Embedding F

Sam Lindley
University of Strathclyde
Sam.Lindley@ed.ac.uk

Abstract

This millennium has seen a great deal of research into embedded domain-specific languages. Primarily, such languages are simply-typed. Focusing on System F, we demonstrate how to embed polymorphic domain specific languages in Haskell and OCaml. We exploit recent language extensions including kind polymorphism and first-class modules.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Languages, Theory

Keywords Polymorphism, domain specific languages, higher-order abstract syntax, OCaml, Haskell

1. Introduction

With the advent of GADTs it has become common to embed domain-specific languages in Haskell. HOAS (Higher Order Abstract Syntax) [21] representations are convenient for programming, as hygiene is handled by the host language. However, sometimes first-order representations are more appropriate, for instance, when implementing intensional algorithms, such as compiler optimisations, that inspect the structure of object terms. Previously [3], we showed how to "unembed" simply-typed HOAS terms as first-order well-typed de Bruijn terms [14], allowing DSL programmers to have their cake and eat it.

Others [22, 25] have considered HOAS embeddings of System F [15] in richer languages. In this paper, we focus primarily on the problem of extending the *first-order* representation of well-typed de Bruijn terms with polymorphism. We also show how to extend unembedding with polymorphism.

The main contributions of this paper are threefold:

- Well-typed first-order encodings of System F in F_ω, Haskell, and OCaml, using a de Bruijn representation for System F terms, and host language types as System F types.
- A well-typed first-order encoding of F_ω in Haskell, using a de Bruijn representation for F_ω terms, and Haskell types as F_ω types.
- An extension of previous work on unembedding HOAS representations as first-order representations to support polymorphic languages.

The rest of the paper is structured as follows. Section 2 extends the standard GADT-based de Bruijn encoding of simply-typed lambda calculus to System F. Section 3 shows that essentially the same approach can be adapted to embed System F in F_ω by decomposing the GADTs into existentials and equality proofs. Section 4 re-targets the embedding to OCaml 3.12, making use of first-class recursive modules to encode existentials and equality. Section 5 returns to the Haskell implementation, showing that as of GHC 7.4, it actually provides an embedding of F_ω in Haskell. Section 6 adapts the approach to a HOAS representation and shows that *unembedding* scales to polymorphic object languages. Section 7 discusses related work, and finally Section 8 concludes.

2. System F in Haskell

2.1 Well-typed terms

The first-order de Bruijn encoding of typing judgements for simply-typed lambda calculus in GHC using GADTs is rather direct. First, variable judgements are defined as follows:

```
data Var :: ⋆ → ⋆ → ⋆ where
   Z :: Var (Γ, a) a
   S :: Var Γ a → Var (Γ, b) a
```

The first argument to `Var` is a type environment and the second argument is a type. Thus, `n :: Var Γ a` states that in type environment Γ the variable n has type `a`. The variables themselves are simply unary de Bruijn variables.

Typing judgements for terms are defined as follows:

```
data Exp :: ⋆ → ⋆ → ⋆ where
  Var :: Var Γ a → Exp Γ a
  Lam :: Exp (Γ, a) b → Exp Γ (a → b)
  App :: Exp Γ (a → b) → Exp Γ a → Exp Γ b
```

Again, the first argument to `Exp` is a type environment and the second argument is a type. Thus, `e :: Exp Γ a` states that in type environment Γ, the term `e` has type `a`. Notice, for instance, how the type for `Lam` exactly parallels the introduction rule for functions: if we have a term `e` whose type is `b` in type environment $(\Gamma, \mathtt{a})$, then we can abstract over `a` to introduce a term `Lam e` whose type is $\mathtt{a} \to \mathtt{b}$ in type environment Γ.

If we construct a term using the constructors of `Exp`, then it must, by construction, represent a well-typed term, because we have directly encoded the typing rules in the GADT. The idea of this representation dates back at least to Altenkirch and Reus [1], who applied it in a dependent type theory.

In order to extend well-typed terms to support polymorphism, we must encode type abstractions and applications. Just as lambda abstractions can be encoded using either a de Bruijn representation or using host language lambda abstractions [21], so type abstractions can be encoded using either a de Bruijn representation or host

WGP'12, September 9, 2012, Copenhagen, Denmark.

language polymorphism. In this paper we explore the latter option, aiming to choose an encoding of object types that is as close as possible to the equivalent host language types.

2.2 Polymorphism

Let us try to extend the `Exp` type to include type lambda and type application expressions. This amounts to encoding the introduction and elimination rules for universal quantification. The introduction rule is:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash (\Lambda\alpha.M) : \forall\alpha.A}\alpha \notin FTV(\Gamma)$$

Note that the type A may contain the type variable α. We can model this property by representing A as a type constructor applied to a Haskell type variable. But we need to be careful to appropriately quantify over the Haskell type variable:

```
TLam :: ( ∀a.Exp Γ (k a)) → Exp Γ ( ∀a.k a)
```

For the premise we quantify over `a` outside the `Exp` type constructor; for the conclusion, we quantify over `a` inside the `Exp` type constructor. This is all well and good, but unfortunately Haskell does not play well with GADTs indexed by polymorphic types. It is possible to work around this problem by hand-coding GADTs using Leibniz equality and enabling the `ImpredicativeTypes` language extension. However, doing so requires some effort, and type inference in this setting is rather fragile. A more robust work-around is to define a `newtype` for boxing the polymorphism, which eases the job that type inference must perform:

```
newtype Poly k = Poly {unPoly ::  ∀a.k a}
```

Now, we have:

```
TLam :: ( ∀a.Exp Γ (k a)) → Exp Γ (Poly k)
```

The elimination rule for polymorphism is:

$$\frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash M\ B : A[B/\alpha]}$$

We again use the `Poly` constructor for the polymorphic type in the premise, while the substitution in the conclusion is implemented simply by applying the type constructor:

```
TApp :: Exp Γ (Poly k) → Exp Γ (k a)
```

The data type for expressions becomes:

```
data Exp :: ⋆ → ⋆ → ⋆ where
  Var  :: Var Γ a → Exp Γ a
  Lam  :: Exp (Γ, a) b → Exp Γ (a → b)
  App  :: Exp Γ (a → b) → Exp Γ a → Exp Γ b
  TLam :: ( ∀a.Exp Γ (k a)) → Exp Γ (Poly k)
  TApp :: Exp Γ (Poly k) → Exp Γ (k a)
```

GHC is happy to accept this, but we still have a problem. Look carefully at the types, and it becomes clear that it is impossible to actually construct any interesting `TLam` expressions because the only way of constructing an expression of type `k a` is as a variable (which admits any type at all). The only other possibility would be a type application, but that requires us to have already constructed a polymorphic expression using `TLam`.

The problem is that Haskell type constructors are second-class citizens. We must define them as `newtypes`, and then we need to explicitly box and unbox values to and from the `newtype`. For instance, suppose we wish to define the polymorphic identity function. We first define a type constructor of the appropriate type:

```
newtype AtoA a = AtoA { unAtoA :: a → a }
```

The identity function `Lam (Var Z)` has type `Exp (a -> a)`, but we need to inject it into the type `forall a.Exp (AtoA a)` before passing it as an argument to `TLam`. Correspondingly, after instantiating the polymorphic identity function at a particular type, we must project from the boxed type before we can apply it to an argument.

2.3 Deriving isomorphisms

In order to coordinate injection and projection we define a type class representing the isomorphism between `newtypes` and their representations.

```
class Iso a b where
  inject :: a → b
  project :: b → a

instance Iso a a where
  inject x = x
  project x = x
```

The special `newtype deriving` mechanism of GHC allows suitable instances to be derived semi-automatically from the canonical instance for `Iso a a`.

Let us add a deriving clause to `AtoA`:

```
newtype AtoA a = AtoA { unAtoA :: a → a }
  deriving (Iso (a → a))
```

This clause automatically derives the functions:

```
inject :: (a → a) → AtoA a
inject x = AtoA (inject x)
project :: AtoA → (a → a)
project x = unAtoA (project x)
```

As `newtypes` are unboxed in the backend, both `inject` and `project` are actually realised as the identity function.

Now we add constructors for injection and projection to our expression type:

```
data Exp :: ⋆ → ⋆ → ⋆ where
  Var     :: Var Γ a → Exp Γ a
  Lam     :: Exp (Γ, a) b → Exp Γ (a → b)
  App     :: Exp Γ (a → b) → Exp Γ a → Exp Γ b
  TLam    :: ( ∀a.Exp Γ (k a)) → Exp Γ (Poly k)
  TApp    :: Exp Γ (Poly k) → Exp Γ (k a)
  Inject  :: Iso a b ⇒ Exp Γ a → Exp Γ b
  Project :: Iso a b ⇒ Exp Γ b → Exp Γ a
```

Finally, we add wrappers for `Inject` and `Project`.

```
inj :: Iso a b ⇒ (a → b) → Exp Γ a → Exp Γ b
inj _ = Inject

proj :: Iso a b ⇒ (a → b) → Exp Γ b → Exp Γ a
proj _ = Project
```

The first argument for each is a proxy used to aid type inference. In both cases it will always be a `newtype` constructor.

2.4 Examples

We define the polymorphic identity function as follows:

```
id_poly :: Exp () (Poly AtoA)
id_poly = TLam (inj AtoA (Lam (Var Z)))
```

and instantiate it at a concrete type:

```
id_int :: Exp () (Int → Int)
id_int = proj AtoA (TApp id_poly)
```

We can now use polymorphism wherever we like. We just need to define an appropriate `newtype` including a `deriving (Iso ...)` clause, inject into the `newtype` before using `TLam`, and project from it after using `TApp`.

It is tempting to try to define a function `tlam = TLam . inj`. Unfortunately, GHC cannot infer a type for such a function, and it does not appear possible to explicitly assign it a type either. The same problem occurs when trying to combine `TApp` and `proj`.

In use, the embedded object language we have developed is not quite the same as System F, but it is equivalent — our language differs only in the way that types are abstracted and applied in terms. We do not need to annotate lambdas with their inputs as GHC infers them for us. For type abstractions, rather than providing a type variable, we provide a description of the polymorphic type in the form of a type constructor into which we inject the body of the type abstraction. For type applications, instead of providing a type, we again provide the type constructor, from which we project a concrete instantiation of a polymorphic term. The resulting presentation of System F is similar to Russo and Vytniotis's QML language [27], which extends ML with System F polymorphism.

The need to define a new type constructor for each polymorphic type is inconvenient, but it seems unavoidable given that type constructors are not first-class in Haskell.

2.5 Nested quantifiers

In Haskell, `newtypes` can only be defined at the top-level. This means that if a type variable from an outer quantifier appears inside a polymorphic type then we need to define a multi-parameter type constructor. For instance, here is how we define the K combinator, which takes two arguments, ignores the first and returns the second:

```
newtype KC' a b = KC' { unKC' :: a → b → a }
  deriving (Iso (a → b → a))
newtype KC a = KC { unKC :: Poly (KC' a) }
  deriving (Iso (Poly (KC' a)))

kc :: Exp () (Poly KC)
kc = TLam (inj KC
       (TLam (inj KC' (Lam (Lam (Var (S Z)))))))
```

2.6 An evaluator

In order to test that our implementation is at least somewhat sensible, we provide an implementation of an evaluator.

```
data Env :: ⋆ → ⋆ where
  Empty :: Env ()
  Extend :: Env Γ → a → Env (Γ, a)

lookupVar :: Env Γ → Var Γ a → a
lookupVar (Extend _ v)   Z     = v
lookupVar (Extend env _) (S n) = lookupVar env n

eval :: Env Γ → Exp Γ a → a
eval env (Var x)     = lookupVar env x
eval env (Lam e)     = λv → (eval (Extend env v) e)
eval env (App f a)   = eval env f $ eval env a
eval env (TLam e)    = Poly (eval env e)
eval env (TApp e)    = unPoly (eval env e)
eval env (Inject e)  = inject (eval env e)
eval env (Project e) = project (eval env e)
```

The cases for variables, lambdas and applications are standard. The other cases just deal with the necessary boxing and unboxing of `newtypes`.

As a trivial example, let us instantiate the identity function at type `Int`:

```
id_int :: Exp () (Int → Int)
id_int = proj AtoA (TApp id_poly)
```

If we now do `eval Empty id_int 42`, then GHC returns `42` as expected.

$$
\begin{array}{ll}
\text{(kinds)} & K ::= \star \mid K \to K \\
\text{(types)} & A, B ::= \alpha \mid A \to B \mid \forall \alpha^K.A \mid \lambda \alpha^K.A \mid A\ B \\
\text{(terms)} & e, f ::= x \mid \lambda x^A.e \mid e\ f \mid \Lambda \alpha^K.e \mid e\ A \\
\text{(environments)} & \Gamma, \Delta ::= \cdot \mid \Gamma, \alpha : K \mid \Gamma, x : A
\end{array}
$$

Figure 1. F_ω syntax

3. System F in F_ω

Having developed an embedding of System F in Haskell, we now show that the same idea can be adapted to embed System F in F_ω by translating away the GADTs into existential types and type-level equality coercions. Though F_ω does not have GADTs, it does have first-class type constructors in the form of lambda abstractions at the level of types, which do make life considerably easier. In the next Section we use the embedding in F_ω as inspiration for an embedding in OCaml using recursive first-class modules.

The syntax of F_ω is given in Figure 1. The typing rules and reduction rules for System F and F_ω can be found in standard textbooks (e.g. [15, 24]).

3.1 Syntactic sugar

We assume standard encodings of n-ary lambdas ($A_1 \to \cdots \to A_n \to B$), products ($A_1 \times \cdots \times A_n$), sums ($A_1 + \cdots + A_n$), and universal quantifiers ($\forall \alpha_1^{K_1} \ldots \alpha_n^{K_n}.A$). In addition, we assume a standard encoding of n-ary existential quantifiers ($\exists \alpha_1^{K_1} \ldots \alpha_n^{K_n}.A$) in terms of universal quantifiers, and the following sugar for using n-ary existentials.

$$
\begin{array}{l}
\exists.A \equiv A \\
\exists \alpha_1 \ldots \alpha_n.A \equiv \exists \alpha_1.\exists \alpha_2 \ldots \alpha_n.A
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pack}\ []e\ \mathsf{as}\ B \equiv e \\
\mathsf{pack}\ [A_1 \ldots A_n]e\ \mathsf{as}\ B \equiv \\
\quad \mathsf{pack}\ [A_1](\mathsf{pack}\ [A_2 \ldots A_n]e\ \mathsf{as}\ B')\ \mathsf{as}\ B \\
\qquad \text{where } B \longrightarrow_* \exists \alpha_1.B' \text{ and } B' = \exists \alpha_2\ \ldots\ \alpha_n.B''
\end{array}
$$

$$
\begin{array}{l}
\mathsf{unpack}\ e\ \mathsf{as}\ []x\ \mathsf{in}\ e' \equiv e'[e/x] \\
\mathsf{unpack}\ e\ \mathsf{as}\ [\alpha_1, \ldots, \alpha_m]x\ \mathsf{in}\ e' \equiv \\
\quad \mathsf{unpack}\ e\ \mathsf{as}\ [\alpha_1]z_1\ \mathsf{in}\ \mathsf{unpack}\ z_1\ \mathsf{as}\ [\alpha_2, \ldots, \alpha_m]z_m\ \mathsf{in}\ [z_m/x] \\
\qquad \text{where } z_1, \ldots, z_m \text{ are not free in } e'
\end{array}
$$

$$
\begin{array}{l}
\mathsf{inj}_k\ [\alpha_1\ \ldots\ \alpha_m](x_1, \ldots, x_n) \Rightarrow e \equiv \\
\quad \mathsf{inj}_k\ y\ \Rightarrow\ \mathsf{unpack}\ y\ \mathsf{as}\ [\alpha_1, \ldots, \alpha_m]z\ \mathsf{in} \\
\qquad\qquad e[\mathsf{proj}_1\ z/x_1, \ldots, \mathsf{proj}_m\ z/x_n] \\
\qquad \text{where } x \text{ and } y \text{ are not free in } e
\end{array}
$$

3.2 Leibniz equality

In order to simulate GADTs in F_ω, we must encode type-level equality constraints. Leibniz equality is expressed in F_ω as the type $\forall \alpha\ \beta\ \phi^{\star \to \star \to \star}.\phi\ \alpha \to \phi\ \beta$ (see, e.g., [4]). The intuition is that if the types bound to α and β are equal then they should be equal in any context, where a context is exactly a type constructor $\phi^{\star \to \star \to \star}$.

Figure 2 defines Leibniz equality. We write $Eq\ A\ B$ for the type of proofs that type A and type B are equal. We write $refl\ A$ (reflexivity) for the proof that type A is equal to itself, and $cast\ A\ B\ p$ for the function that uses equality proof p of type $Eq\ A\ B$ to convert any value of type A to the same value at type B. (It is also straightforward to define functions for symmetry and transitivity, but they are not necessary here.)

3.3 The embedding

To simplify the presentation, we will assume that our version of F_ω has been extended with recursive types, and define our type

$$
\begin{array}{l}
Eq :: \star \to \star \to \star \\
Eq = \lambda\alpha\,\beta.\forall\phi^{\star\to\star}.\phi\,\alpha \to \phi\,\beta \\
\\
refl\ :\ \forall\alpha.Eq\,\alpha\,\alpha \\
refl = \Lambda\alpha\,\phi^{\star\to\star}.\lambda x^{\phi\,\alpha}.x \\
\\
cast\ :\ \forall\alpha\,\beta.Eq\,\alpha\,\beta \to \alpha \to \beta \\
cast = \Lambda\alpha\,\beta.\lambda p^{Eq\,\alpha\,\beta}\,x^{\alpha}.p\,(\lambda\gamma.\gamma)\,x
\end{array}
$$

Figure 2. Type equality in F_ω

$$
\begin{array}{l}
Z, S :: \star \to \star \to \star \\
\quad Z = \lambda\Gamma\,\alpha.\ \exists\Delta.\quad Eq\,\Gamma\,(\Delta \times \alpha) \\
\quad S = \lambda\Gamma\,\alpha.\ \exists\Delta\,\beta.\ Eq\,\Gamma\,(\Delta \times \beta) \times Var\,\Delta\,\alpha \\
\\
Var, Lam, App, TLam, TApp :: \star \to \star \to \star \\
\quad Var = \lambda\Gamma\,\alpha.\ Z\,\Gamma\,\alpha + S\,\Gamma\,\alpha \\
\quad Lam = \lambda\Gamma\,\alpha.\ \exists\beta\,\gamma.\qquad Eq\,(\beta \to \gamma)\,\alpha \times Exp\,(\Gamma \times \beta)\,\gamma \\
\quad App = \lambda\Gamma\,\alpha.\ \exists\beta.\qquad\ Exp\,\Gamma\,(\beta \to \alpha) \times Exp\,\Gamma\,\beta \\
TLam = \lambda\Gamma\,\alpha.\ \exists\phi^{\star\to\star}.\quad Eq\,(\forall\beta.\phi\,\beta)\,\alpha \times (\forall\beta.Exp\,\Gamma\,(\phi\,\beta)) \\
TApp = \lambda\Gamma\,\alpha.\ \exists\beta\,\phi^{\star\to\star}.\ Eq\,(\phi\,\beta)\,\alpha \times Exp\,\Gamma\,(\forall\gamma.\phi\,\gamma) \\
\\
Exp :: \star \to \star \to \star \\
Exp = \lambda\Gamma\,\alpha.\ Var\,\Gamma\,\alpha + \\
\qquad\qquad Lam\,\Gamma\,\alpha + App\,\Gamma\,\alpha + \\
\qquad\qquad TLam\,\Gamma\,\alpha + TApp\,\Gamma\,\alpha
\end{array}
$$

Figure 3. Embedding System F in F_ω

constructors mutually recursively using a Scott encoding [18]. The embedding is given in Figure 3.

For variables we have type constructors Z and S. For $Z\,\Gamma\,\alpha$, we just need to provide a proof that the first element of Γ is α, that is, a proof that there exists an environment Δ, such that Γ is equal to $(\Delta \times \alpha)$. For $S\,\Gamma\,\alpha$, we must provide a proof that Γ is non-empty, along with a variable that can be typed by the tail of Γ. These type constructors are really just the same as those in the Haskell implementation. The key difference is that the implicit equality proofs provided by the GADT representation have had to be made explicit.

The type constructors for expressions follow a similar pattern. The Var constructor just dispatches to S and Z. The Lam constructor existentially quantifies over the argument and result types and includes a proof that they combine to form the type of the resulting expression. The App constructor existentially quantifies over the function argument type. Both $TLam$ and $TApp$ existentially quantify over the type constructor ϕ that is used to encode a universal quantifier. The $TApp$ constructor additionally existentially quantifies over the type argument. Apart from the existential quantification and equality proofs, the embedding is essentially the same as the Haskell implementation, and can equally well be read as a fairly direct transcription of the typing rules for System F.

The Exp constructor gathers together all of the expression constructors into a single sum type.

3.4 An evaluator

Assuming we have recursive functions (which we can define anyway once we have recursive types), we can write an evaluator for our System F embedding in F_ω, which is again similar to the Haskell implementation. The evaluator is given in Figure 4. We omit the evident type arguments for readability. A key difference is that the environment is just Γ rather than the encoding of a GADT

$$
\begin{array}{l}
lookup\ :\ \forall\Gamma\,\alpha.\Gamma \to Var\,\Gamma\,\alpha \to \alpha \\
lookup = \Lambda\Gamma\,\alpha.\lambda env^{\Gamma}\,n^{Var\,\Gamma\,\alpha}. \\
\quad \mathsf{case}\ n\ \mathsf{of}\ \mathsf{inj}_1\,[\Delta](p) \qquad \Rightarrow \mathsf{proj}_2\,(cast\,p\,env) \\
\qquad\qquad \mathsf{inj}_2\,[\Delta\,\beta](p,n) \Rightarrow lookup\,(\mathsf{proj}_1\,(cast\,p\,env))\,n \\
\\
\quad eval\ :\ \forall\Gamma\,\alpha.\Gamma \to Exp\,\Gamma\,\alpha \to \alpha \\
\quad eval = \Lambda\Gamma\,\alpha.\lambda env^{\Gamma}\,e^{Exp\,\Gamma\,\alpha}. \\
\quad \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1\,n \qquad\qquad\quad \Rightarrow lookup\ env\ n \\
\qquad\qquad \mathsf{inj}_2\,[\beta\,\gamma](p,b) \quad \Rightarrow cast\,p\,(\lambda v^{\beta}.eval\,(env,v)\,b) \\
\qquad\qquad \mathsf{inj}_3\,[\beta](f,a) \quad\ \Rightarrow (eval\ env\ f)\ (eval\ env\ a) \\
\qquad\qquad \mathsf{inj}_4\,[\phi^{\star\to\star}](p,b) \Rightarrow cast\,p\,(\Lambda\beta.eval\ env\ (b\,\beta)) \\
\qquad\qquad \mathsf{inj}_5\,[\beta\,\phi^{\star\to\star}](p,e) \Rightarrow cast\,p\,((eval\ env\ e)\,\beta)
\end{array}
$$

Figure 4. An evaluator for System F in F_ω

$$
\begin{array}{ll}
\text{(types)} & A, B ::= \alpha \mid A \to B \mid \forall\alpha.A \\
\text{(variables)} & n ::= \mathrm{z} \mid \mathrm{s}\ n \\
\text{(terms)} & M, N ::= n \mid \lambda^{A}.M \mid M\ N \mid \Lambda\alpha.M \mid M\ A \\
\text{(environments)} & \Gamma, \Delta ::= \cdot \mid \Gamma, A
\end{array}
$$

Figure 5. System F de Bruijn syntax

representing well-formed environments. This works for our use-case, and indeed we could have done the same for the Haskell implementation. It does not, however, seem possible to realise a faithful representation of the `Env` GADT from the Haskell implementation in the F_ω version. This is due to a well-known limitation of the Leibniz encoding of GADTs, which is unable to capture the injectivity of GADTs [11]. (Essentially the problem is that the type system is unable to capture the property that $\phi\,A = \phi\,B$ implies that $A = B$.)

The other main difference from the Haskell evaluator is that we must explicitly apply the equality proofs using the $cast$ function.

3.5 Soundness of the embedding

For System F we use a de Bruijn representation, whereas for F_ω we use variable names. The syntax of our de Bruijn representation of System F is given in Figure 5.

We now define a function for embedding System F typing judgements in F_ω, and give a soundness result. The function $[\![-]\!]$ maps System F typing judgements ($\Gamma \vdash_F M : A$) to F_ω typing judgements ($\Gamma \vdash_{F_\omega} e : A$). (Note that $[\![-]\!]$ is a meta-level function defined outside of either language.) The auxiliary System F judgement $\Gamma \ni_F n : A$ states that in type environment Γ de Bruijn variable n has type A.

Informally, the soundness result we seek should state that if we evaluate an embedded System F term, then the result is $\beta\eta$-convertible to the lifting of the System F term to the equivalent F_ω term. Thus, as well as the embedding function $[\![-]\!]$, we also need to define a function $\lceil - \rceil$ to lift a de Bruijn representation of a System F term to a standard Barendregt-style representation of an F_ω term. The embedding and lifting functions are defined in Figure 6.

We write $\Gamma \vdash_{F_\omega} e = f : A$ if $\Gamma \vdash_{F_\omega} e : A$, $\Gamma \vdash_{F_\omega} f : A$ and e is $\beta\eta$-convertible to f.

THEOREM 1 (Soundness).

1. *If* $\Delta \vdash_{F_\omega} env : \Gamma$ *and* $\Gamma \ni_F n : A$, *then*
$$\Delta \vdash_{F_\omega} lookup\,\Gamma\,A\,env\,[\![\Gamma \ni_F n : A]\!] = \lceil n \rceil_{env} : A$$
2. *If* $\Delta \vdash_{F_\omega} env : \Gamma$ *and* $\Gamma \vdash_F M : A$, *then*
$$\Delta \vdash_{F_\omega} eval\,\Gamma\,A\,env\,[\![\Gamma \vdash_F M : A]\!] = \lceil M \rceil_{env} : A$$

$$
\begin{aligned}
[\![\Gamma \ni_F A]\!] &: \mathit{Var}\,\Gamma\,A \\
[\![\Gamma, A \ni_F \mathsf{z} : A]\!] &= \mathsf{inj}_1\,(\mathsf{pack}\,[\Gamma](\mathit{refl}\,(\Gamma \times A))\,\mathsf{as}\,Z\,(\Gamma \times A)\,A) \\
[\![\Gamma, A \ni_F \mathsf{s}\,n : B]\!] &= \mathsf{inj}_2\,(\mathsf{pack}\,[\Gamma\,A](\mathit{refl}\,(\Gamma \times A), [\![\Gamma \ni_F n : B]\!])\,\mathsf{as}\,S\,(\Gamma \times A)\,B)
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash_F A]\!] &: \mathit{Exp}\,\Gamma\,A \\
[\![\Gamma, A \vdash_F n : A]\!] &= \mathsf{inj}_1\,([\![\Gamma, A \ni_F n : A]\!]) \\
[\![\Gamma \vdash_F \lambda^A.M : A \to B]\!] &= \mathsf{inj}_2\,(\mathsf{pack}\,[A\,B](\mathit{refl}\,(A \to B), [\![\Gamma, A \vdash_F M : B]\!])\,\mathsf{as}\,\mathit{Lam}\,\Gamma\,(A \to B)) \\
[\![\Gamma \vdash_F M^{A \to B}\,N : B]\!] &= \mathsf{inj}_3\,(\mathsf{pack}\,[A](([\![\Gamma \vdash_F M : A \to B]\!], [\![\Gamma \vdash_F N : A]\!])\,\mathsf{as}\,\mathit{App}\,\Gamma\,B) \\
[\![\Gamma \vdash_F \Lambda\alpha.M : \forall\alpha.A]\!] &= \mathsf{inj}_4\,(\mathsf{pack}\,[\forall\alpha.A](\mathit{refl}\,(\forall\alpha.A), \Lambda\alpha.[\![\Gamma \vdash_F M : A]\!])\,\mathsf{as}\,\mathit{TLam}\,\Gamma\,(\forall\alpha.A)) \\
[\![\Gamma \vdash_F M^{\forall\alpha.B'}\,A : B]\!] &= \mathsf{inj}_5\,(\mathsf{pack}\,[A\,(\lambda\alpha.B')](\mathit{refl}\,B, [\![\Gamma \vdash_F M : \forall\alpha.B']\!])\,\mathsf{as}\,\mathit{TApp}\,\Gamma\,B)
\end{aligned}
$$

$$
\begin{aligned}
\lceil \mathsf{z} \rceil_{env} &= \mathsf{proj}_2\ env & \lceil \lambda^A.M \rceil_{env} &= \lambda x^A.\lceil M \rceil_{(env,x)}, \quad x \text{ fresh} \\
\lceil \mathsf{s}\,n \rceil_{env} &= \lceil n \rceil_{(\mathsf{proj}_1\ env)} & \lceil M\,N \rceil_{env} &= \lceil M \rceil_{env} \lceil N \rceil_{env} \\
& & \lceil \Lambda\alpha.M \rceil_{env} &= \Lambda\alpha.\lceil M \rceil_{env} \\
\lceil M \rceil &= \lceil M \rceil_{()} & \lceil M\,A \rceil_{env} &= \lceil M \rceil_{env}\,A
\end{aligned}
$$

Figure 6. The embedding and lifting functions

The proof is by induction on the structure of derivations.

COROLLARY 2. *If* $\vdash_F M : A$, *then*

$$\vdash_{F_\omega} \mathit{eval}\,\Gamma\,A\,()\,[\![\vdash_F M : A]\!] = \lceil M \rceil : A$$

4. System F in OCaml

We will now show that the whole development can be redone in OCaml. Mostly this makes the implementation more painful, primarily because the current version of OCaml 3.12.1 does not support GADTs and module syntax can be somewhat heavyweight as a way for implementing existentials. However, in at least one way the implementation is a little cleaner. Unlike in GHC, OCaml type constructors do not have to be boxed. This means that the expression language does not need to be extended with forms for injection and projection.

Our starting point is Yallop and Kiselyov's encoding of GADTs in OCaml [29]. As we have already observed, GADTs are essentially just existential types with type-level equality coercions. Existentials can be encoded in OCaml using higher-rank universal types, which can be encoded using records, objects, or modules. We will, however, adopt a more direct encoding in terms of first-class recursive modules. As we have already seen, equality conversions can be implemented using Leibniz equality, which is definable as soon as one has universal quantification over type constructors. Yallop and Kiselyov, implement Leibniz equality using first-class modules.

We will not repeat Yallop and Kiselyov's development here, but instead just include the interfaces we use. First we define a unary type constructor:

```
module type TyCon = sig type 'a tc end
```

As modules are first-class in OCaml 3.12, we can treat this module signature as a first-class type.

The signature for Leibniz equality is show in Figure 7. For our purposes we only need the equality type `eq`, the reflexivity axiom `refl`, and the cast operation `cast`.

We begin with the encoding of variable typing judgements, which is shown in Figure 8. The sub-modules `Z` and `S` each represents an existential. The existentially bound type variables are those that are not bound in the `var` data type. The value components of the module define the body of the existential. For instance the `Z` module represents the type $\exists\Delta.(\Gamma, \Delta * \alpha)\ eq$. It uses Leibniz equality to encode a proof that there exists some Δ such that Γ is equal to $\Delta * \alpha$.

```
module type EQ =
sig
  (* A value of type (s, t) eq is a proof that types
     s and t are the same. *)
  type ('a, 'b) eq

  (* The reflexivity axiom. *)
  val refl : unit → ('a, 'a) eq

  (* Leibniz's substitution axiom. *)
  module Subst (TC : TyCon) :
  sig
    val subst : ('a, 'b) eq → ('a TC.tc, 'b TC.tc) eq
  end

  (* Given a proof that type s and type t are equal, we
     can convert s to t. *)
  val cast : ('a, 'b) eq → 'a → 'b
end
```

Figure 7. Leibniz equality in OCaml

```
module rec Var : sig
  module type Z =
  sig
    type Γ   type a
    type Δ
    val p : (Γ, Δ * a) eq
  end

  module type S =
  sig
    type Γ    type a
    type Δ    type b
    val p : (Γ, Δ * b) eq
    val n : (Δ, a) Var.var
  end

  type ('Γ,'a) var =
    | Z of (module Z with type Γ='Γ and type a='a)
    | S of (module S with type Γ='Γ and type a='a)
end = Var

open Var
```

Figure 8. De Bruijn variables in OCaml

```
module rec Exp : sig
  module type LAM =
  sig
    type Γ    type a
    type b    type c
    val p : (b → c, a) eq
    val e : (Γ * b, c) Exp.exp
  end

  module type APP =
  sig
    type Γ    type a
    type b
    val f : (Γ, b → a) Exp.exp
    val e : (Γ, b) Exp.exp
  end

  module type TLAM =
  sig
    type Γ    type a
    type 'b tc
    val p : (⟨poly : 'b.'b tc⟩, a) eq
    val e : ⟨poly : 'b.unit → (Γ, 'b tc) Exp.exp⟩
  end

  module type TAPP =
  sig
    type Γ    type a
    type b    type 'c tc
    val p : (b tc, a) eq
    val e : (Γ, ⟨poly : 'c.'c tc⟩) Exp.exp
  end

  type ('Γ,'a) exp =
    | Var  of ('Γ,'a) var
    | Lam  of (module LAM  with type Γ='Γ and type a='a)
    | App  of (module APP  with type Γ='Γ and type a='a)
    | TLam of (module TLAM with type Γ='Γ and type a='a)
    | TApp of (module TAPP with type Γ='Γ and type a='a)
end = Exp

open Exp
```

Figure 9. System F expressions in OCaml

The encoding of expressions (Figure 9) is similar. Polymorphic types are boxed inside an object type similarly to the `Poly` `newtype` used in the GHC implementation. Notice, however, that we create a thunk for the body of a type abstraction. This is in order to prevent the ML value restriction from getting in the way of polymorphism. We choose objects here instead of records as being structural there fields do not pollute the global namespace.

Before writing some examples, we first define some smart constructors (Figures 10 and 11). These allow us to invoke each non-polymorphic syntax constructor without having to explicitly define a module, and to invoke each polymorphic syntax constructor through a module that defines the appropriate type constructor. For the polymorphic argument to `TL.tlam` we use a record instead of an object as callers of the function will actually have to create these values and record creation syntax is significantly more concise than object creation syntax.

Now, consider the polymorphic identity function. Its type constructor is given by the module:

```
module AtoA = struct type 'a tc = 'a → 'a end
```

We write the polymorphic identity function as:

```
let id_poly () : (unit, ⟨poly : 'a.'a → 'a⟩) exp =
  let module M = TL(AtoA) in
```

```
let z : 'Γ 'a . unit → ('Γ * 'a, 'a) var =
    fun (type Δ') (type a') () →
      let module M = struct
        type Γ = (Δ' * a')
        type a = a'
        type Δ = Δ'
        let p = refl()
      end in
        Z (module M : Var.Z with
             type Γ=(Δ' * a') and type a=a')

let s : 'Δ 'b 'a . ('Δ, 'a) var → ('Δ * 'b, 'a) var =
  fun (type Δ') (type b') (type a') n' →
    let module M = struct
      type Γ = (Δ' * b')
      type a = a'
      type Δ = Δ'
      type b = b'
      let p = refl()
      let n = n'
    end in
      S (module M : Var.S with
           type Γ=(Δ' * b') and type a=a')
```

Figure 10. Smart constructors for de Bruijn variables in OCaml

```
      M.tlam {M.poly=fun () → lam (var (z()))}
```

and we can specialise it to integers as follows:

```
  let id_int : (unit, int → int) exp =
    let module M = TA(AtoA) in
      M.tapp (id_poly())
```

Notice that although the syntax is not exactly concise, unlike in GHC we do not have to box and unbox type constructors: OCaml knows that `'a AtoA.tc` is the same type as `'a→'a`.

Now let us consider the `K` combinator. As OCaml supports local module declarations, we need only ever define unary type constructors:

```
let ck () :
  (unit, ⟨ poly : 'a.⟨poly : 'b.'a → 'b → 'a⟩ ⟩) exp =
  let module M =
    TL(struct
         type 'a tc = ⟨poly : 'b.'a → 'b → 'a⟩ end) in
      M.tlam {M.poly = fun (type a) () →
        let module N =
          TL(struct type 'b tc = a → 'b → a end) in
            N.tlam {N.poly=fun () →
                      lam(lam(var(s(z()))))}}
```

Also note that we can inline functor parameters.

We can write an evaluator for our OCaml implementation (Figure 12) that is essentially the same as the F_ω one. The only real difference is the boilerplate for managing modules and polymorphism.

5. F_ω in Haskell

GHC 7.4 supports polymorphism over kinds. It turns out that this means our embedding of System F is actually an embedding of F_ω. All we need to do is generalise the argument of `Poly` to operate over arbitrary kinds. In fact we do not need to change the code at all, we just need to enable the `PolyKinds` language extension. Without `PolyKinds`, the `Poly` constructor is assigned the type:

```
Poly :: ( ∀a. k a) → Poly k
```

With `PolyKinds` enabled, it receives the mysterious type:

```
Poly :: ( ∀a. k a) → Poly ⋆ k
```

```
let rec lookup : 'Γ 'a . 'Γ → ('Γ,'a) var → 'a =
  fun (type Γ) (type a) env →
    function
      | Z m →
        let module M = (val m : Z with type Γ=Γ and type a=a) in
          snd (cast M.p env)
      | S m →
        let module M = (val m : S with type Γ=Γ and type a=a) in
          lookup (fst (cast M.p env)) M.n

let rec eval : 'Γ 'a . 'Γ → ('Γ,'a) Exp.exp → 'a =
  fun (type Γ) (type a) env →
    function
      | Var n → lookup env n
      | Lam m →
          let module M = (val m : LAM with type Γ=Γ and type a=a) in
            cast M.p (fun v → eval (env, v) M.e)
      | App m →
          let module M = (val m : APP with type Γ=Γ and type a=a) in
            (eval env M.f) (eval env M.e)
      | TLam m →
          let module M = (val m : TLAM with type Γ=Γ and type a=a) in
            (cast M.p) (object method poly : 'b.'b M.tc = eval env (M.e#poly ()) end)
      | TApp m →
          let module M = (val m : TAPP with type Γ=Γ and type a=a) in
            (cast M.p) (eval env M.e)#poly
```

Figure 12. System F evaluator in OCaml

which indicates that the `k` type constructor is polymorphic. Currently there is no way for the programmer to assert that a kind should be polymorphic, but GHC can infer this for itself.

Given that we can embed F_ω in GHC and we can embed System F in F_ω, we could of course embed System F in the embedded F_ω!

As a simple example, let us just consider the basic interface to Leibniz equality. The type constructor is:

```
newtype Leibniz a b (k :: ⋆ → ⋆) =
  Leibniz { unLeibniz :: k a → k b }
    deriving (Iso (k a → k b))
```

The definitions of `refl` and `cast` are straightforward:

```
newtype Id a = Id { unId :: a }
  deriving (Iso a)

refl :: Exp () (Poly (Leibniz a a))
refl = TLam (inj Leibniz (Lam (Var Z)))

cast :: Exp () (Poly (Leibniz a b) → Id a → Id b)
cast = Lam (proj Leibniz (TApp (Var Z)))
```

Unfortunately, the definition of `cast` leaves us with some inconvenient `Id` constructors. We can eliminate them by η-expanding:

```
cast' :: Exp () (Poly (Leibniz a b) → a → b)
cast' = Lam (Lam
         (proj Id (App (proj Leibniz (TApp (Var (S Z))))
                       (inj Id (Var Z)))))
```

As illustrated by `cast`, GHC's second-class type constructors are generally more problematic in the F_ω setting.

6. HOAS, type representations and unembedding

6.1 HOAS

Higher-order abstract syntax (HOAS) [21], is often more convenient for writing terms than a de Bruijn representation, because it allows the programmer to use the bindings of the host language to encode bindings of an object language. Others [22, 25] have considered HOAS embeddings of System F in F_ω (and in the latter case, embeddings of richer languages in extensions of F_ω). We can straightforwardly adapt Rendel et al.'s embedding [25], and implement it in Haskell. This also scales to an embedding of F_ω, taking advantage of kind polymorphism.

First, let us begin with a HOAS representation for simply-typed lambda calculus.

```
class Lambda exp where
    lam   :: (exp a → exp b) → exp (a → b)
    app   :: exp (a → b) → exp a → exp b
```

Following our previous work on *unembedding* [3], where we used the so-called "finally tagless" approach [7] (which amounts to a Scott-Mogensen encoding of lambda terms [20]), we encode HOAS terms using a Haskell type class. The idea is that different instances of the `Lambda` type class implement different interpreters for simply-typed lambda calculus.

We can straightforwardly extend `Lambda` with polymorphism in almost exactly the same way that we added polymorphism to the first-order representation.

```
class Lambda exp where
    lam      :: (exp a → exp b) → exp (a → b)
    app      :: exp (a → b) → exp a → exp b
    tlam     :: ( ∀a.exp (k a)) → exp (Poly k)
    tapp     :: exp (Poly k) → ( ∀a.exp (k a))
    hinject  :: Iso a b ⇒ exp a → exp b
    hproject :: Iso a b ⇒ exp b → exp a
```

The first four functions above have almost the same signatures as the embedding of System F in Rendel et al.'s work [25] (except they embed a richer language in a richer language). The last two functions deal with boxing and unboxing constructors. As with the first-order case, it is convenient to define wrapper functions in order to aid type inference:

```
hinj :: (Lambda exp, Iso a b) ⇒
          (a → b) → exp a → exp b
hinj _ = hinject

hproj :: (Lambda exp, Iso a b) ⇒
```

```
(* simple terms *)
let var : 'Γ 'a.('Γ, 'a) var → ('Γ, 'a) exp =
  fun n → Var n

let lam :  'Γ 'b 'c.(('Γ * 'b), 'c) exp →
                          ('Γ, 'b → 'c) exp =
  fun (type Γ') (type b') (type c') body →
    let module M = struct
      type Γ = Γ'
      type a = b' → c'
      type b = b'
      type c = c'
      let p = refl()
      let e = body
    end in
      Lam (module M : Exp.LAM with
             type Γ=Γ' and type a=b' → c')

let app :  'Γ 'b 'a.('Γ, 'b → 'a) exp →
('Γ, 'b) exp → ('Γ, 'a) exp =
  fun (type Γ') (type b') (type a') f e →
    let module M = struct
      type Γ = Γ'
      type a = a'
      type b = b'
      let f = f
      let e = e
    end in
      App (module M : Exp.APP with
             type Γ=Γ' and type a=a')

(* polymorphic terms *)
module TL (TC : TyCon) = struct
  type 'Γ body = {poly : 'a.unit → ('Γ, 'a TC.tc) exp}
  let tlam : 'Γ.'Γ body →
            ('Γ, ⟨poly : 'b.'b TC.tc⟩) exp =
    fun (type Γ') body →
      let module M = struct
        type Γ = Γ'
        type a = ⟨poly : 'b.'b TC.tc⟩
        type 'a tc = 'a TC.tc
        let p = refl()
        let e =
          object
            method poly : 'a.unit →
                    (Γ, 'a TC.tc) exp=body.poly
          end
      end in
        TLam (module M : Exp.TLAM with
                type Γ=Γ' and
                type a = ⟨poly : 'b.'b TC.tc⟩)
end

module TA (TC : TyCon) = struct
  let tapp : 'Γ 'b.('Γ, ⟨poly : 'a.'a TC.tc⟩) exp →
                                  ('Γ, 'b TC.tc) exp =
    fun (type Γ') (type b') body →
      let module M = struct
        type Γ = Γ'
        type b = b'
        type 'a tc = 'a TC.tc
        type a = b tc
        let p = refl()
        let e = body
      end in
        TApp (module M : Exp.TAPP with
                type Γ=Γ' and type a=b' TC.tc)
end
```

Figure 11. Smart constructors for System F in OCaml

```
            (a → b) → exp b → exp a
  hproj _ = project
```

Closed HOAS terms of type `a` are represented simply as `Lambda` expressions of type `a`:

```
type ClosedHoas a = Lambda exp ⇒ exp a
```

For instance, we can now define the polymorphic identity function as:

```
hoas_id_poly :: ClosedHoas (Poly AtoA)
hoas_id_poly = tlam (hinj AtoA (lam (λx → x)))
```

One of the canonical instances we can specify defines an evaluator for HOAS System F terms:

```
newtype Val a = Val {unVal :: a}

instance Lambda Val where
    lam f             = Val $ unVal ∘ f ∘ Val
    Val f `app` Val a = Val $ f a
    tlam v            = Val $ Poly (unVal v)
    tapp v            = Val $ unPoly (unVal v)
    hinject v         = Val $ inject (unVal v)
    hproject v        = Val $ project (unVal v)
```

Carette et al. [7] explore a range of interesting interpretations for implementing fold-like operations on simply-typed languages. In Section 6.3 we provide another canonical instance for converting the HOAS representation to a first-order representation.

6.2 Type representations

One issue with naive attempts at implementing HOAS is that in order to correctly represent an embedded language we need to restrict the form of host language functions admissible in the encoding of object language lambdas. Without such a restriction, it becomes possible to write *exotic terms*, which can inspect the form of function arguments and behave differently depending on their form. Using the "finally tagless" / Scott-Mogensen encoding approach, this is not possible. In particular, in our type class-based implementation, the `exp` type constructor is abstract, so it is not possible to do anything untoward with function arguments.

A more subtle issue is that if we do not restrict types somehow, then we can create term representations whose types are not object language types: *exotic types*. For instance, in:

```
id_exotic :: Exp () (Var () Int → Var () Int)
id_exotic = Lam (Var Z)
```

the type `Var () Int -> Var () Int` is not a System F type. Exotic types are less of a concern than exotic terms, because exotic types behave like abstract types; they do not introduce any new behaviours. Nonetheless, we can rule out exotic types by encoding *representable types* [12] using GADTs and type classes. In fact, even our first-order encoding does not preclude exotic types.

For simple types, we can define the following GADT:

```
data Rep :: ⋆ → ⋆ where
   Int   :: Rep Int
   (:→) :: Rep a → Rep b → Rep (a → b)
```

By construction, if `v :: Rep A`, then `A` can only be built from the `Int` and `->` type constructors. For convenience, following Cheney and Hinze [10], we also define a type class `Representable`:

```
class Representable a where rep :: Rep a
instance Representable Int where rep = Int
instance (Representable a, Representable b) ⇒
  Representable (a → b) where
    rep = rep :→ rep
```

Now we can enforce simple typing by adding type class constraints of the form `Representable a =>`.

Let us now consider how to extend these definitions to enforce System F typing. In order to define representations for polymorphism and type constructor applications we need to define what it is to be a representable type constructor.

```
class RepresentableCon (k :: ⋆ → ⋆) where
instance (Iso a (k b)) ⇒ RepresentableCon k where
```

The `RepresentableCon` type class asserts that a type constructor `k` is representable if it is a `newtype` constructor for a `newtype` that derives the `Iso` type class. (Note that the constraint `Iso a (k b)` on the instance requires the `UndecidableInstances` language extension.)

Now we extend the `Rep` GADT:

```
data Rep :: ⋆ → ⋆ where
   Int   :: Rep Int
   (:→) :: Rep a → Rep b → Rep (a → b)
   All   :: (RepresentableCon k) ⇒ Rep (Poly k)
   Con   :: (RepresentableCon k) ⇒ Rep a → Rep (k a)
```

and correspondingly the `Representable` type class:

```
class Representable a where rep :: Rep a
instance Representable Int where rep = Int
instance (Representable a, Representable b) ⇒
  Representable (a → b) where
    rep = rep :→ rep
instance (RepresentableCon k) ⇒
  Representable (Poly k) where
    rep = All
instance (Representable a, RepresentableCon k) ⇒
  Representable (k a) where
    rep = Con rep
```

Having defined representable types, we can now put them to use. We use them to constrain the `Exp` type:

```
data Exp  :: ⋆ → ⋆ → ⋆ where
   Var     :: Representable a ⇒ Var Γ a → Exp Γ a
   Lam     :: Representable a ⇒
                Exp (a, Γ) b → Exp Γ (a → b)
   App     :: Exp Γ (a → b) → Exp Γ a → Exp Γ b
   TLam    :: ( ∀a.Representable a ⇒ Exp Γ (k a)) →
                                      Exp Γ (Poly k)
   TApp    :: Representable a ⇒
                Exp Γ (Poly k) → Exp Γ (k a)
   Inject  :: Iso a (k b) ⇒ Exp Γ a → Exp Γ (k b)
   Project :: Iso a (k b) ⇒ Exp Γ (k b) → Exp Γ a
```

The type of a variable must be representable. We need to constrain the argument to a lambda abstraction, but we do not need to constrain the return type as the other constraints ensure that it is only possible to construct a body that has a representable type. For polymorphism, we constrain the argument types for type constructors to be representable. We also need to amend `Poly` to quantify over representable types:

```
newtype Poly k =
  Poly {unPoly :: ∀a.Representable a ⇒ k a}
```

What we pay for this is that we can no longer embed F_ω, as `Poly` is no longer polymorphic in the kind `k`:

```
Poly :: ( ∀a.Representable a ⇒ k a) → Poly k
```

The modifications to the HOAS representation are similar:

```
class Lambda exp where
    lam      :: Representable a ⇒
                  (exp a → exp b) → exp (a → b)
    app      :: exp (a → b) → exp a → exp b
    tlam     :: ( ∀a.Representable a ⇒ exp (k a)) →
                                        exp (Poly k)
    tapp     :: exp (Poly k) →
                  ( ∀a.Representable a ⇒ exp (k a))
    hinject  :: Iso a (k b) ⇒ exp a → exp (k b)
    hproject :: Iso a (k b) ⇒ exp (k b) → exp a
```

Now if we write:

```
(Lam (Var Z)) :: Exp () (Exp () Int → Exp () Int)
```

then we get an error message:

```
No instance for (Iso a (Exp () b))
  arising from a use of ‘Lam’
Possible fix: add an instance declaration for
  (Iso a (Exp () b))
In the expression:
  (Lam (Var Z)) :: Exp () (Exp () Int → Exp () Int)
```

as `Exp () Int → Exp () Int` is not a representable System F type.

6.3 Unembedding

In prior work [3], we demonstrated how to *unembed* well-typed higher-order abstract syntax as well-typed first-order syntax for simply-typed object languages. We now show how to extend unembedding to polymorphic object languages.

The motivation for unembedding is that while higher-order abstract syntax is generally more convenient for the programmer to write, and also well-suited to implementing fold-like operations using the finally-tagless approach [7], first-order abstract syntax is often better suited for implementing intensional analyses (e.g., compiler optimisations typically operate on a first-order intermediate representation).

Following our previous work [3], we begin with a GADT for representing typing contexts:

```
data Ctx :: ⋆ → ⋆ where
   CtxZ :: Ctx ()
   CtxS :: Representable a ⇒ Ctx Γ → Ctx (Γ, a)
```

We now represent (open) terms as first-order System F expressions parameterised by typing contexts:

```
newtype Term a = Term {unTerm ::  ∀Γ.Ctx Γ → Exp Γ a}
```

The type class instance for interpretting HOAS terms as first-order terms is as follows:

```
instance Lambda Term where
   lam f      =
     Term $ λc → Lam (unTerm (f (Term $ λd →
                    Var (shift d (CtxS c)))) (CtxS c))
   (Term m) ‘app‘ (Term n) =
                 Term $ λc → App (m c) (n c)
   tlam m     = Term $ λc → TLam (unTerm m c)
   tapp m     = Term $ λc → TApp (unTerm m c)
   hinject m  = Term $ λc → Inject (unTerm m c)
   hproject m = Term $ λc → Project (unTerm m c)
```

It is entirely straightforward apart from the `shift` function for the `lam` case, which is far from straightforward. The issue is that we need to generate an occurrence of the bound variable, but it may be used inside other lambda abstractions, in which case we need to increment it for each one. Even in the untyped case, proving correctness requires parametricity [2], so we cannot hope to capture all of the desired static typing constraints in Haskell. We define `shift` as follows:

```
-- precondition: Ctx (Δ, a) is a prefix of Ctx Γ
shift :: Ctx Γ → Ctx (Δ, a) → Var Γ a
shift c1 c2 = shift' (len c1 - len c2) c1
```

```
  where
    shift' :: Int → Ctx Γ → Var Γ a
    shift' 0 (CtxS _) = unsafeCoerce Z
    shift' n (CtxS c) = S (shift' (n-1) c)

    len :: Ctx n → Int
    len CtxZ     = 0
    len (CtxS c) = 1 + len c
```

Previously [3], we used a type-safe cast [28] in place of the unsafe coercion, but given that this part of the code cannot be validated by GHC anyway, we choose to simplify it here and use `unsafeCoerce`. Informally, it is clear that the precondition will be satisfied, as inside the body of a lambda the typing context can only be extended. We leave formal verification in the style of Atkey [2] to future work.

For convenience, we wrap the `Term` type class in a function for converting closed HOAS expressions to closed first-order expressions:

```
toClosedExp :: ClosedHoas a → Exp () a
toClosedExp v = unTerm v CtxZ
```

Now we can write:

```
toClosedExp hoas_id_poly
```

One can of course extend the HOAS representation to open terms by parameterising by an environment representing free variables. Having done that one can define functions to convert from the HOAS representation to open first-order terms, and vice-versa:

```
type Hoas Γ a = Lambda exp ⇒ HEnv exp Γ → exp a
toExp :: Ctx Γ → Hoas Γ a → Exp Γ a
toHoas :: Exp Γ a → Hoas Γ a
```

(See [3] for details on how to implement these functions.) The extension to System F is straightforward.

7. Related work

Some related work has already been discussed in the main body of the paper.

Other unpublished attempts at embedding System F in Haskell include Ryan Ingram's de Bruijn representation [16] and Dan Burton's HOAS representation [6]. Both differ from our approach in that the representation of object types diverges significantly from the representation. The former uses a de Bruijn representation for type abstractions as well as term abstractions. This is also the approach taken, for instance, by Benton et al.'s encoding of System F in Coq [5].

A number of recent articles have considered languages that support *typed self-representation*, that is an internal typed representation of the entire host language. Rendel et al [25] generalise F_ω, defining a universal kind of kinds, which leads to an inconsistent logic, but also a language that can interpret itself. Jay and Palsberg [17] introduce a pattern calculus that supports typed-self interpretation. Carette and Stump [8] present a variant of lambda calculus called *Archon*, which supports "direct-reflection" — the ability to inspect and decompose all terms. They do not include a type system for Archon, but express an intention to design one.

Another line of work looks at encoding dependent type theory in dependent type theory using universe constructions [9, 13, 19].

8. Conclusion

We have succeeded in what we set out to do in the sense that we have developed well-typed first-order encodings of System F in Haskell and OCaml that use essentially the same representation for host and object language types. By extending unembedding to support polymorphism, we can even write embedded code using higher-order abstract syntax, and subsequently convert it to the first-order representation for intensional analysis.

We have not yet explored practical applications of this work. It is clear that extending embedded languages with polymorphism adds expressive power, but it it not yet clear to what extent this extra expressive power is necessary. Often one can get away with using the polymorphism of the host language as a proxy for polymorphism in the embedded language.

In Haskell, the second-class nature of type constructors makes our representation somewhat inconvenient to use. Although OCaml does not suffer from that problem, it is lacking in other ways. In particular, it does not have GADTs and the first-class module syntax is rather verbose for our purposes. OCaml version 4 will include GADTs, which should help somewhat.

Ultimately, we wonder whether meta programming based on well-typed quotation [23, 26] may prove more fruitful in practice.

Acknowledgments

I would like to thank Dimitrios Vytniotis for suggesting the use of `newtype deriving`, and Bob Atkey and the anonymous reviewers for helpful feedback. This work was supported by a Google Research Award.

References

[1] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL*, pages 453–468, 1999.

[2] R. Atkey. Syntax for free: Representing syntax with binding using parametricity. In *Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.

[3] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Haskell*, pages 37–48, 2009.

[4] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *ICFP '02*, pages 157–166, New York, NY, USA, 2002. ACM.

[5] N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 2011.

[6] D. Burton, 2012. `https://github.com/DanBurton/Blog/blob/master/Literate%20Haskell/SystemF.lhs`.

[7] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[8] J. Carette and A. Stump. Towards typing for small-step direct reflection. In *PEPM*, pages 93–96, 2012.

[9] J. Chapman. Type theory should eat itself. *Electr. Notes Theor. Comput. Sci.*, 228:21–36, 2009.

[10] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02*, New York, NY, USA, 2002. ACM.

[11] J. Cheney and R. Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003. `http://ecommons.library.cornell.edu/handle/1813/5614`.

[12] K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP*, pages 301–312, 1998.

[13] N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *TYPES*, pages 93–109, 2006.

[14] N. G. de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 1972.

[15] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[16] R. Ingram. Type-system fun: a type-safe embedding of System F lambda calculus into Haskell, 2007. `http://www.haskell.org/pipermail/haskell-cafe/2007-December/037246.html`.

[17] C. B. Jay and J. Palsberg. Typed self-interpretation by pattern matching. In *ICFP*, pages 247–258, 2011.

[18] Y. Mandelbaum and A. Stump. GADTs for the OCaml masses. In *Workshop on ML*, 2009.

[19] C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *WGP*, pages 1–12, 2010.

[20] T. Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[21] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

[22] F. Pfenning and P. Lee. Metacircularity in the polymorphic lambda-calculus. *Theor. Comput. Sci.*, 89(1):137–159, 1991.

[23] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*, 2008.

[24] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[25] T. Rendel, K. Ostermann, and C. Hofer. Typed self-representation. In *PLDI*, pages 293–303, 2009.

[26] M. Rhiger. Staged computation with staged lexical scope. In *ESOP*, pages 559–578, 2012.

[27] C. V. Russo and D. Vytiniotis. QML: Explicit first-class polymorphism for ML. In *Workshop on ML*, 2009.

[28] S. Weirich. Type-safe cast. *Journal of Functional Programming*, 14(6):681–695, 2004.

[29] J. Yallop and O. Kiselyov. First-class modules: hidden power and tantalizing promises. In *Workshop on ML*, 2010.