decompList :: EQ [a] [b] -> EQ a b
```

The alternative method is to represent type equality as follows.

```
type EQ a b = (a->b,b->a)
refl :: EQ a a
refl = (id,id)
sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)
trans :: EQ a b -> EQ b c -> EQ a c
trans (f1,g1) (f2,g2) = (f2.f1,g1.g2)
list :: EQ a b -> EQ [a] [b]
list (f,g) = (map f, map g)
```

The advantage is that decomposition is possible for some types but not for all as will see at the end of this section. Though, many (if not all) realistic GADT programs can be translated based on this encoding [27]. On the other hand, the (serious) disadvantage of this representation is that it may incur a severe run-time penalty. Consider the definition of `list` where we have to apply the coercion functions to each element.

Let's attempt an encoding of the trie example found in [8]. A trie is a finite map from keys to values whose structure depends on the type of keys, here encoded as products and sums in GADT variants:

```
data Either a b where
   Left  :: a -> Either a b
   Right :: b -> Either a b
data Trie k v where
  TUnit ::
    Maybe v                 -> Trie () v
  TSum  :: forall k1 k2.
    Trie k1 v -> Trie k2 v -> Trie (Either k1 k2) v
  TProd :: forall k1 k2.
    Trie k1 (Trie k2 v)     -> Trie (k1, k2) v
```

A trie for a unit type is maybe one value, a trie for a sum is a product of tries, and a trie for a product is a composition of tries. An important operation on tries is the merging of two maps with the same domain and co-domain.

```
merge :: (v -> v -> v)
      -> Trie k v -> Trie k v -> Trie k v
merge c (TUnit Nothing ) (TUnit Nothing  ) =
  TUnit Nothing
merge c (TUnit Nothing ) (TUnit (Just v')) =
  TUnit (Just v')
merge c (TUnit (Just v)) (TUnit Nothing  ) =
  TUnit (Just v)
merge c (TUnit (Just v)) (TUnit (Just v')) =
  TUnit (Just (c v v'))
merge c (TSum ta tb)     (TSum ta' tb')    =
  TSum (merge c ta ta') (merge c tb tb')
merge c (TProd ta)       (TProd ta')       =
  TProd (merge (merge c) ta ta')
```

The second two last equations are interesting. The patterns of the first and second argument constrain `k` to `Either k1 k2` and `Either k1' k2'`, respectively. Hence, we have

$$\mathtt{Either\,k1\,k2} = \mathtt{k} = \mathtt{Either\,k1'\,k2'}$$

from which we can follow $\mathtt{k1} = \mathtt{k1}'$ and $\mathtt{k2} = \mathtt{k2}'$. The point is that to translate the above to $\mathrm{F_C}$, we need to construct a coercion that witness these equalities, we need decomposition.

To encode the trie example, we need (among others) a function

```
decomp :: EQ (Either a b) (Either c d) -> EQ a b
```

But it seems impossible to define such a function if we use Leibniz equality.

Let's consider the "other" type equality representation. To ensure correctness of the encoding scheme, we need to maintain the invariant that for any type coercion function `coerce :: EQ a b -> EQ c d` we have that `coerce` applied to a pair of identity functions yields another pair of identity functions. We are more lucky here, a function `decomp:: EQ (Either a b) (Either c d) -> EQ a c` with the above property is actually definable.

For simplicity, we only give parts of the definition of `decomp`.

```
decomp1 :: (Either a b -> Either c d) -> (a->c)
decomp1 f  = \ a -> case (f (Left a)) of
                      Left c -> c
```

We inject the `a` value into the `Either` data type, apply the incoming coercing function and then extract the `c` value. It is easy to verify that the invariant is satisfied.

There are many other examples which can be translated using the "other" type equality representation [27]. In fact, it almost seems that all practical examples can be encoded. Though, not every decomposition function is definable. Here is the (contrived) critical example.

```
data Foo a  where
   K :: Foo a
data Erk a b c where
   I :: c -> Erk a a c
f :: Erk (Foo a) (Foo Int) a -> a
f (I x) = x + 1
```

First, we convince ourselves that the above program is well-typed. The pattern `I x` in combination with the type annotation implies that $\mathtt{Foo\ a} = \mathtt{Foo\ Int}$. By decomposition, we conclude that $\mathtt{a} = \mathtt{Int}$. Thus, the program text `x + 1` can be given type `Int`. Hence, the above is well-typed. To translate the above, we need to define a function of type `EQ (Foo a) (Foo Int) -> EQ a Int`. We claim it is impossible to define such a function with satisfies the invariant. It suffices to show that a function

```
decompFoo :: (Foo a->Foo Int)->(a->Int)
```

with the property that `decompFoo ( x->x)` evaluates to `x->x` is not definable.

The problem here is that a value of type `a` cannot be injected into a value of type `Foo a`. So, clearly the incoming function of type `Foo a->Foo Int` is useless. Effectively, we could omit the function parameter altogether. Parametricity tells us that any function of type `a->Int` must be a constant function. Hence, `decompFoo` applied to any function of type `Foo a->Foo Int` yields a constant function. Hence, an encoding of the above critical example is impossible.

In fact, the "decomposition" problem is hardly surprising given that similar issues arise when translating type class programs [11].

```
class Foo a where foo :: a->Int
instance Foo a => Foo [a] where
    foo [] = 1
    foo _  = 2
bar :: Foo [a] => a->Int
bar = foo
```

Based on the System F-style translation scheme described in [11], we are unable to translate function `bar`. The program text demands a dictionary for `Foo a` but the annotation only supplies a dictionary for `Foo [a]`. This is the wrong way around. The instance declaration tells us how to construct `Foo [a]` given `Foo a` but the other direction does not hold in general.