Unembedding Domain-Specific Languages

Robert Atkey Sam Lindley Jeremy Yallop

LFCS, School of Informatics, The University of Edinburgh

{bob.atkey,sam.lindley,jeremy.yallop}@ed.ac.uk

Abstract

Higher-order abstract syntax provides a convenient way of embedding domain-specific languages, but is awkward to analyse and manipulate directly.

We explore the boundaries of higher-order abstract syntax. Our key tool is the unembedding of embedded terms as de Bruijn terms, enabling intensional analysis. As part of our solution we present techniques for separating the definition of an embedded program from its interpretation, giving modular extensions of the embedded language, and different ways to encode the types of the embedded language.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (functional) programming

General Terms Languages, Theory

Keywords domain-specific languages, higher-order abstract syntax, type classes, unembedding

1. Introduction

Embedding a domain-specific language (DSL) within a host language involves writing a set of combinators in the host language that define the syntax and semantics of the embedded language. Haskell plays host to a wide range of embedded DSLs, including languages for database queries [Leijen and Meijer 1999], financial contracts [Peyton Jones et al. 2000], parsing [Leijen and Meijer 2001], web programming [Thiemann 2002], production of diagrams [Kuhlmann 2001] and spreadsheets [Augustsson et al. 2008].

An embedded language has two principal advantages over a stand-alone implementation. First, using the syntax and semantics of the host language to define those of the embedded language reduces the burden on both the implementor (who does not need to write a parser and interpreter from scratch) and the user (who does not need to learn an entirely new language and toolchain). Second, integration of the embedded language — with the host language, and with other DSLs — becomes almost trivial. It is easy to see why one might wish to use, say, languages for web programming and database queries within a single program; if both are implemented as embeddings into Haskell then integration is as straightforward as combining any other two libraries.

Perhaps the most familiar example of an embedded DSL is the monadic language for imperative programming that is part of the Haskell standard library. A notable feature of the monadic language is the separation between the definition of the symbols of the language, which are introduced as the methods of the *Monad* type class, and the interpretation of those symbols, given as instances of the class. This approach enables a range of interpretations to be associated with a single language — a contrast to the embedded languages enumerated earlier, which generally each admit a single interpretation.

If the embedded language supports binding a number of difficulties may arise. The interface to the embedded language must ensure that there are no mismatches between bindings and uses of variables (such as attempts to use unbound or incorrectly-typed variables); issues such as substitution and alpha-equivalence introduce further subtleties. *Higher-order abstract syntax* [Pfenning and Elliott 1988] (HOAS) provides an elegant solution to these difficulties. HOAS uses the binding constructs of the host language to provide binding in the embedded language, resulting in embedded language binders that are easy both to use and to interpret.

However, while HOAS provides a convenient interface to an embedded language, it is a less convenient representation for encoding certain analyses. In particular, it is difficult to perform intensional analyses such as closure conversion or the shrinking reductions optimisation outlined in Section 2.4, as the representation is constructed from functions, which cannot be directly manipulated.

It is clear that higher-order abstract syntax and inductive term representations each have distinct advantages for embedded languages. Elsewhere, the first author provides a proof that the higher-order abstract syntax representation of terms is isomorphic to an inductive representation [Atkey 2009a]. Here we apply Atkey's result, showing how to convert between the two representations, and so reap the benefits of both.

We summarise the contents and contributions of this paper as follows:

- We start in Section 2 with an embedding of the untyped λ-calculus, using the parametric polymorphic representation of higher-order abstract syntax terms. This representation was advocated by Washburn and Weirich [2008], but dates back to at least Coquand and Huet [1985]. We show how to convert this representation to a concrete de Bruijn one, using the mapping defined in Atkey [2009a]. This allows more straightforward expression of intensional analyses, such as the shrinking reductions optimisation.

  We then examine the proof of the isomorphism between the HOAS and de Bruijn representations in more detail to produce an almost fully well-typed conversion between the Haskell HOAS type and a GADT representing well-formed de Bruijn terms. Interestingly, well-typing of this conversion relies on the parametricity of Haskell's polymorphism, and so even complex extensions to Haskell's type system, such as dependent types, would not be able to successfully type this translation. Our first

Haskell'09, September 3, 2009, Edinburgh, Scotland, UK.

main contribution is the explanation and translation of the proof into Haskell.

- Our representation of embedded languages as type classes is put to use in Section 3, where we show how to modularly construct embedded language definitions. For example, we can independently define language components such as the λ-calculus, booleans and arithmetic. Our second main contribution is to show how to extend an embedded language with flexible pattern matching and how to translate back-and-forth to well-formed de Bruijn terms.
- Having explored the case for untyped languages we turn to typed languages in Section 4. We carefully examine the issue of how embedded language types are represented, and work to ensure that type variables used in the representation of embedded language terms do not leak into the embedded language itself. Thus we prevent *exotically typed* terms as well as exotic terms in our HOAS representation. As far as we are aware, this distinction has not been noted before by other authors using typed HOAS, e.g. [Carette et al. 2009]. Our third main contribution is the extension of the well-typed conversion from HOAS to de Bruijn to the typed case, identifying where we had to circumvent the Haskell typechecker. Another contribution is the identification and explanation of exotically typed terms in Church encodings, a subject we feel deserves further study.
- Our final contributions are two larger examples in Section 5: unembedding of mobile code from a convenient higher-order abstract syntax representation, and an embedding of the Nested Relational Calculus via higher-order abstract syntax.
- Section 6 surveys related work.

The source file for this paper is a literate Haskell program. The extracted code and further examples are available at the following URL:
http://homepages.inf.ed.ac.uk/ratkey/unembedding/.

2. Unembedding untyped languages

We first explore the case for untyped embedded languages. Even without types at the embedded language level, an embedding of this form is not straightforward, due to the presence of variable binding and α-equivalence in the embedded language. We start by showing how to handle the prototypical language with binding.

2.1 Representing the λ-calculus

Traditionally, the λ-calculus is presented with three term formers: variables, λ-abstractions and applications. Since we are using the host-language to represent embedded language variables, we reduce the term formers to two, and place them in a type class:

```
class UntypedLambda exp where
  lam :: (exp → exp) → exp
  app :: exp → exp → exp
```

To represent closed terms, we abstract over the type variable `exp`, where `exp` is an instance of `UntypedLambda`:

```
type Hoas = ∀exp. UntypedLambda exp ⇒ exp
```

Encoding a given untyped λ-calculus term in this representation becomes a matter of taking the term you first thought of, inserting `lam`s and `app`s into the correct places, and using Haskell's own binding and variables for binding and variables in the embedded-language. For example, to represent the λ-calculus term $\lambda x.\lambda y.xy$, we use:

```
example1 :: Hoas
example1 = lam (λx → lam (λy → x ‘app‘ y))
```

Our host language, Haskell, becomes a macro language for our embedded language. As an example, this function creates Church numerals for any given integer:

```
numeral :: Integer → Hoas
numeral n = lam (λs → (lam (λz → body s z n)))
  where body s z 0 = z
        body s z n = s ‘app‘ (body s z (n-1))
```

Following the work of Pfenning and Elliott [1988], the use of host language binding to represent embedded language binding has also been attempted by the use of algebraic datatypes. For example, Fegaras and Sheard [1996] start from the following datatype:

```
data Term = Lam (Term → Term)
          | App Term Term
```

One can use this datatype to write down representations of terms, but Fegaras and Sheard are forced to extend this in order to define folds over the abstract syntax trees:

```
data Term a = Lam (Term a → Term a)
            | App (Term a) (Term a)
            | Var a
```

The additional constructor and type argument are used in the implementation of the `fold` function to pass accumulated values through. It is not intended that the `Var` constructor be used in user programs.

The problem with this representation is that it permits so-called *exotic terms*, members of the type that are not representatives of λ-calculus terms. For example:

```
Lam (λx → case x of Lam _   → x
                    | App _ _ → Lam (λx → x))
```

The body of the λ-abstraction in this "term" is either x or $\lambda x.x$, depending on whether the passed in term is itself a λ-abstraction or an application. Fegaras and Sheard mitigate this problem by defining an ad-hoc type system that distinguishes between datatypes that may be analysed by cases and those that may be folded over as HOAS. The type system ensures that the `Var` constructor is never used by the programmer.

The advantage of the HOAS representation that we use, which was originally proposed by Coquand and Huet [1985], is that exotic terms are prohibited [Atkey 2009a] (with the proviso that infinite terms are allowed when we embed inside Haskell). In our opinion, it is better to define types that tightly represent the data we wish to compute with, and not to rely on the discipline of failible programmers or ad-hoc extensions to the type system.

2.2 Folding over Syntax

Our representation of closed λ-terms amounts to a Church encoding of the syntax of the calculus, similar to the Church encodings of inductive datatypes such as the natural numbers. Unfolding the type `Hoas`, we can read it as the System F type:

$$C_\lambda = \forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

Compare this to the Church encoding of natural numbers:

$$C_{\mathsf{nat}} = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$$

For C_{nat}, we represent natural numbers by their fold operators. A value of type C_{nat}, given some type α and two constructors, one of type α and one of type $\alpha \to \alpha$ (which we can think of as zero and successor), must construct a value of type α. Since the type α is unknown when the value of type C_{nat} is constructed, we can only use these two constructors to produce a value of type α. It is this property that ensures that we only represent natural numbers.

Likewise, for the C_λ type, we have an abstract type α, and two constructors, one for λ-abstraction and one for application. The construction for λ-abstraction is special in that there is a negative occurence of α in its arguments. This does not fit into the classical theory of polymorphic Church encodings, but is crucial to the HOAS representation of binding. We sketch how parametricity is used below, in Section 2.6.

As for the Church encoded natural numbers, we can treat the type C_λ as a fold operator over terms represented using HOAS. We can use this to compute over terms, as demonstrated by Washburn and Weirich [2008]. Returning to Haskell, folds over terms are expressed by giving instances of the `UntypedLambda` type class. For example, to compute the size of a term:

```
newtype Size = Size { size :: Integer }

instance UntypedLambda Size where
  lam f     = Size $ 1 + size (f (Size 1))
  x ‘app‘ y = Size $ 1 + size x + size y

getSize :: Hoas → Integer
getSize term = size term
```

The case for `app` is straightforward; the size of an application is one plus the sizes of its subterms. For a λ-abstraction, we first add one for the λ itself, then we compute the size of the body. As we represent bodies by host-language λ-abstractions we must apply them to something to get an answer. In this case the body `f` will have type `Size → Size`, so we pass in what we think the size of a variable will be, and we will get back the size of the whole subterm.

A more exotic instance of a fold over the syntax of a λ-term is the denotational semantics of a term, i.e. an evaluator. We first define a “domain” for the semantics of the call-by-name λ-calculus:

```
data Value = VFunc (Value → Value)
```

Now the definitions for `lam` and `app` are straightforward:

```
instance UntypedLambda Value where
  lam f             = VFunc f
  (VFunc f) ‘app‘ y = f y

eval :: Hoas → Value
eval term = term
```

2.3 Unembedding the λ-calculus

Writing computations over the syntax of our embedded language is all well and good, but there are many functions that we may wish to express that are awkward, inefficient, or maybe impossible to express as folds. However, the HOAS representation is certainly convenient for embedding embedded language terms inside Haskell, so we seek a conversion from HOAS to a form that is amenable to intensional analysis.

A popular choice for representing languages with binding is de Bruijn indices, where each bound variable is represented as a pointer to the binder that binds it [de Bruijn 1972]. We can represent de Bruijn terms by the following type:

```
data DBTerm = Var Int
            | Lam DBTerm
            | App DBTerm DBTerm
            deriving (Show,Eq)
```

To convert from `Hoas` to `DBTerm`, we abstract over the number of binders that surround the term we are currently constructing.

```
newtype DB = DB { unDB :: Int → DBTerm }
```

The intention is that `unDB x n` will return a de Bruijn term, closed in a context of depth `n`. To define a fold over the HOAS representation, we give an instance of `UntypedLambda` for `DB`:

```
instance UntypedLambda DB where
  lam f   = DB $ λi → let v = λj → Var (j-(i+1)) in
                      Lam (unDB (f (DB v)) (i+1))
  app x y = DB $ λi → App (unDB x i) (unDB y i)

toTerm :: Hoas → DBTerm
toTerm v = unDB v 0
```

Converting a HOAS application to a de Bruijn application is straightforward; we simply pass through the current depth of the context to the subterms. Converting a λ-abstraction is more complicated. Clearly, we must use the `Lam` constructor to generate a de Bruijn λ-abstraction, and, since we are going under a binder, we must up the depth of the context by one. As with the size example above, we must also pass in a representation of the bound variable to the host-language λ-abstraction representing the body of the embedded language λ-abstraction. This representation will be instantiated at some depth `j`, which will always be greater than `i`. We then compute the difference between the depth of the variable and the depth of the binder as `j−(i+1)`, which is the correct de Bruijn index for the bound variable.

We can represent an open HOAS term as a function from an environment, represented as a list of HOAS terms, to a HOAS term.

```
type Hoas’ = ∀exp.UntypedLambda exp ⇒ [exp] → exp
```

It is worth pointing out that this encoding is technically incorrect as such functions can inspect the length of the list and so need not represent real terms. We could rectify the problem by making environments total, that is, restricting them to be infinite lists (where cofinitely many entries map variables to themselves). Rather than worrying about this issue now we resolve it later when we consider well-formed de Bruijn terms in Section 2.6.

Now we can convert an open HOAS term to a de Bruijn term by first supplying it with a total environment mapping every variable to itself, interpreting everything in the `DB` instance of `UntypedLambda` as we do for closed terms.

```
toTerm’ :: Hoas’ → DBTerm
toTerm’ v = unDB w 0
  where w     = v (env 0)
        env j = DB (λi → Var (i+j)) : env (j+1)
```

Conversion from HOAS to de Bruijn representations have already been presented by other workers; see, for example, some slides of Olivier Danvy[1]. In his formulation, the HOAS terms are represented by the algebraic datatype we saw in Section 2.1. Hence exotic terms are permitted by the type, and it seems unlikely that his conversion to de Bruijn could be extended to a well-typed one in the way that we do below in Section 2.6.

2.4 Intensional analysis

The big advantage of converting HOAS terms to de Bruijn terms is that this allows us to perform intensional analyses. As a simple example of an analysis that is difficult to perform directly on HOAS terms we consider *shrinking reductions* [Appel and Jim 1997]. Shrinking reductions arise as the restriction of β-reduction (i.e. inlining) to cases where the bound variable is used zero (dead-code elimination) or one (linear inlining) times. As well as reducing function call overhead, shrinking reductions expose opportunities for further optimisations such as common sub-expression elimination and more aggressive inlining.

The difficulty with implementing shrinking reductions is that dead-code elimination at one redex can expose further shrinking reductions at a completely different position in the term, so attempts at writing a straightforward compositional algorithm fail. We give

[1] http://www.brics.dk/~danvy/Slides/mfps98-up2.ps. Thanks to an anonymous reviewer for this link.

a naive algorithm that re-traverses the whole reduct whenever a redex is reduced. The only interesting case in the `shrink` function is that of a β-redex where the number of uses is less than or equal to one. This uses the standard de Bruijn machinery to perform the substitution [Pierce 2002]. More efficient imperative algorithms exist [Appel and Jim 1997, Benton et al. 2004, Kennedy 2007]. The key point is that these algorithms are intensional. It seems unlikely that shrinking reductions can be expressed easily as a fold.

```
usesOf n (Var m)     = if n==m then 1 else 0
usesOf n (Lam t)     = usesOf (n+1) t
usesOf n (App s t)   = usesOf n s + usesOf n t

lift m p (Var n)  | n < p     = Var n
                  | otherwise = Var (n+m)
lift m p (Lam body)  = Lam (lift m (p+1) body)
lift m p (App s t)   = App (lift m p s) (lift m p t)

subst m t (Var n) | n==m     = t
                  | n > m     = Var (n-1)
                  | otherwise = Var n
subst m t (Lam s)    = Lam (subst (m+1) (lift 1 0 t) s)
subst m t (App s s') = App (subst m t s) (subst m t s')

shrink (Var n)   = Var n
shrink (Lam t)   = Lam (shrink t)
shrink (App s t) =
  case s' of
    Lam u | usesOf 0 u ≤ 1 → shrink (subst 0 t' u)
    _                      → App s' t'
  where s' = shrink s
        t' = shrink t
```

2.5 Embedding again

Before we explain why the unembedding process works, we note that going from closed de Bruijn terms back to the HOAS representation is straightforward.

```
fromTerm' :: DBTerm → Hoas'
fromTerm' (Var i)  env = env !! i
fromTerm' (Lam t)  env = lam (λx → fromTerm' t (x:env))
fromTerm' (App x y) env =
  fromTerm' x env `app` fromTerm' y env

fromTerm :: DBTerm → Hoas
fromTerm term = fromTerm' term []
```

We maintain an environment storing all the representations of bound variables that have been acquired down each branch of the term. When we go under a binder, we extend the environment by the newly abstracted variable. This definition is unfortunately partial (due to the indexing function `(!!)`) since we have not yet guaranteed that the input will be a closed de Bruijn term. In the next sub-section we resolve this problem.

2.6 Well-formed de Bruijn terms

We can guarantee that we only deal with closed de Bruijn terms by using the well-known encoding of de Bruijn terms into GADTs [Sheard et al. 2005]. In this representation, we explicitly record the depth of the context in a type parameter. We first define two vacuous type constructors to represent natural numbers at the type level.

```
data Zero
data Succ a
```

To represent variables we make use of the `Fin` GADT, where the type `Fin n` represents the type of natural numbers less than `n`. The `Zero` and `Succ` type constructors are used as phantom types.

```
data Fin :: ⋆ → ⋆ where
  FinZ :: Fin (Succ a)
  FinS :: Fin a → Fin (Succ a)
```

The type of well-formed de Bruijn terms for a given context is captured by the following GADT. The type `WFTerm Zero` will then represent all closed de Bruijn terms.

```
data WFTerm :: ⋆ → ⋆ where
  WFVar :: Fin a → WFTerm a
  WFLam :: WFTerm (Succ a) → WFTerm a
  WFApp :: WFTerm a → WFTerm a → WFTerm a
```

Writing down terms in this representation is tedious due to the use of `FinS (FinS FinZ)` etc. to represent variables. The HOAS approach has a definite advantage over de Bruijn terms in this respect.

The `toTerm` function we defined above always generates closed terms, and we now have a datatype that can be used to represent closed terms. It is possible to give a version of `toTerm` that has the correct type, but we will have to work around the Haskell type system for it to work. To see why, we sketch the key part of the proof of adequacy of the Church encoding of λ-calculus syntax—the type C_λ—given by the first author [Atkey 2009a].

As alluded to above, the correctness of the Church encoding method relies on the parametric polymorphism provided by the $\forall\alpha$ quantifier. Given a value of type α, the only action we can perform with this value is to use it as a variable; we cannot analyse values of type α, for if we could, then our function would not be parametric in the choice of α. The standard way to make such arguments rigorous is to use Reynolds' formalisation of parametricity [Reynolds 1974] that states that for any choices τ_1 and τ_2 for α, and any binary relation between τ_1 and τ_2, this relation is preserved by the implementation of the body of the type abstraction.

To prove that the `toTerm` function always produces well-formed de Bruijn terms, we apply Reynolds' technique with two minor modifications: we restrict to unary relations and we index our relations by natural numbers. The indexing must satisfy the constraint that if $R^i(x)$ and $j \geq i$, then $R^j(x)$. This means that we require *Kripke* relations over the usual ordering on the natural numbers.

In the `toTerm` function, we instantiate the type α with the type `Int → DBTerm`. The Kripke relation we require on this type is $R^i(\mathtt{t}) \Leftrightarrow \forall j \geq i.\ j \vdash (\mathtt{t}\ j)$, where $j \vdash t$ means that the de Bruijn term t is well-formed in contexts of depth j. If we know $R^0(\mathtt{t})$, then `t 0` will be a closed de Bruijn term. Following usual proofs by parametricity, we prove this property for `toTerm` by showing that our implementations of `lam` and `app` preserve R. For `app` this is straightforward. For `lam`, it boils down to showing that for a context of depth i the de Bruijn representation of variables we pass in always gives a well-formed variable in some context of depth j, where $j \geq i + 1$, and in particular $j > 0$. The machinery of Kripke relations always ensures that we know that the context depths always increase as we proceed under binders in the term (see [Atkey 2009a] for more details).

We give a more strongly typed conversion from HOAS to de Bruijn, using the insight from this proof. First we simulate part of the refinement of the type `Int → DBTerm` by the relation R, using a GADT to reflect type-level natural numbers down to the term level:

```
data Nat :: ⋆ → ⋆ where
  NatZ :: Nat Zero
  NatS :: Nat a → Nat (Succ a)

newtype WFDB = WFDB { unWFDB :: ∀j. Nat j → WFTerm j }
```

We do not include the part of the refinement that states that j is greater than some i (although this is possible with GADTs) because the additional type variable this would entail does not appear in the definition of the class UntypedLambda. The advantage of the HOAS representation over the well-formed de Bruijn is that we do not have to explicitly keep track of contexts; the Kripke indexing of our refining relation keeps track of the context for us in the proof.

The little piece of arithmetic $j - (i+1)$ in the toTerm function above must now be represented in a way that demonstrates to the type checker that we have correctly accounted for the indices. The functions natToFin and weaken handle conversion from naturals to inhabitants of the Fin type and injection of members of Fin types into larger ones. The shift function does the actual arithmetic.

```
natToFin :: Nat a → Fin (Succ a)
natToFin NatZ     = FinZ
natToFin (NatS n) = FinS (natToFin n)

weaken :: Fin a → Fin (Succ a)
weaken FinZ     = FinZ
weaken (FinS n) = FinS (weaken n)

shift :: Nat j → Nat i → Fin j
shift NatZ     _        = ⊥
shift (NatS x) NatZ     = natToFin x
shift (NatS x) (NatS y) = weaken $ shift x y
```

By the argument above, the case when the first argument of shift is NatZ will never occur when we invoke it from within the fold over the the HOAS representation, so it is safe to return $\perp$ (i.e. undefined). In any case, there is no non-$\perp$ inhabitant of the type Fin Zero to give here.

The actual code to carry out the conversion is exactly the same as before, except with the arithmetic replaced by the more strongly-typed versions.

```
instance UntypedLambda WFDB where
  lam f = WFDB $
          λi → let v = λj → WFVar (shift j i)
                in
                 WFLam (unWFDB (f (WFDB v)) (NatS i))
  x ‘app‘ y = WFDB $
          λi → WFApp (unWFDB x i) (unWFDB y i)

toWFTerm :: Hoas → WFTerm Zero
toWFTerm v = unWFDB v NatZ
```

The point where Haskell's type system does not provide us with enough information is in the call to shift, where we know from the parametricity proof that $j \geq i+1$ and hence $j > 0$. Moving to a more powerful type system with better support for reasoning about arithmetic, such as Coq [The Coq development team 2009] or Agda [The Agda2 development team 2009], would not help us here. One could easily write a version of the shift function that takes a proof that $j \geq i+1$ as an argument, but we have no way of obtaining a proof of this property without appeal to the parametricity of the HOAS representation. We see two options here for a completely well-typed solution: we could alter the HOAS interface to include information about the current depth of binders in terms, but this would abrogate the advantage of HOAS, which is that contexts are handled by the meta-language; or, we could incorporate parametricity principles into the type system, as has been done previously in Plotkin-Abadi Logic [Plotkin and Abadi 1993] and System R [Abadi et al. 1993]. The second option is complicated by our requirement here for Kripke relations and to use parametricity to prove well-typedness rather than only equalities between terms.

In order to handle open terms we introduce a type of environments WFEnv which takes two type arguments: the type of values and the size of the environment.

```
data WFEnv :: ⋆ → ⋆ → ⋆ where
  WFEmpty  :: WFEnv exp Zero
  WFExtend :: WFEnv exp n → exp → WFEnv exp (Succ n)

lookWF :: WFEnv exp n → Fin n → exp
lookWF (WFExtend _ v)   FinZ     = v
lookWF (WFExtend env _) (FinS n) = lookWF env n
```

Open well-formed HOAS terms with n free variables are defined as functions from well-formed term environments of size n to terms.

```
type WFHoas’ n =
  ∀exp.UntypedLambda exp ⇒ WFEnv exp n → exp
```

Now we can define the translation from well-formed open higher-order abstract syntax terms to well-formed open de Bruijn terms. Whereas toTerm’ had to build an infinite environment mapping free variables to themselves, because the number of free variables did not appear in the type, we now build a finite environment whose length is equal to the number of free variables. We also need to supply the length at the term level using the natural number GADT.

```
toWFTerm’ :: Nat n → WFHoas’ n → WFTerm n
toWFTerm’ n v = unWFDB (v (makeEnv n)) n
  where
    makeEnv :: Nat n → WFEnv WFDB n
    makeEnv NatZ = WFEmpty
    makeEnv (NatS i) =
      WFExtend
        (makeEnv i)
        (WFDB (λj → WFVar (shift j i)))
```

Conversion back from WFTerm to Hoas is straightforward.

```
toWFHoas’ :: WFTerm n → WFHoas’ n
toWFHoas’ (WFVar n) = λenv → lookWF env n
toWFHoas’ (WFLam t) =
  λenv → lam (λx → toWFHoas’ t (WFExtend env x))
toWFHoas’ (WFApp f p) =
  λenv → toWFHoas’ f env ‘app‘ toWFHoas’ p env

toWFHoas :: WFTerm Zero → Hoas
toWFHoas t = toWFHoas’ t WFEmpty
```

The functions toWFTerm and toWFHoas are in fact mutually inverse, and hence the two representations are isomorphic. See Atkey [2009a] for the proof.

3. Language extensions

Having established the main techniques for moving between inductive and higher-order encodings of embedded languages, we now consider a number of extensions.

3.1 More term constructors

We begin by adding boolean terms. As before, we create a type class containing the term formers of our language: constants for true and false, and a construct for conditional branching.

```
class Booleans exp where
  true  :: exp
  false :: exp
  cond  :: exp → exp → exp → exp
```

We do not need to combine this explicitly with UntypedLambda: terms formed from true, false, cond, lam and app may be mingled freely. For example, we can define a function not as follows:

```
not = lam (λx → cond x false true)
```

This receives the following type:

```
not :: (Booleans exp, UntypedLambda exp) ⇒ exp
```

However, for convenience we may wish to give a name to the embedded language that includes both functions and booleans, and we can do so by defining a new class that is a subclass of UntypedLambda and Booleans.

```
class (Booleans exp, UntypedLambda exp) ⇒
  BooleanLambda exp
```

We can now give our definition of not the following more concise type:

```
not :: BooleanLambda exp ⇒ exp
```

In Section 2 we defined a number of functions on untyped λ expressions. We can extend these straightforwardly to our augmented language by defining instances of Booleans. For example, we can extend the size function by defining the following instance:

```
instance Booleans Size where
  true       = Size $ 1
  false      = Size $ 1
  cond c t e = Size $ size c + size t + size e
```

In order to extend the functions for evaluation and conversion to de Bruijn terms we must modify the datatypes used as the domains of those functions. For evaluation we must add constructors for true and false to the Value type.

```
data Value = VFunc (Value → Value) | VTrue | VFalse
```

Then we can extend the evaluation function to booleans by writing an instance of Booleans at type Value.

```
instance Booleans Value where
  true             = VTrue
  false            = VFalse
  cond VTrue  t _ = t
  cond VFalse _ e = e
```

Note that the definitions for both cond and app are now partial, since the embedded language is untyped: there is nothing to prevent programs which attempt to apply a boolean, or use a function as the first argument to cond. In Section 4 we investigate the embedding of typed languages, with total interpreters.

For conversion to well-formed de Bruijn terms we must modify the WFTerm datatype to add constructors for true, false and cond.

```
data WFTerm :: ⋆ → ⋆ where
  WFVar   :: Fin a → WFTerm a
  WFLam   :: WFTerm (Succ a) → WFTerm a
  WFApp   :: WFTerm a → WFTerm a → WFTerm a
  WFTrue  :: WFTerm a
  WFFalse :: WFTerm a
  WFCond :: WFTerm a → WFTerm a → WFTerm a
                                       → WFTerm a
```

Extending the conversion function to booleans is then a simple matter of writing an instance of Booleans at the type WFDB.

```
instance Booleans WFDB where
  true  = WFDB (λi → WFTrue)
  false = WFDB (λi → WFFalse)
  cond c t e  = WFDB (λi → WFCond (unWFDB c i)
                                  (unWFDB t i)
                                  (unWFDB e i))
```

Term formers for integers, pairs, sums, and so on, can be added straightforwardly in the same fashion.

Adding integers is of additional interest in that it allows integration with the standard Num type class. We can extend the Value datatype with an additional constructor for integers, and then use the arithmetic operations of the Num class within terms of the embedded language. For example, the following term defines a binary addition function in the embedded language:

```
lam (λx → lam (λy → x + y))
   :: (UntypedLambda exp, Num exp) ⇒ exp
```

We can, of course, extend evaluation to such terms by defining instances of Num at the Value type; the other functions, such as conversion to the de Bruijn representation, can be extended similarly.

3.2 Conflating levels

The embedded languages we have looked at so far have all maintained a strict separation between the host and embedded levels. A simple example where we mix the levels, which was also used in Atkey [2009a], is a language of arithmetic expressions with a "let" construct and with host language functions contained within terms.

```
class ArithExpr exp where
  let_    :: exp → (exp → exp) → exp
  integer :: Int → exp
  binop   :: (Int → Int → Int) → exp → exp → exp

type AExpr = ∀exp. ArithExpr exp ⇒ exp
```

An example term in this representation is:

```
example8 :: AExpr
example8 = let_ (integer 8) $ λx →
           let_ (integer 9) $ λy →
            binop (+) x y
```

Using the techniques described in Section 2.6, it is clear to see how we can translate this representation to a type of well-formed de Bruijn terms.

The point of this example is to show how function types can be used in two different ways in the HOAS representation. In the let_ operation, functions are used to represent embedded language binding. In the binop operation we use the function type computationally as a host language function. Licata et al. [2008] define a new logical system based on a proof theoretic analysis of focussing to mix the computational and representation function spaces. Using parametric polymorphism, we get the same functionality for free.

3.3 Pattern matching

To this point, we have only considered languages where variables are bound individually. Realistic programming languages feature pattern matching that allows binding of multiple variables at once. It is possible to simulate this by the use of functions as cases in pattern matches, but this gets untidy due to the additional lam constructors required. Also, we may not want to have λ-abstraction in our embedded language. To see how to include pattern matching, we start by considering a language extension with sums and pairs.

We define a type class for introduction forms for pairs and sums:

```
class PairsAndSums exp where
  pair :: exp → exp → exp
  inl  :: exp → exp
  inr  :: exp → exp
```

A simple language extension that allows pattern matching on pairs and sums can be captured with the following type class:

```
class BasicPatternMatch exp where
  pair_match :: exp → ((exp,exp) → exp) → exp
  sum_match  :: exp → (exp → exp) → (exp → exp)
                                              → exp
```

These operations are certainly complete for matching against pairs and sums, but we do not have the flexibility in matching patterns that exists in our host language. To get this flexibility we must abstract over patterns. We represent patterns as containers of kind $\star \to \star$:

```
data Id a          = V a
data Pair f1 f2 a  = f1 a × f2 a
data Inl f a       = Inl (f a)
data Inr f a       = Inr (f a)
```

The HOAS representation of a pattern matching case will take a function of type `f exp → exp`, where we require that `f` is a container constructed from the above constructors. For example, to match against the left-hand component of a sum, which contains a pair, we would use a function like:

```
λ(Inl (V x × V y)) → pair x y)
              :: (Inl (Pair Id Id) exp → exp)
```

Note that when `f` is `Pair`, this will give the same type as the `pair_match` combinator above.

We must be able to restrict to containers generated by the above constructors. We do so by employing the following GADT:

```
data Pattern :: (⋆ → ⋆) → ⋆ → ⋆ where
  PVar  :: Pattern Id (Succ Zero)
  PPair :: Nat x → Pattern f1 x → Pattern f2 y →
                     Pattern (Pair f1 f2) (x :+: y)
  PInl  :: Pattern f x → Pattern (Inl f) x
  PInr  :: Pattern f x → Pattern (Inr f) x
```

The second argument in this GADT records the number of variables in the pattern. This numeric argument will be used to account for the extra context used by the pattern in the de Bruijn representation. The spare-looking `Nat x` argument in `PPair` is used as a witness for constructing proofs of type equalities in the conversion between HOAS and de Bruijn. We define type-level addition by the following type family:

```
type family   n :+: m :: ⋆
type instance Zero     :+: n = n
type instance (Succ n) :+: m = Succ (n :+: m)
```

A HOAS pattern matching case consists of a pattern representation and a function to represent the variables bound in the pattern:

```
data Case exp = ∀f n. Case (Pattern f n) (f exp → exp)
```

A type class defines our pattern matching language extension:

```
class PatternMatch exp where
  match :: exp → [Case exp] → exp
```

This representation is hampered by the need to explicitly describe each pattern before use:

```
matcher0 x = match x
  [ Case (PPair (NatS NatZ) PVar PVar) $
                            λ(V x × V y) → pair x y
  , Case (PInl PVar) $ λ(Inl (V x)) → x ]
```

We get the compiler to do the work for us by using an existential type and a type class:

```
data IPat f = ∀n. IPat (Nat n) (Pattern f n)

class ImplicitPattern f where
  patRep :: IPat f
```

We define instances for each `f` that interests us. The additional `Nat n` argument in `IPat` is used to fill in the `Nat x` argument in the `PPair` constructor. We can now define a combinator that allows convenient expression of pattern matching cases:

```
clause :: ∀f exp.
  ImplicitPattern f ⇒ (f exp → exp) → Case exp
clause body = case patRep of
               IPat _ pattern → Case pattern body
```

This combinator gives a slicker syntax for pattern matching:

```
matcher x = match x
  [ clause $ λ(V x × V y) → pair x y
  , clause $ λ(Inl (V x)) → x ]
```

We can unembed this HOAS representation to guaranteed well-formed de Bruijn terms by a process similar to the one we used above. The de Bruijn representation of pattern match cases consists of a pair of a pattern and a term. In this representation we must explicitly keep track of the context, something that the HOAS representation handles for us.

```
data WFCase a =
  ∀f b. WFCase (Pattern f b) (WFTerm (a :+: b))

data WFTerm :: ⋆ → ⋆ where
  WFVar   :: Fin a → WFTerm a
  WFMatch :: WFTerm a → [WFCase a] → WFTerm a
  WFPair  :: WFTerm a → WFTerm a → WFTerm a
  WFInl   :: WFTerm a → WFTerm a
  WFInr   :: WFTerm a → WFTerm a
  WFLam   :: WFTerm (Succ a) → WFTerm a
  WFApp   :: WFTerm a → WFTerm a → WFTerm a
```

As above, we translate from HOAS to de Bruijn representation by defining a fold over the HOAS term. The case for `match` is:

```
instance PatternMatch WFDB where
  match e cases = WFDB $
    λi → WFMatch (unWFDB e i) (map (doCase i) cases)
      where
        doCase :: ∀i. Nat i → Case WFDB → WFCase i
        doCase i (Case pattern f) =
          let (x, j) = mkPat pattern i
          in  WFCase pattern (unWFDB (f x) j)
```

The helper function used here is `mkPat`, which has type

```
mkPat :: Pattern f n → Nat i → (f WFDB, Nat (i :+: n))
```

This function takes a pattern representation, the current size of the context and returns the appropriate container full of variable representations and the new size of the context. We omit the implementation of this function for want of space. The core of the implementation relies on an idiomatic traversal [McBride and Paterson 2008] of the shape of the pattern, generating the correct variable representations as we go and incrementing the size of the context. To keep track of the size of the context in the types, we use a *parameterised* applicative functor [Cooper et al. 2008], the idiomatic analogue of a parameterised monad [Atkey 2009b]. The term-level representations of natural numbers used in patterns are used to construct witnesses for the proofs of associativity and commutativity of plus, which are required to type this function.

Conversion back again from de Bruijn to HOAS relies on a helper function of the following type:

```
mkEnv :: ∀i exp f j.
         Nat i → WFEnv exp i → Pattern f j →
            f exp → WFEnv exp (i :+: j)
```

This function takes the current size of the context (which can always be deduced from the environment argument), a conversion environment and a pattern representation, and returns a function that maps pattern instances to extended environments. By composing `mkEnv` with the main conversion function from de Bruijn terms, we obtain a conversion function for the de Bruijn representation of pattern matching cases.

4. Unembedding typed languages

We now turn to the representation and unembedding of typed languages, at least when the types of our embedded language is a subset of the types of Haskell. This is mostly an exercise in decorating the constructions of the previous sections with type information, but there is a subtlety involved in representing the types of the embedded language, which we relate in our first subsection.

4.1 Simply-typed λ-calculus, naively

Given the representation of the untyped λ-calculus above, an obvious way to represent a typed language in the manner we have used above is by the following type class, where we decorate all the occurences of `exp` with type variables. This is the representation of typed embedded languages used by Carette et al. [2009].

```
class TypedLambda0 exp where
  tlam0 :: (exp a → exp b) → exp (a → b)
  tapp0 :: exp (a → b) → exp a → exp b
```

Closed simply-typed terms would now be represented by the type:

```
type THoas0 a = ∀exp. TypedLambda0 exp ⇒ exp a
```

and we can apparently go ahead and represent terms in the simply-typed λ-calculus:

```
example3 :: THoas0 (Bool → (Bool → Bool) → Bool)
example3 = tlam0 (λx → tlam0 (λy → y ‘tapp0‘ x))
```

However, there is a hidden problem lurking in this representation. The type machinery that we use to ensure that bound variables are represented correctly may leak into the types that are used in the represented term. We can see this more clearly by writing out the type `TypedLambda0` explicitly as an F_ω type, where the polymorphism is completely explicit:

$$\begin{aligned}\lambda\tau.\forall\alpha : \star \to \star.\ & (\forall\sigma_1\sigma_2.\,(\alpha\,\sigma_1 \to \alpha\,\sigma_2) \to \alpha\,(\sigma_1 \to \sigma_2)) \to \\ & (\forall\sigma_1\sigma_2.\,\alpha\,(\sigma_1 \to \sigma_2) \to \alpha\,\sigma_1 \to \alpha\,\sigma_2) \to \\ & \alpha\,\tau\end{aligned}$$

Now consider a typical term which starts with $\Lambda\alpha.\lambda tlam.tapp....$ and goes on to apply $tlam$ and $tapp$ to construct a representation of a simply-typed λ-calculus term. The problem arises because we have a type constructor α available for use in constructing the represented term. We can instantiate the types σ_1 and σ_2 in the two constructors using α. This will lead to representations of simply-typed λ-calculus terms that contain subterms whose types depend on the result type of the specific fold operation that we perform over terms. Hence, while this representation does not allow "exotic terms", it does allow *exotically typed* terms.

An example of an exotically typed term in this representation is the following:

```
exotic :: ∀exp. TypedLambda0 exp ⇒ exp (Bool → Bool)
exotic = tlam0 (λx → tlam0 (λy → y))
           ‘tapp0‘ (tlam0 (λ(z :: exp (exp Int)) → z))
```

This "represents" the simply typed term:

$$(\lambda x^{exp(Int)\to exp(Int)}.\lambda y^{Bool}.y)(\lambda z^{exp(Int)}.z)$$

When we write a fold over the representation `exotic`, we will instantiate the type `exp` with the type we are using for accumulation. Thus the term `exotic` will technically represent different simply-typed terms for different folds.

This confusion between host and embedded language types manifests itself in the failure of the proof of an isomorphism between this church encoding of typed HOAS and the de Bruijn representation. After the conversion of `exotic` to de Bruijn, we will have a representation of the simply typed term:

$$(\lambda x^{TDB(Int)\to TDB(Int)}.\lambda y^{Bool}.y)(\lambda z^{TDB(Int)}.z)$$

where the placeholder `exp` has been replaced by the type constructor `TDB` used in the conversion to de Bruijn. Converting this term back to typed HOAS preserves this constructor, giving a term that differs in its types to the original term.

An interesting question to ask is: exactly what is being represented by the type `THoas0`, if it is not just the simply-typed terms? We currently have no answer to this. Maybe we are representing terms with the term syntax of the simply-typed λ-calculus, but the types of Haskell. On the other hand, the fact that the quantified constructor `exp` used in the representation will change according to the type of the fold that we perform over represented terms is troubling.

Note that, due to the fact that the type variable `a`, which represents the type of the whole term, appears outside the scope of `exp` in the type `THoas0`, we can never get terms that are exotically typed at the top level; only subterms with types that do not contribute to the top-level type may be exotically typed, as in the `exotic` example above.

Aside from the theoretical problem, there is a point about which type system our embedded language should be able to have. If we are going to unembed an embedded language effectively, then we should be able to get our hands on representations of object-level types. Moreover, many intensional analyses that we may wish to perform are type-directed, so explicit knowledge of the embedded language types involved is required. To do this we cannot straightforwardly piggy-back off Haskell's type system (though we are forced to rely on it to represent object-level types, by the stratification between types and terms in Haskell's type theory). To fix this problem, we define explicit representations for embedded language types in the next subsection.

4.2 The closed kind of simple types

We define a GADT `Rep` for representing simple types and hence precluding exotic types. This connects a term-level representation of simple types with a type-level representation of types (in which the underlying types are Haskell types). Explicitly writing type representations everywhere would be tedious, so we follow Cheney and Hinze [2002] and define the type class `Representable` of simple types. This allows the compiler to infer and propagate many type representations for us.

```
data Rep :: ⋆ → ⋆ where
   Bool  :: Rep Bool
   (:→) :: (Representable a, Representable b) ⇒
         Rep a → Rep b → Rep (a→b)

class Representable a where rep :: Rep a

instance Representable Bool where rep = Bool

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

Note that the leaves of a `Rep` must be `Bool` constructors, and so it is only possible to build representations of simple types. The restriction to simple types is made more explicit with the `Representable` type class. In effect `Representable` is the closed kind of simple types.

A key function that we can define against values of type `Rep` is the conditional cast operator, which has type:

```
cast :: Rep a → Rep b → Maybe ( ∀f. f a → f b)
```

We omit the implementation of this function to save space. The basic implementation idea is given by Weirich [2004].

4.3 Simply-typed λ-calculus, wisely

The type class for simply-typed lambda terms is just like the naive one we gave above, except that the constructors are now augmented with type representations.

```
class TypedLambda exp where
  tlam :: (Representable a, Representable b) ⇒
```

```
    (exp a → exp b) → exp (a → b)
  tapp :: (Representable a, Representable b) ⇒
    exp (a → b) → exp a → exp b

type THoas a = ∀exp. TypedLambda exp ⇒ exp a
```

Although the `Representable` type class restricts `THoas` terms to simple types, we can still assign a `THoas` term a polymorphic type.

```
example4 :: (Representable a, Representable b) ⇒
              THoas ((a → b) → a → b)
example4 = tlam (λx → tlam (λy → x `tapp` y))
```

Of course, this polymorphism is only at the meta level; we are in fact defining a family of typing derivations of simply-typed terms. We can instantiate `example4` many times with different simple types for a and b. However, if we wish to unembed it (using the function `toTTerm` that we define below) then we must pick a specific type by supplying an explicit type annotation.

```
example5 =
 toTTerm (example4 :: THoas ((Bool→Bool)→Bool→Bool))
```

Sometimes the compiler will not be able to infer the types that we need in terms. This happens when a subterm contains a type that does not contribute to the top-level type of the term. These are also the situations in which exotically typed terms arise. For example, the declaration

```
example6 :: (Representable a) ⇒ THoas (a → a)
example6 = tlam (λx → tlam (λy → y))
                                      `tapp` tlam (λz→ z)
```

causes GHC to complain that there is an ambiguous type variable arising from the third use of `tlam`. We must fix the type of `z` to some concrete simple type in order for this to be a proper representation. It is possible to do this by using type ascriptions at the Haskell level, but it is simpler to do so by defining a combinator that takes an explicit type representation as an argument:

```
tlam' ::
  (Representable a, Representable b, TypedLambda exp) ⇒
    Rep a → (exp a → exp b) → exp (a → b)
tlam' _ = tlam
```

The term can now be accepted by the Haskell type checker by fixing the embedded language type of `z`:

```
example7 :: (Representable a) ⇒ THoas (a → a)
example7 = tlam (λx → tlam (λy → y))
            `tapp` (tlam' Bool (λz → z))
```

Defining an evaluator for these terms is now straightforward. We can simply interpret each embedded language type by its host language counterpart:

```
newtype TEval a = TEval { unTEval :: a }
```

The instance of `TypedLambda` for `TEval` is straightforward:

```
instance TypedLambda TEval where
  tlam f                   = TEval (unTEval ∘ f ∘ TEval)
  TEval f `tapp` TEval a = TEval (f a)

teval :: THoas a → a
teval t = unTEval t
```

We note that the HOAS representation is usually very convenient for defining evaluators. In particular, this representation frees us from keeping track of environments. Also, note that exotically typed terms do not prevent us from writing an evaluator. If evaluation is all one wants to do with embedded terms, then restricting terms to a subset of types is not required.

4.4 Translating to de Bruijn and back

Where we used the natural numbers GADT to record the depth of a context in the representation of well-formed de Bruijn terms, we now need to include the list of types of the variables in that context. At the type level, we use the unit type to represent the empty context, and pair types to represent a context extended by an additional type. At the term level, we maintain a list of (implicit) type representations:

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

The simply-typed analogue of the `Fin` GADT is the GADT `Index`. At the type level this encodes a pair of a type list and the type of a distinguished element in that list; at the term level it encodes the index of that element.

```
data Index :: ⋆ → ⋆ → ⋆ where
  IndexZ :: Index (a, ctx) a
  IndexS :: Index ctx a → Index (b, ctx) a
```

The type constructor `TTerm` for simply-typed de Bruijn terms takes two parameters: the first is a type list encoding the types of the free variables, and the second is the type of the term itself.

```
data TTerm :: ⋆ → ⋆ → ⋆ where
  TVar :: Representable a ⇒ Index ctx a → TTerm ctx a
  TLam :: (Representable a, Representable b) ⇒
            TTerm (a, ctx) b → TTerm ctx (a → b)
  TApp :: (Representable a, Representable b) ⇒
         TTerm ctx (a→b) → TTerm ctx a → TTerm ctx b
```

The translation to de Bruijn terms is similar to that for well-formed untyped terms. We again give the basic fold over the HOAS term representation as an instance of the `TypedLambda` class:

```
newtype TDB a =
  TDB { unTDB :: ∀ctx. Ctx ctx → TTerm ctx a }

instance TypedLambda TDB where
  tlam (f::TDB a → TDB b) =
    TDB$ λi→ let v = λj → TVar (tshift j (CtxS i))
              in TLam (unTDB (f (TDB v)) (CtxS i))
  (TDB x) `tapp` (TDB y) = TDB$ λi → TApp (x i) (y i)
```

The key difference is in the replacement of the `shift` function that computes the de Bruijn index for the bound variable by the type-aware version `tshift`. To explain the `tshift` function, we re-examine the proof that this fold always produces well-formed de Bruijn terms. In the untyped case, the proof relies on Kripke relations indexed by natural numbers, where the natural number records the depth of the context. Now that we also have types to worry about, we use relations indexed by lists of embedded language types, ordered by the standard prefix ordering; we define $R^{\Gamma}_{\sigma}(\mathtt{t}) \Leftrightarrow \forall\Gamma' \geq \Gamma.\Gamma' \vdash (\mathtt{t}\ \Gamma') : \sigma$, where $\Gamma \vdash t : \sigma$ is the typing judgement of the simply-typed λ-calculus.

In the case for `tlam`, we again have two contexts `i` and `j`, where `i` is the context surrounding the λ-abstraction, and `j` is the context surrounding the bound variable occurence. By a parametricity argument, and the way in which we have defined our Kripke relation, we know that `(a, i)` will always be a prefix of `j`, and so we obtain a well-formed de Bruijn index by computing the difference between the depths of the contexts. We implement this by the following functions:

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

tshift' :: Int → Ctx j → Ctx (a, i) → Index j a
```

```
tshift' _ CtxZ       _        = ⊥
tshift' 0 (CtxS _) (CtxS _) =
                    fromJust (cast rep rep) IndexZ
tshift' n (CtxS c1) c2      =
                    IndexS (tshift' (n-1) c1 c2)

tshift :: Ctx j → Ctx (a, i) → Index j a
tshift c1 c2 = tshift' (len c1 - len c2) c1 c2
```

As with the untyped case, we have had to feed the Haskell type checker with bottoms to represent cases that can never occur. Firstly, the case when j is shorter than (a,i) can never happen, as with the untyped version. Secondly, we use a well-typed cast to show that the type a does occur in j at the point we think it should. Given that we know the cast will succeed, it would likely be more efficient to simply replace the cast with a call to unsafeCoerce. We chose not to here because we wanted to see how far we could push the type system.

Were we to use the representation given by the type THoas0, which allows exotically typed terms, it would still be possible to write a conversion to de Bruijn representation, but it would be necessary to replace the use of cast in tshift' with unsafeCoerce, since we do not have any type representations to check. Also, the de Bruijn representation would not be able to contain any Representable typeclass constraints, meaning that we could not write intensional analyses that depend on the types of embedded-language terms.

In order to be able to define the type of open simply-typed HOAS we need to define a GADT for environments.

```
data TEnv :: (⋆ → ⋆) → ⋆ → ⋆ where
  TEmpty :: TEnv exp ()
  TExtend :: TEnv exp ctx → exp a → TEnv exp (a, ctx)

lookT :: TEnv exp ctx → Index ctx a → exp a
lookT (TExtend _   v) IndexZ     = v
lookT (TExtend env _) (IndexS n) = lookT env n
```

Now we can define a type for open simply-typed HOAS terms.

```
type THoas' ctx a = ∀(exp :: ⋆ → ⋆).
  TypedLambda exp ⇒ TEnv exp ctx → exp a
```

The translations between HOAS and de Bruijn representations and vice-versa fall out naturally.

```
toTHoas' :: TTerm ctx a → THoas' ctx a
toTHoas' (TVar n)   = λenv → lookT env n
toTHoas' (TLam t)   =
  λenv → tlam (λx → toTHoas' t (TExtend env x))
toTHoas' (TApp f p) =
  λenv → toTHoas' f env `tapp` toTHoas' p env

toTHoas :: TTerm () a → THoas a
toTHoas t = toTHoas' t TEmpty

toTTerm' :: Ctx ctx → THoas' ctx a → TTerm ctx a
toTTerm' ctx v = unTDB w ctx
  where  w = v (makeEnv ctx)
         makeEnv :: Ctx ctx → TEnv TDB ctx
         makeEnv CtxZ = TEmpty
         makeEnv (CtxS j) =
           TExtend (makeEnv j)
             (TDB (λi → TVar (tshift i (CtxS j))))

toTTerm :: THoas a → TTerm () a
toTTerm v = unTDB v CtxZ
```

5. Examples

We give two examples where unembedding plays an essential role.

5.1 Mobile code

Our first example involves sending programs of an embedded language over a network to be executed at some remote location. In order to make the programs a little more useful than pure lambda terms we extend the embedding of typed λ calculus given in Section 4.3 to include constructors and destructors for booleans. We define the TypedBooleans class independently of TypedLambda, and define a new class, Mobile, for the language formed by combining the two.

```
class TypedBooleans exp where
  ttrue  :: exp Bool
  tfalse :: exp Bool
  tcond  ::
    Representable a ⇒
      exp Bool → exp a → exp a → exp a

class (TypedBooleans exp, TypedLambda exp) ⇒ Mobile exp
```

Next, we define concrete representations for types and terms, together with automatically-derived parsers and printers.

```
data URep = UBool | URep -u-> URep deriving (Show, Read)

data MTerm = MVar Int
           | MLam URep MTerm | MApp MTerm MTerm
           | MTrue | MFalse | MCond MTerm MTerm MTerm
             deriving (Show, Read)
```

Section 2 showed how to unembed untyped HOAS terms to untyped de Bruijn terms; obtaining untyped de Bruijn terms from typed terms is broadly similar. The type MDB is analogous to DB (Section 2.3), but the phantom parameter discards type information.

```
newtype MDB a = MDB { unMDB :: Int → MTerm }
```

Defining instances of Mobile and its superclasses for MDB gives a translation to MTerm; composing this translation with show gives us a marshalling function for Mobile. (In an actual program it would, of course, be preferable to use a more efficient marshalling scheme.) We omit the details of the translation, which follow the pattern seen in Section 2.3.

```
marshal :: (∀exp. Mobile exp ⇒ exp a) → String
marshal t = show (unMDB t 0)
```

Erasing types during marshalling is comparatively straightforward; reconstructing types is more involved. We begin with a definition, Typed, that pairs a term with a representation of its type, hiding the type variable that carries the type information.

```
data Typed :: (⋆ → ⋆) → ⋆ where
  (:::) :: Representable a ⇒ exp a → Rep a → Typed exp
```

We use Typed to write a function that re-embeds MTerm values as typed HOAS terms. The function toHoas takes an untyped term and an environment of typed terms for the free variables; it returns a typed term. Since type checking may fail — the term may refer to variables not present in the environment, or may be untypeable — the function is partial, as indicated by the Maybe in the return type.

```
toHoas :: (TypedLambda exp, TypedBooleans exp) ⇒
  MTerm → [Typed exp] → Maybe (Typed exp)
```

We omit the implementation, but the general techniques for reconstructing typed terms from untyped representations are well-known: see, for example, work by Baars and Swierstra [2002]. Composing toHoas with the parser for MTerm gives an unmarshalling function for closed terms.

```
unmarshal :: String →
  (∀exp. Mobile exp ⇒ Maybe (Typed exp))
unmarshal s = toHoas (read s) []
```

Combined with an evaluator for terms as defined in Section 4.3, `marshal` and `unmarshal` allow us to construct HOAS terms, send them over a network, and evaluate them on another host.

5.2 Nested relational calculus

Our second example is based on the Nested Relational Calculus (NRC) [Tannen et al. 1992]. NRC is a query language based on comprehensions, with terms for functions, pairs, unit, booleans and sets. As the name suggests, NRC permits nested queries, unlike SQL, which restricts the type of queries to a collection of records of base type. However, there are translations from suitably-typed NRC terms to flat queries [Cooper 2009, Grust et al. 2009]. The specification of these translations involves intensional analysis; it is therefore easier to define them on a concrete representation of terms than as a mapping from higher-order abstract syntax.

Once again we can reuse the embeddings presented in earlier sections. We combine the `TypedLambda` and `TypedBoolean` languages of Sections 4.3 and 5.1 with embeddings of term formers for pairs, units and sets; these are straightforward, so we give only the case for sets as an example. There are four term formers, for empty and singleton sets, set union, and comprehension; this last uses Haskell's binding to bind the variable, in standard HOAS style.

```
class TypedSets exp where
  empty  :: Representable a ⇒
            exp (Set a)
  single :: Representable a ⇒
            exp a → exp (Set a)
  union  :: Representable a ⇒
            exp (Set a) → exp (Set a) → exp (Set a)
  for    :: (Representable a, Representable b) ⇒
       exp (Set a) → (exp a→exp (Set b)) → exp (Set b)

class (TypedLambda exp, TypedBooleans exp,
       TypedUnit exp,   TypedPairs exp,
                        TypedSets exp) ⇒ NRC exp
```

We must also extend the `Rep` datatype and `Representable` class to include the new types.

```
data Rep :: ⋆ → ⋆ where
  ...
  Set :: Representable a ⇒ Rep a → Rep (Set a)

instance Representable a ⇒ Representable (Set a) where
  rep = Set rep
```

Using the techniques presented in earlier sections, we can unembed terms of `NRC` to obtain a concrete representation on which translations to a flat calculus can be defined. The term formers of the language ensure that embedded terms are correctly typed; we can also assign a type to the translation function that restricts its input to queries that can be translated to a flat query language such as SQL. Given these guarantees, we are free to dispense with types in the concrete representation used internally, making it easier to write the translation of interest.

The combination of a carefully-typed external interface and an untyped core is used in a number of embedded languages; for example, by Leijen and Meijer [1999] for SQL queries and by Lindley [2008] for statically-typed XHTML contexts. Our presentation here has the additional property that the external language (based on HOAS) is more convenient for the user than the internal language (de Bruijn terms), while the internal language is more convenient for analysis.

6. Related work

The idea of encoding syntax with binding using the host language's binding constructs goes back to Church [1940]. As far as we are aware Coquand and Huet [1985] were the first to remark that the syntax of untyped lambda-calculus can be encoded using the universally quantified type:

$$\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

Pfenning and Elliott [1988] proposed higher-order abstract syntax as a general means for encoding name binding using the meta language. Washburn and Weirich [2008] also present essentially this type and show how functions can be defined over the syntax by means of folds.

Programming with explicit folds is awkward. Carette et al. [2009] give a comprehensive account of how to achieve the same effect using Haskell type classes or ML modules. Our work is in the same vein. Where Carette et al concentrate on implementing different compositional interpretations of HOAS our main focus is on unembedding to a first-order syntax in order to allow intensional analyses. Hofer et al. [2008] apply Carette et al's techniques in the context of Scala. As they remark, many standard optimisations one wants to perform in a compiler are difficult to define compositionally. Our unembedding provides a solution to this problem. Hofer et al also discuss composing languages in a similar way to us. Their setting is somewhat complicated by the object-oriented features of Scala.

Meijer and Hutton [1995] and Fegaras and Sheard [1996] show how to define folds or *catamorphisms* for data types with embedded functions. As we discussed in Section 2.1, the data type that Fegaras and Sheard use to represent terms does not use parametricity to disallow exotic terms, and so does not allow an unembedding function to be defined. Fegaras and Sheard also use HOAS to represent cyclic data structures and graphs, essentially by encoding then using explicit sharing via a `let` construct and recursion using a `fix` construct. Ghani et al. [2006] represent cyclic data structures using a de Bruijn representation in nested datatypes. Our unembedding process gives a translation from Fegaras and Sheard's HOAS representation to the Ghani et al.'s de Bruijn representation.

Pientka [2008] introduces a sophisticated type system that provides direct support for recursion over HOAS datatypes. In contrast, our approach supports recursion over HOAS datatypes within the standard Haskell type system. There is a similarity between our representation of open simply-typed terms using HOAS and hers, but we must leave a detailed comparison to future work.

Elliott et al. [2003] give an in-depth account of how to compile domain-specific embedded languages, but they do not treat HOAS.

Rhiger [2003] details an interpretation of simply-typed HOAS as an inductive datatype. His work differs from ours in that he only considers a single interpretation and he relies on a single global abstract type to disallow exotic terms and to ensure that the target terms are well-typed.

In their work on implementing type-preserving compilers in Haskell, Guillemette and Monnier [2007, 2008] mention conversion of HOAS to a de Bruijn representation. Their implementation sounds similar to ours, but they do not spell out the details. They do not mention the need to restrict the type representations in the embedded language. Their work does provide a good example of an intensional analysis—closure conversion—that would be difficult to express as a fold over the HOAS representation.

Pfenning and Lee [1991] examine the question of embedding a polymorphic language within F_ω, with a view to defining a well-typed evaluator function. They use a nearly-HOAS representation with parametricity, where λ-abstraction case is represented by a constructor with type $\forall\alpha\beta.(\alpha \to \mathtt{exp}\ \beta) \to \mathtt{exp}\ (\alpha \to \beta)$. Hence they do not disallow exotic terms. They are slightly more ambitious in that they attempt to embed a polymorphic language, something that we have not considered here. Guillemette and Monnier [2008] embed a polymorphic language using HOAS, but they resort to using de Bruijn indices to represent type variables, which makes the embedding less usable.

Oliveira et al. [2006] investigate modularity in the context of generic programming. Our use of type classes to give modular extensions of embedded DSLs is essentially the same as their encoding of extensible generic functions.

Our unembedding translations are reminiscent of normalisation by evaluation (NBE) [Berger et al. 1998]. The idea of NBE is to obtain normal forms by first interpreting terms in some model and then defining a *reify* function mapping values in the model back to normal forms. The key is to choose a model that includes enough syntactic hooks in order to be able to define the *reify* function. In fact our unembeddings can be seen as degenerate cases of NBE. HOAS is a model of α-conversion and the *reify* function is given by the `DB` instance of the `UntypedLambda` type class.

Acknowledgements Atkey is supported by grant EP/G006032/1 from EPSRC. We would like to thank the anonymous reviewers for helpful comments, and Bruno Oliveira for pointing us to related work.

References

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *POPL*, pages 157–170, 1993.

Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

Robert 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, 2009a.

Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3 & 4):355–376, 2009b.

Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage dsl embedded in Haskell. In *ICFP*, pages 225–228, 2008.

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02*, pages 157–166, New York, NY, USA, 2002. ACM.

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio V. Russo. Shrinking reductions in SML.NET. In *IFL*, pages 142–159, 2004.

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalisation by evaluation. In *Prospects for Hardware Foundations*, 1998.

Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated. *Journal of Functional Programming*, 2009. To appear.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02*, New York, NY, USA, 2002. ACM.

Alonso Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

Ezra Cooper. The script-writer's dream: How to write great sql in your own language, and be sure it will succeed. In *DBPL*, 2009. To appear.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *APLAS*, December 2008.

Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra (1)*, pages 151–184, 1985.

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

Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL*, pages 284–294, 1996.

N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In H. Nilsson, editor, *Proc. of 7th Symp. on Trends in Functional Programming, TFP 2006 (Nottingham, Apr. 2006)*, 2006.

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *SIGMOD 2009*, Providence, Rhode Island, June 2009. To appear.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in Haskell. In *Haskell*, pages 83–92, 2007.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *ICFP*, pages 75–86, 2008.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *GPCE*, pages 137–148, 2008.

Andrew Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

Marco Kuhlmann. Functional metapost for latex, 2001.

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL'99*, pages 109–122, Austin, Texas, October 1999.

Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on Binding and Computation. In *LICS*, pages 241–252, 2008.

Sam Lindley. Many holes in Hindley-Milner. In *ML '08*, 2008.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2009. URL http://coq.inria.fr. Version 8.2.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA*, pages 324–333, 1995.

Bruno Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, pages 199–216, 2006.

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ICFP '00*, pages 280–292, New York, NY, USA, 2000. ACM.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI*, pages 199–208, 1988.

Frank Pfenning and Peter Lee. Metacircularity in the polymorphic lambda-calculus. *Theor. Comput. Sci.*, 89(1):137–159, 1991.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*, pages 371–382, 2008.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993. ISBN 3-540-56517-5.

John C Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.

Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005.

Val Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *ICDT '92*, pages 140–154. Springer-Verlag, 1992.

The Agda2 development team. The agda2 website. http://wiki.portal.chalmers.se/agda/, 2009.

Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *PADL*, pages 192–208, 2002.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(1):87–140, 2008.

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