Embedding Polymorphic Dynamic Typing

Thomas van Noort Wouter Swierstra Peter Achten Rinus Plasmeijer

Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL, Nijmegen, The Netherlands

{thomas, w.swierstra, p.achten, rinus}@cs.ru.nl

Abstract

Dynamic typing in a statically typed functional language allows us to defer type unification until run time. This is typically useful when interacting with the 'outside' world where the type of values involved may not be known statically. Haskell has minimal support for dynamic typing, it only supports monomorphism. Clean, on the other hand, has a more rich and mature dynamic typing system where polymorphism is supported as well. An interesting difference is that Haskell offers monomorphic dynamic typing via a library, while Clean offers polymorphic dynamic typing via built-in language support. In the Clean approach there is a great deal of freedom in the implementation in the compiler since the dynamic typing system is defined on abstract syntax trees, whereas the Haskell approach does not need to extend the core language and hence reduces the complexity of the language and compiler. In this paper we investigate what it takes for a functional language to embed polymorphic dynamic typing. We explore an embedding in Haskell using generalised algebraic datatypes and argue that a universe for the representation of types needs to separated from its interpretation as a type. We motivate the need for a dependently-typed functional language like Agda and perform the embedding using structural equality on type representations. Finally, we extend this approach with an instance-of algorithm and define a framework for the corresponding cast function.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords dependently-typed programming, dynamic typing, generalised algebraic datatypes, instantiation, polymorphism, unification, universe construction

1. Introduction

Dynamic typing in a statically typed functional language such as Clean and Haskell allows us to defer type unification until run time. This is typically useful when interacting with the 'outside' world: when values are exchanged between applications by deserialisation from disk, input is provided by a user, or when values are obtained via a network connection. In such situations, the types of the values at hand may not be known until run time and type checking is deferred. Values and functions are wrapped together with a representation of their type in a uniform black box, as their type is statically not known. Such a value is unwrapped by pattern matching and casting to a specific type. Although type unification can fail at run time when a dynamic value presents an unexpected type, the static type system guarantees that when pattern matching succeeds, the unwrapped value can be used safely. Hence, the advantages of a static typing system are not compromised when crossing the boundary to a dynamically typed world.

Haskell has minimal support for dynamic typing, it only supports monomorphism (Baars and Swierstra, 2002; Cheney and Hinze, 2002). The *de facto* Haskell compiler GHC includes a library function $toDyn$ to wrap any monomorphic value, such as an increment function on integers[1], in a dynamic:

$$\begin{aligned} &incDyn :: Dynamic \\ &incDyn = toDyn\ (\lambda x \to x + 1) \end{aligned}$$

Then, a value is unwrapped using the library function $fromDyn$ which performs a cast, where the required type is specified by the context in which it is unwrapped:

$$\begin{aligned} &inc :: Maybe\ (Int \to Int) \\ &inc = fromDyn\ incDyn \end{aligned}$$

Clean, on the other hand, has a more rich and mature dynamic typing system that is built in; it adopted ML's support for monomorphism (Abadi et al., 1991; Pil, 1997) and polymorphism (Abadi et al., 1994; Leroy and Mauny, 1993). Having such an extensive dynamic typing system does not only improve orthogonality with the static typing system (Van Noort et al., 2010), but also has important applications (Plasmeijer and Van Weelden, 2005; Plasmeijer et al., 2011). In Clean, we wrap a value, for example the polymorphic identity function, in a dynamic by using the corresponding keyword:

$$\begin{aligned} &idDyn :: Dynamic \\ &idDyn = \mathbf{dynamic}\ (\lambda x \to x) \end{aligned}$$

Then, we unwrap such a value by pattern matching and providing a required type using the :: annotation:

$$\begin{aligned} &id :: Maybe\ (A.a : a \to a) \\ &id = \\ &\quad \mathbf{case}\ idDyn\ \mathbf{of} \\ &\qquad (f :: A.a : a \to a) \to Just\ f \\ &\qquad \_ \qquad\qquad\quad \to Nothing \end{aligned}$$

It is important to observe that the required type does not need to be structurally equal to the type found in the dynamic; it is allowed to be more specific than the type given. Thus, we can instantiate the

[1] We ignore that in Haskell the $(+)$ operator has an ad-hoc polymorphic type and assume it to have the monomorphic type $Int \to Int \to Int$.

WGP'11, September 18, 2011, Tokyo, Japan.

type that is contained with the value in the dynamic. For example, assume we require the result to be a function of type $Int \to Int$:

```
idInt :: Maybe (Int → Int)
idInt =
  case idDyn of
    (f :: Int → Int) → Just f
    _                → Nothing
```

Here, the required type is unified with the type of the value in the dynamic and when this succeeds, the value is implicitly coerced to the required type and returned.

An interesting difference between the approaches in the two languages is that Haskell offers monomorphic dynamic typing via a library, while Clean offers a more expressive system with support for polymorphism via built-in language support. The advantage of the Clean approach is that there is a great deal of freedom in the implementation in the compiler since the dynamic typing system is defined on abstract syntax trees of expressions and types. On the other hand, the Haskell approach does not need to extend the core language and hence reduces the complexity of the language and compiler. Embedding it as a library also serves as a nice demonstration of the expressivity of a language.

In this paper we investigate what it takes for a functional language to embed polymorphic dynamic typing. We limit our scope to a system with predicative polymorphism. That is, bound variables can only be instantiated by base types without variables. Concretely, our contributions are the following:

- We show how to embed monomorphic dynamic typing in Haskell using generalised algebraic datatypes. We explore a straightforward extension of this approach to polymorphic dynamic typing and identify the difficulties with doing so, limiting ourselves to structural equality of type representations instead of their unification (Section 2).
- We motivate the need for a dependently-typed functional language like Agda and define the embedding of polymorphic dynamic typing using structural equality on type representations (Section 3).
- We extend this approach with an instance-of algorithm on type representations (Section 4), based on earlier work on first-order unification by McBride (2003).
- We construct a framework for the cast function that performs the unwrapping (Section 5), which turns out be surprisingly intricate. Although our implementation is not yet complete since we require two postulated lemmas, we have exactly identified the steps to embed polymorphic dynamic typing in Agda.

Finally, we discuss related work (Section 6) and conclude with a brief discussion and future work (Section 7).

2. Embedding in a functional language

Before we jump into the deep end, we first consider the embedding of monomorphic dynamic typing in Haskell using generalised algebraic datatypes (Section 2.1). Then, we explore a straightforward extension of this approach to polymorphic dynamic typing with structural equality of type representations (Section 2.2) and identify the difficulties with doing so (Section 2.3).

2.1 Monomorphic dynamic typing

As alluded to in the introduction, a dynamic value constitutes a value and a representation of its type. Hence, for us to describe a dynamic value, we first need a datatype that describes types. A naive approach is to define a universe for the representation of types as a vanilla datatype:

```
data U = INT
       | PAIR U U
       | U :⇒: U
```

The universe describes integer, pair, and function types respectively. Then, a dynamic value is defined as follows:

```
data Dyn = forall a . Dyn U a
```

A dynamic value is a black box, hence, the type of the value contained is existentially quantified[2]. The main problem with this approach already becomes apparent. How is it reflected that the value of type U represents the type a? This becomes even more clear when we write down the type of the function that casts a dynamic to a required type:

```
cast :: U → Dyn → Maybe a
```

Again, the relation between the resulting value of this cast function and the required type is missing.

In Haskell, generalised algebraic datatypes (Peyton Jones et al., 2006) provide a solution. We define U again, but now include a type parameter that reflects the type that the universe represents:

```
data U :: * → * where
  INT   :: U Int
  PAIR  :: U a → U b → U (a, b)
  (:⇒:) :: U a → U b → U (a → b)
```

A value of type $U\ a$ reflects the type a that it represents. Hence, when we define Dyn again, the type of the value is visible in the representation that is contained in the dynamic value:

```
data Dyn = forall a . Dyn (U a) a
```

To be able to unwrap the existentially quantified value from the dynamic, we need to compare the contained representation with a representation and then prove that these reflect the same type. This proof of equality is defined by a generalised algebraic datatype, stating that both type parameters are equal:

```
data (:≡:) :: * → * → * where
  Refl :: a :≡: a
```

The function that decides if two representations reflect equal types uses structural recursion on the arguments in parallel, collecting proofs along the way:

```
decU :: U a → U b → Maybe (a :≡: b)
decU INT          INT          = Just Refl
decU (PAIR u1 u1') (PAIR u2 u2') =
  case (decU u1 u2, decU u1' u2') of
    (Just Refl, Just Refl) → Just Refl
    _                      → Nothing
decU (u1 :⇒: u1')  (u2 :⇒: u2')  =
  case (decU u1 u2, decU u1' u2') of
    (Just Refl, Just Refl) → Just Refl
    _                      → Nothing
decU _             _             = Nothing
```

Note that we need to explicitly pattern match on the $Refl$ constructor obtained from recursion to actually deploy the proof.

Using this function, we are able to define the cast function which performs the unwrapping:

```
cast :: U a → Dyn → Maybe a
cast u1 (Dyn u2 x) =
  case decU u1 u2 of
    Just Refl → Just x
    Nothing   → Nothing
```

[2] Ironically, this is denoted in Haskell using the **forall** keyword.

When the required representation reflects the same type as the representation in the dynamic value, the proof $Refl$ tells us that we can safely return the value that is contained in the dynamic.

2.2 Polymorphic dynamic typing

Generalised algebraic datatypes allow us to attach an actual type to a representation of a monomorphic type. But can we also use this approach to represent polymorphic types? This requires a way to bind and reference variables. Typically, this is achieved either by using De Bruijn indices (De Bruijn, 1972) or through higher-order abstract syntax (Pfenning and Elliot, 1988).

2.2.1 De Bruijn indices

Before we can reference variables in the representation of a type, we first need a way to bind variables. We introduce another generalised algebraic datatype that allows us to introduce quantifiers to bind variables:

$$
\begin{aligned}
&\mathbf{data}\ V :: * \to * \to * \ \mathbf{where}\\
&\quad BASE\ \ \ :: U\ a\ env \to V\ a\ env\\
&\quad FORALL :: V\ a\ (U\ b\ (), env) \to V\ a\ env
\end{aligned}
$$

A universe must reflect the type that it represents. Since we need occurrences of the same variable to reflect the same type, we have to carry around additional administration. We use an environment to keep track of which indices map to which variables. The constructor $FORALL$ that introduces a variable extends this environment. Since we are talking about quantification over representations of types, the environment is not extended with a type b but with a representation $U\ b\ ()$. We enforce predicativity of the embedding by using U instead of V and requiring this representation to be closed and not use any variables by passing the unit type for the environment. A base universe, similar to the one defined in Section 2.1, that includes references to variables in an environment is defined as follows:

$$
\begin{aligned}
&\mathbf{data}\ U :: * \to * \to * \ \mathbf{where}\\
&\quad INT\ \ :: U\ Int\ env\\
&\quad PAIR :: U\ a\ env \to U\ b\ env \to U\ (a, b)\ env\\
&\quad (:\Rightarrow:)\ :: U\ a\ env \to U\ b\ env \to U\ (a \to b)\ env\\
&\quad VAR\ :: Ref\ (U\ a\ ())\ env \to U\ a\ env
\end{aligned}
$$

Here, we see that the type that a variable occurrence represents is obtained from the environment that is passed along. We use De Bruijn indices to reference variables in a typed environment in the style of Pasălić and Linger (2004):

$$
\begin{aligned}
&\mathbf{data}\ Ref :: * \to * \to * \ \mathbf{where}\\
&\quad Rz :: Ref\ a\ (a, env)\\
&\quad Rs :: Ref\ a\ env \to Ref\ a\ (b, env)
\end{aligned}
$$

A value of type $Ref\ a\ env$ references a value of type a in the environment env. The Rz constructor references the top variable in the environment, the Rs constructor states that the variable is found in the tail of the environment. Using these definitions of the universes U and V, for instance, the type $\mathbf{forall}\ a\ .\ a \to a$ of the polymorphic identity function is represented by the value $FORALL\ (BASE\ (VAR\ Rz :\Rightarrow: VAR\ Rz))$.

Then, a dynamic no longer carries a representation U, but a closed representation V:

$$
\mathbf{data}\ Dyn = \mathbf{forall}\ a\ .\ Dyn\ (V\ a\ ())\ a
$$

Again, to be able to unwrap the value of the dynamic, we need a function that decides if two representations reflect equal types. But now the representations can include references to variables in an environment:

$$
\begin{aligned}
&decV :: V\ a\ env \to V\ b\ env' \to Maybe\ (a :\equiv: b)\\
&decV\ (BASE\ u1)\ \ \ (BASE\ u2)\ \ \ =\\
&\quad \mathbf{case}\ decU\ u1\ u2\ \mathbf{of}\\
&\qquad Just\ Refl \to Just\ Refl\\
&\qquad Nothing\ \ \to Nothing\\
&decV\ (FORALL\ v1)\ (FORALL\ v2) =\\
&\quad \mathbf{case}\ decV\ v1\ v2\ \mathbf{of}\\
&\qquad Just\ Refl \to Just\ Refl\\
&\qquad Nothing\ \ \to Nothing\\
&decV\ \_\ \ \ \ \ \ \ \ \ \ \_\ \ \ \ \ \ \ \ \ \ = Nothing
\end{aligned}
$$

While the representation contained in a dynamic value uses a closed environment, recall that in the definition of the $FORALL$ constructor, the extension of the environment is existentially quantified. Hence, to be able to recurse, the type of $decV$ is required to work on environments of a different type. Consequently, the decidable equality function on the base universe also mentions different environments:

$$
\begin{aligned}
&decU :: U\ a\ env \to U\ b\ env' \to Maybe\ (a :\equiv: b)\\
&decU\ INT\ \ \ \ \ \ \ \ \ \ INT\ \ \ \ \ \ \ \ \ \ = Just\ Refl\\
&decU\ (PAIR\ u1\ u1')\ (PAIR\ u2\ u2') =\\
&\quad \mathbf{case}\ (decU\ u1\ u2, decU\ u1'\ u2')\ \mathbf{of}\\
&\qquad (Just\ Refl, Just\ Refl) \to Just\ Refl\\
&\qquad \_\ \to Nothing\\
&decU\ (u1 :\Rightarrow: u1')\ \ \ (u2 :\Rightarrow: u2')\ \ \ =\\
&\quad \mathbf{case}\ (decU\ u1\ u2, decU\ u1'\ u2')\ \mathbf{of}\\
&\qquad (Just\ Refl, Just\ Refl) \to Just\ Refl\\
&\qquad \_\ \to Nothing\\
&decU\ (VAR\ i)\ \ \ \ \ \ \ \ (VAR\ j)\ \ \ \ \ \ \ \ =\\
&\quad \mathbf{case}\ decRef\ i\ j\ \mathbf{of}\\
&\qquad Just\ Refl \to Just\ Refl\\
&\qquad Nothing\ \ \to Nothing\\
&decU\ \_\ \ \ \ \ \ \ \ \ \ \_\ \ \ \ \ \ \ \ \ \ = Nothing
\end{aligned}
$$

In comparison to the function $decU$ from Section 2.1, only its type is changed and the branch for variables is added. It uses an additional function to decide when two references are equal:

$$
\begin{aligned}
&decRef :: Ref\ a\ env \to Ref\ b\ env' \to Maybe\ (a :\equiv: b)\\
&decRef\ Rz\ \ \ \ Rz\ \ \ \ = ...\\
&decRef\ (Rs\ i)\ (Rs\ j) =\\
&\quad \mathbf{case}\ decRef\ i\ j\ \mathbf{of}\\
&\qquad Just\ Refl \to Just\ Refl\\
&\qquad Nothing\ \ \to Nothing\\
&decRef\ \_\ \ \ \ \_\ \ \ \ = Nothing
\end{aligned}
$$

While the inductive case for Rs structurally recurses, the base case for Rz cannot be defined. Since the environments to which we refer are different, as argued before, so are the references to the top variable.

We cannot alleviate this problem since this would require the decidable equality functions to work on environments of the same type. As mentioned earlier, the $FORALL$ constructor extends the environment via existential quantification. Consequently, we cannot compare such representations since the references to variables in the environment are not known to reflect the same type.

2.2.2 Higher-order abstract syntax

Instead of adding a construct to reference variables in our universe, the second approach piggybacks on the use of variables in the host language; a technique that is called higher-order abstract syntax. Hence, we can reuse the definition of U from Section 2.1 and do not need to extend it with a constructor for variables. We give another definition of the universe V to introduce quantifiers and bind variables:

```
data V :: * → * where
  BASE   :: U a → V a
  FORALL :: (U a → V b) → V b
```

Now, binding a variable involves taking a function that given a representation, returns a representation that possibly binds more variables. Again, we impose predicativity by defining the domain of the function to be of the universe U instead of V. Closing the term is enforced by the use of the binding structure of the host language. In this approach, the type of the polymorphic identity function is represented as $FORALL\ (\lambda a \to BASE\ (a :\Rightarrow: a))$.

A dynamic value is defined in a straightforward fashion:

```
data Dyn = forall a . Dyn (V a) a
```

We continue by defining the decidable equality function on the new definition of V:

```
decV :: V a → V b → Maybe (a :≡: b)
decV (BASE u1)    (BASE u2)    =
  case decU u1 u2 of
    Just Refl → Just Refl
    Nothing   → Nothing
decV (FORALL v1) (FORALL v2) = ...
decV _           _           = Nothing
```

The branch for the base universe proceeds with the monomorphic decidable equality function as defined earlier in Section 2.1. Unfortunately, the constructor that binds variables cannot proceed in the usual fashion since the representations are functions, and their domain is not finite. Hence, we cannot compare these functions using existential equality. Another approach would be to saturate the functions with dummy values such as De Bruijn indices, and compare the resulting representations (Atkey et al., 2009). But to be able to distinguish the different dummy values, we need an environment and we have already shown this path to be a dead end.

2.3 Difficulties

Using generalised algebraic datatypes we are able to attach the type that a constructor reflects to its representation. This has proven to be essential to enforce the relation between values and the representations of their types. Unfortunately, this property prevents the definition of polymorphic dynamic typing. Since we need the same variables to reflect the same type, the universes for the representation of polymorphic types have to carry around an environment. However, the extension of the environment is existentially quantified, obstructing comparison of references to such environments.

The only way to circumvent these troubles is to postpone the use of an environment and hence the attachment of types to representations. This demands a separation between a universe for the representation of types and its interpretation as a type. The intuition is that the separation allows us to first perform the desired operations on representations, after which we perform the interpretation at the latest moment. This allows us, for instance, to compare representations without having any attached interpretation in the way.

Ideally, we would like to define a function that interprets a representation and returns the type that it must reflect. Haskell provides some way to do type-level computations via generalised algebraic datatypes and type families (Schrijvers et al., 2008). While it is possible to embed polymorphic dynamic typing in Haskell by making heavy use of these tools, we believe that a dependently-typed language provides a more natural approach.

3. Embedding in a dependently-typed language

In this section we use Agda (Norell, 2007 & 2008) and discuss how monomorphic dynamic typing can be embedded in this language (Section 3.1). Then, we show how to elegantly embed polymorphic dynamic typing, for now limiting ourselves to using structural equality of representations (Section 3.2).

3.1 Monomorphic dynamic typing

We begin by defining a universe to represent monomorphic types:

```
data U : Set where
  NAT  : U
  PAIR : U → U → U
  _⇒_  : U → U → U
```

The difference with the representation from Section 2.1 is that the interpretation of the representation (i.e., the type that a representation reflects) is detached. The corresponding type is obtained by the function el that computes the type that a value of this universe represents:

```
elU : U → Set
elU NAT          = Nat
elU (PAIR u u')  = Pair (elU u) (elU u')
elU (u ⇒ u')     = elU u → elU u'
```

This function returns a type when given a value. The base case returns the monomorphic type Nat whereas the other branches for pairs and functions recurse while constructing a pair or function type. Then, a dynamic value constitutes a representation with a value of the interpreted type:

```
data Dyn : Set where
  dyn : (u : U) → elU u → Dyn
```

A cast function needs a proof that a provided representation is equal to the representation that is contained in the dynamic, before unwrapping its value:

```
data _≡_ {a : Set} (x : a) : a → Set where
  Refl : x ≡ x
```

This datatype is similar to the Haskell datatype (:≡:) from Section 2.1, although there is one important difference: this equality states that two values are equal while the Haskell datatype states that two types are equal. Here, we only need a proof on the value level since a representation U does not reflect the type it represents. The function that decides if two representations are equal is defined as follows:

```
decU : forall {n} → (u1 u2 : U n) →
         Maybe (u1 ≡ u2)
decU NAT          NAT   = Just Refl
decU (PAIR u1 u1') (PAIR u2 u2')
  with decU u1 u2 | decU u1' u2'
decU (PAIR u1 u1') (PAIR .u1 .u1')
    | Just Refl | Just Refl = Just Refl
... | _         | _         = Nothing
decU (u1 ⇒ u1')  (u2 ⇒ u2')
  with decU u1 u2 | decU u1' u2'
decU (u1 ⇒ u1')  (.u1 ⇒ .u1')
    | Just Refl | Just Refl = Just Refl
... | _         | _         = Nothing
decU _             _       = Nothing
```

Coming up with a proof in the case of integer representations is easy. In the branches for pairs and functions we pair-wise recurse using the **with** keyword and pattern match on the results. Note that when pattern matching on a proof Refl, we have to restate the branch and use the dot-notation to explicitly state that the two compared elements are equal. Otherwise, we use the shorthand notation ... to restate the original branch before we come to pattern matching the results of recursion.

Next, we define the cast function:

```
cast : (u1 : U) → Dyn → Maybe (elU u1)
cast u1 (dyn u2 x) with decU u1 u2
cast u1 (dyn .u1 x) | Just Refl = Just x
...                 | Nothing   = Nothing
```

When the function decU gives the proof Refl that both representations are equal, we can return the value that is contained in the dynamic. Note that although the proof only states the equality of the representation values, the type checker also learns that elU u1 equals elU u2.

3.2 Polymorphic dynamic typing

The above definitions are straightforwardly defined in Agda and resemble the approach explained in Section 2.1. Next, we will extend the Agda approach to polymorphic types.

First, we need a way to introduce quantifiers and bind variables. To be able to close a representation, we include a type parameter that indicates the number of variables that a representation can use:

```
data V (n : Nat) : Set where
  BASE   : U n → V n
  FORALL : V (Succ n) → V n
```

The FORALL constructor permits the use of an additional variable, which are referenced in the datatype U:

```
data U (n : Nat) : Set where
  NAT  : U n
  PAIR : U n → U n → U n
  _⇒_  : U n → U n → U n
  VAR  : Fin n → U n
```

The number of variables in a representation is carried along the way. The datatype Fin n describes the finite number of values between Zero and n:

```
data Fin : Nat → Set where
  Fz : forall {n} → Fin (Succ n)
  Fs : forall {n} → Fin n → Fin (Succ n)
```

Note that its definition bears a resemblance to the definition of the datatype Ref from Section 2.2.

Using the universes U and V, the type of the polymorphic identity function is represented as FORALL (BASE (VAR Fz ⇒ VAR Fz)). To obtain the actual types that representations reflect, we need interpretation functions. As before in Section 2.2, we use De Bruijn indices in an environment to keep track of the types associated with variables and to guarantee that occurrences of the same variables reflect the same type:

```
data Env : (n : Nat) → Set where
  Nil  : Env Zero
  Cons : forall {n} → U Zero →
           Env n → Env (Succ n)
```

Looking up an entry in an environment is straightforward:

```
lookup : forall {n} → Fin n → Env n → U Zero
lookup Fz     (Cons u _)   = u
lookup (Fs i) (Cons _ env) = lookup i env
```

Note that we do not need to provide a branch for the empty environment since the constructors of the Fin type do not permit a reference to an empty environment. In the base case for Fz we take the head entry, and otherwise we recurse with the tail of the environment.

The interpretation function of the universe V takes such an environment:

```
elV : forall {n} → V n → Env n → Set
elV (BASE u)    env = elU u env
elV (FORALL v) env = forall {u} → elV v (Cons u env)
```

The FORALL constructor quantifies over a representation that uses no variables, again to enforce predicativity, and adds it to the provided environment before recursing. This resembles the type of the equally named constructor as given in Section 2.2. The base case recurses in the interpretation of the universe U:

```
elU : forall {n} → U n → Env n → Set
elU NAT         _   = Nat
elU (PAIR u u') env = Pair (elU u env) (elU u' env)
elU (u ⇒ u')    env = elU u env → elU u' env
elU (VAR i)     env = elU0 (lookup i env)
```

The representations of integers, pairs, and functions map to their respective types. In the case of a variable we use the environment that is passed along to obtain the type that this variable reflects. Because we quantify over representations, we have to interpret its result again using elU0, which we will define next.

Since we are typically dealing with closed representations in dynamic values, we define an interpretation function of such representations separately:

```
elV0 : V Zero → Set
elV0 (BASE u)    = elU0 u
elV0 (FORALL v) = elV (FORALL v) Nil
```

Again, the FORALL constructor quantifies over a variable, but recurses in the earlier defined elV function with the untouched representations and the empty environment Nil. The base case simply recurses in the interpretation of the universe U that does not contain any variable references:

```
elU0 : U Zero → Set
elU0 NAT         = Nat
elU0 (PAIR u u') = Pair (elU0 u) (elU0 u')
elU0 (u ⇒ u')    = elU0 u → elU0 u'
elU0 (VAR ())
```

The function elU0 recurses the structure of the argument representation and produces types along the way. Since we know that such a representation cannot contain any variables (i.e., the type Fin Zero is uninhabited), we have to define the final branch for VAR as an impossible pattern.

Given the universes and their interpretation, we define a dynamic value as follows:

```
data Dyn : Set where
  dyn : (v : V Zero) → elV0 v → Dyn
```

Since a representation no longer directly reflects its corresponding type, as in Section 3.1, we use the interpretation function to compute the desired type for the value in the dynamic.

We continue by defining decidable equality on our representations, such that we are later able to unwrap the value that is contained in the dynamic:

```
decV : forall {n} → (v1 v2 : V n) →
         Maybe (v1 ≡ v2)
decV (BASE u1)    (BASE u2)
  with decU u1 u2
decV (BASE u1)    (BASE .u1)
    | Just Refl       = Just Refl
... | Nothing         = Nothing
```

```
decV (FORALL v1) (FORALL v2)
  with decV v1 v2
decV (FORALL v1) (FORALL .v1)
    | Just Refl        = Just Refl
... | Nothing         = Nothing
decV _              _ = Nothing
```

Representations that introduce quantifiers and bind variables are considered equal if their body is equal. In the base case we recurse in the function for decidable equality of the universe U:

```
decU : forall {n} → (u1 u2 : U n) →
       Maybe (u1 ≡ u2)
decU NAT          NAT  = Just Refl
decU (PAIR u1 u1') (PAIR u2 u2')
  with decU u1 u2 | decU u1' u2'
decU (PAIR u1 u1') (PAIR .u1 .u1')
    | Just Refl | Just Refl = Just Refl
... | _         | _        = Nothing
decU (u1 ⇒ u1')  (u2 ⇒ u2')
  with decU u1 u2 | decU u1' u2'
decU (u1 ⇒ u1')  (.u1 ⇒ .u1')
    | Just Refl | Just Refl = Just Refl
... | _         | _        = Nothing
decU (VAR i)      (VAR j)
  with decFin i j
decU (VAR i)      (VAR .i)
    | Just Refl            = Just Refl
... | _                    = Nothing
decU _            _        = Nothing
```

This definition greatly resembles the definition of decU in Section 3.1. We recurse over the structure in parallel and collect proofs as we go, but we add the branch for variables where we use the function decFin:

```
decFin : forall {n} → (i j : Fin n) → Maybe (i ≡ j)
decFin Fz     Fz                = Just Refl
decFin (Fs i) (Fs j) with decFin i j
decFin (Fs i) (Fs .i) | Just Refl = Just Refl
...                   | Nothing  = Nothing
decFin _      _                 = Nothing
```

Since the references in this approach no longer have a type attached to it, this function merely amounts to verifying that the indices are the same.

Then, we are finally able to define the cast function on the universe that represents polymorphic types:

```
cast : (v1 : V Zero) → Dyn → Maybe (elV0 v1)
cast v1 (dyn v2 x) with decV v1 v2
cast v1 (dyn .v1 x) | Just Refl = Just x
...                 | Nothing  = Nothing
```

This definition is much like the definition of cast in Section 3.1: we compare the given representation with the representation contained in the dynamic and return the value if the proof tells us that the representations are the same.

4. Unification of type representations

The previous section shows how to embed polymorphic dynamic typing in Agda. However, we used structural equality on representations to define the cast function. Typically, you would want to unify the required representation with the one that is found in the dynamic value. Caution is advised since the required representation is not allowed to represent a more general type than that of the representation in the dynamic. In other words, we are only allowed to unwrap a value from a dynamic when the required type is an instance of the type at hand. Hence, we have to perform unification of representations, just like in the Clean system, to determine if one is an instance of the other.

Recalling the example from the introduction, consider the identity function wrapped in a dynamic using our Agda universe for the representation of polymorphic types:

```
idType : V Zero
idType = FORALL (BASE (VAR Fz ⇒ VAR Fz))

idDyn : Dyn
idDyn = dyn idType (λ x → x)
```

Then, the following expression must yield an identity on integers since the required representation is an instance of the one in the dynamic:

```
cast (BASE (NAT ⇒ NAT)) idDyn
```

The other way around, consider a dynamic value that contains the increment function on integers:

```
incDyn : Dyn
incDyn = dyn (BASE (NAT ⇒ NAT)) (λ x → x + 1)
```

When we try to unwrap the function from the dynamic, we are not allowed it to cast it to the more general type of the polymorphic identity function. For example, the following expression must fail and return Nothing:

```
cast idType incDyn
```

The definition of unification of representations in Agda is subtle and requires great care. Agda requires a function to be structurally recursive to ensure its termination. However, most unification algorithms do not exhibit this property since substitutions obtained from one branch are applied to other branches before proceeding in recursion. We base our instance-of algorithm on earlier work by McBride (2003) in Epigram where the algorithm is made structurally recursive by guaranteeing that the entries of substitution each remove a variable. We reuse the parts we need in Agda and adapt the main algorithm to perform an instance-of check rather than unification. For the full details of the original approach we refer the reader to McBride's paper.

In this section, we first discuss the core of unification, namely substitution (Section 4.1). We continue by showing how to perform unification and adapt it to an instance-of algorithm (Section 4.2). Later in Section 5, we will see how the cast function strips off quantifiers and uses the instance-of algorithm on representations in the base universe U. Therefore, this section is only concerned with such representations.

4.1 Substitution

The core of the unification algorithm lies in substitution. This is modelled as an associative list that maps variables to representations that may contain other variables:

```
data AList : Nat → Nat → Set where
  Nil  : forall {n} → AList n n
  Cons : forall {n m} → Fin (Succ n) → U n →
         AList n m → AList (Succ n) m
```

Note that we can use the same constructor names as in the datatype Env from Section 3.2 since Agda supports constructor overloading. The intuition of this datatype is that a substitution of type AList n m maps values of type U n to U m. The empty substitution Nil is the identity substitution, taking a U n to U n. An entry in the substitution takes a U (Succ n) to U m by replacing a variable Fin (Succ n) with a representation U n, given that we have a substitution that takes such U n representations to U m. Hence,

each step of the substitution guarantees that it removes a variable. This fact is later used in the unification algorithm to guarantee structural recursion and its termination.

A substitution is defined as a dependent pair, or Σ-type:

```
data Exists (a : Set) (b : a → Set) : Set where
  Witness : (x : a) → b x → Exists a b

fsts : forall {a b} → Exists a b → a
fsts (Witness x _) = x
```

Then, a substitution only exposes the type of the representation it operates on:

```
Subst : Nat → Set
Subst n = Exists Nat (λ m → AList n m)
```

The witness describes the number of variables to which the substitution maps. Using this approach, we are allowed to choose this number whenever we construct a substitution. As mentioned earlier, this is used to guarantee that each entry in the substitution reduces the number of variables.

Such a substitution is applied to a representation using the following function:

```
apply : forall {n} → (subst : Subst n) →
        U n → U (fsts subst)
apply _              NAT          = NAT
apply subst          (PAIR u u')  =
  PAIR (apply subst u) (apply subst u')
apply subst          (u ⇒ u')     =
  apply subst u ⇒ apply subst u'
apply (Witness _ xs) (VAR i)      = sub xs i
```

The type of this function states that the number of variables left in the result of the application is dictated by the substitution. The function recurses over the structure of representation until it encounters a variable. Here, it uses a function that applies the actual associated list to the variable at hand:

```
sub : forall {m n} → AList m n → Fin m → U n
sub Nil           j = VAR j
sub (Cons i t xs) j = mapVar (sub xs) ((t for i) j)
```

If the substitution is empty, we reconstruct the variable. Otherwise, we replace the variable that is contained in the substitution with its associated representation. We continue by applying the tail of the substitution to this result since this possibly contains new variables that need to be substituted. The function mapVar recurses over a representation and applies a function to each variable:

```
mapVar : forall {m n} → (Fin m → U n) →
         U m → U n
mapVar _ NAT          = NAT
mapVar f (PAIR u u')  =
  PAIR (mapVar f u) (mapVar f u')
mapVar f (u ⇒ u')     =
  mapVar f u ⇒ mapVar f u'
mapVar f (VAR i)      = f i
```

The following function performs the actual substitution:

```
_for_ : forall {n} → U n → Fin (Succ n) →
        Fin (Succ n) → U n
(t for i) j with thick i j
... | Just j'   = VAR j'
... | Nothing = t
```

First, it verifies that the variable to be replaced is equal to the variable under substitution. If this is not the case, we obtain a 'smaller' Fin value and return it. Otherwise, the variables are equal and we return the representation for which we must substitute. The function thick performs the comparison:

```
thick : forall {n} → Fin (Succ n) →
        Fin (Succ n) → Maybe (Fin n)
thick {Zero}   Fz     Fz     = Nothing
thick {Zero}   _      (Fs ())
thick {Zero}   (Fs ()) _
thick {Succ n} Fz     Fz     = Nothing
thick {Succ n} Fz     (Fs j) = Just j
thick {Succ n} (Fs _) Fz     = Just Fz
thick {Succ n} (Fs i) (Fs j) with thick i j
... | Just j'                = Just (Fs j')
... | Nothing                = Nothing
```

The function thick i maps a value of type Fin (Succ n) to Maybe (Fin n). If the two arguments of thick are equal, it will return Nothing. If the second argument j is larger than i, the call to thick i j decrements j to produce a value of type Fin n; if j is less than i, the call to thick i j leaves j untouched, but only changes its type. The function thick is defined by induction on n. When n is Zero, there is only one possible inhabitant of Fin (Succ Zero) and hence we return Nothing. For larger n, we compare i and j and return Nothing when they are equal.

4.2 Instance-of algorithm

Now that we have defined the type of substitutions and their application, we continue by defining the functions that construct substitutions. As we will see later, the unification algorithm traverses the structure of representations. There, it will unify variables with variables, as well as variables with arbitrary representations. We define two functions to deal with these cases, flexFlex and flexRigid.

The first function is defined as follows:

```
flexFlex : forall {n} → Fin n → Fin n → Subst n
flexFlex {Zero}   () ()
flexFlex {Succ n} i  j with thick i j
... | Just j'  = Witness n (Cons i (VAR j') Nil)
... | Nothing = Witness (Succ n) Nil
```

We already know that Fin Zero has no inhabitants, hence, we define this as an impossible pattern. In the other case, we verify if both variables happen to be the same using the function thick from Section 4.1. If not, we construct a singleton substitution that maps the first argument of flexFlex to the result of thick. Otherwise, the variables were equal and we return the empty substitution. Here we use the witness to choose and specify the number of variables to which the resulting substitution maps. Since the singleton substitution will remove a variable from a representation, its witness is n while in the other case the number of variables is left untouched and hence is Succ n.

The second function is different in the sense that it works on variable and non-variable representations:

```
flexRigid : forall {n} → Fin n → U n →
            Maybe (Subst n)
flexRigid {Zero}   () _
flexRigid {Succ n} i  u with check i u
... | Just u'  = Just (Witness n (Cons i u' Nil))
... | Nothing = Nothing
```

Again, the base case is an impossible pattern. The inductive case proceeds with the occur-check. If the variable does not occur in the representation, we obtain a representation with fewer variables, similar to the function thick. Otherwise, the variable does occur in the representation and the unification as a whole must fail, hence we return Nothing. The function check performs the occur-check:

```
check : forall {n} → Fin (Succ n) → U (Succ n) →
          Maybe (U n)
check _ NAT             = Just NAT
check i   (PAIR u u') with check i u | check i u'
... | Just cu | Just cu' = Just (PAIR cu cu')
... | _       | _        = Nothing
check i   (u ⇒ u') with check i u | check i u'
... | Just cu | Just cu' = Just (cu ⇒ cu')
... | _       | _        = Nothing
check i   (VAR j) with thick i j
... | Just j'            = Just (VAR j')
... | Nothing            = Nothing
```

It traverses the representation, reconstructing the results obtained from recursion in the branches for pairs and functions. In the branch for a variable it uses the function thick to test if the two variables are equal.

Now, we come to the actual unification algorithm. Remember that in contrast to the original algorithm, we want to define a function that determines if the first argument is an instance of the second argument and returns the witnessing substitution. It uses an accumulating parameter for the substitutions constructed thus far:

```
iofAcc : forall {n} → U n → U n → Subst n →
           Maybe (Subst n)
```

Although the two arguments are required to use the same number of variables, in practice this is not always the case. In Section 5.4 we will consider that we first align representations by weakening so that they have the same number of variables.

The definition of the function proceeds first by structural recursion on representations that do not contain variables. First, the base case for integers returns the accumulated substitution:

```
iofAcc NAT NAT subst = Just subst
```

Then, the cases for pairs and functions simply recurse over their branches:

```
iofAcc (PAIR u1 u1') (PAIR u2 u2') subst
  with iofAcc u1 u2 subst
... | Just subst' = iofAcc u1' u2' subst'
... | Nothing     = Nothing
iofAcc (u1 ⇒ u1')  (u2 ⇒ u2')  subst
  with iofAcc u1 u2 subst
... | Just subst' = iofAcc u1' u2' subst'
... | Nothing     = Nothing
```

It is important that we do not immediately apply the substitution obtained from recursion in the left elements to the right elements, since this would not be structurally recursive. Instead, we simply pass on the obtained substitution and postpone its application.

The branches for variables use the previously defined functions for the construction of substitutions:

```
iofAcc (VAR i) (VAR j) (Witness _ Nil) = Just (flexFlex j i)
iofAcc v       (VAR j) (Witness _ Nil) = flexRigid j v
```

This is the point where we adapt McBride's algorithm. In the case of two variables, we swap the arguments to flexFlex such that a resulting substitution maps j to i. We also remove the branch where the first argument is a variable and only allow the second argument to be a variable. These two modifications enforce the fact that the first must be an instance of the second.

The previous cases never apply the constructed substitutions. Instead, this is postponed until none of the above cases hold:

```
iofAcc _ _ (Witness _ Nil) = Nothing
iofAcc u1 u2 (Witness n (Cons i t xs))
  with iofAcc (mapVar (t for i) u1) (mapVar (t for i) u2)
         (Witness n xs)
... | Just (Witness n' xs') = Just (Witness n' (Cons i t xs'))
... | Nothing               = Nothing
```

When there is no substitution left to apply, the function fails with Nothing. Otherwise, we take the head of the substitution and try to substitute each variable in both representations. We recurse with the tail of the substitution and reconstruct the result.

Then, all that is left to do is to start the accumulating function:

```
iof : forall {n} → U n → U n → Maybe (Subst n)
iof {n} u1 u2 = iofAcc u1 u2 (Witness n Nil)
```

We provide the empty substitution to the accumulating function, which concludes the definition of the instance-of algorithm.

5. Cast

In Section 3, we defined a straightforward cast function in Agda that uses structural equality on representations. In the previous section we have stated our desire to verify if one representation is an instance of the other using unification. If this is the case, we know that it is safe to unwrap the corresponding value from a dynamic, but we must also convince Agda of this fact. It turns out that the definition of the cast function that uses the instance-of algorithm is more intricate than we anticipated at first.

In this section we show how to define such a cast function[3]. Recall the type of the cast function:

```
cast : (v1 : V Zero) → Dyn → Maybe (elV0 v1)
```

The representation that is contained in the dynamic, say v2, is different from the argument representation. To be able to unwrap its corresponding value, we have to transform a value of type elV0 v2 to elV0 v1. The trick is to take advantage of the form of the interpretation functions from Section 3.2. These functions receive an environment that is used to assign the same type to the same occurrences of a variable. Hence, this is our entrypoint to instantiate variables using the substitution gained from performing the instance-of algorithm! But before we get to that point, we have to transform the value that is contained in the dynamic to a form where we can provide it an environment of our choosing. Then, we transform it back to be a value of the type that we are returning. The framework is defined by the following steps:

1. The type elV0 v2 of the value in the dynamic is unpacked such that it is in a form where it takes an environment, being the empty environment at first.
2. The quantifiers of v2 are stripped, thereby requiring an entry in the environment for each variable.
3. The variables in this environment are merged with the substitution obtained from the instance-of algorithm.
4. The value is coerced using a correctness proof of the instance-of algorithm such that its type includes the argument representation v1 and not v2.
5. The variables that are not instantiated are quantified by dressing with the quantifiers of v1, thereby emptying the environment of any variables.
6. The remaining empty environment is packed back into place to obtain the type elV0 v1.

[3] Since we are juggling with variables, many steps in this section require proofs such as $n + \mathsf{Zero} \equiv n$, $n + \mathsf{Succ}\ m \equiv \mathsf{Succ}\ (n + m)$, and $n + m \equiv m + n$. We leave out such coercions since these are irrelevant and merely clutter the presentation of the code.

The most interesting part of this framework lies in the middle two steps; this is where we instantiate the variables and perform the coercion from one representation to the other. We discuss each of these steps separately (Sections 5.1 to 5.6). Finally, these steps are combined with the instance-of algorithm from the previous section into a single cast function (Section 5.7).

5.1 Unpack the empty environment

The representation that is included with the value in a dynamic is closed. To be able to later instantiate the variables in this representation via the environment, we first have to gain access to an environment. Therefore, we start by unpacking the empty environment in the interpretation function:

```
unpackEnvV : (v : V Zero) → elV0 v →
                 ((env : Env Zero) → elV v env)
unpackEnvV (BASE u)    x Nil = unpackEnvU u x
unpackEnvV (FORALL _) x Nil = x
```

A value belonging to a FORALL constructor does not require any further work since its interpretation in elV0 is defined as the interpretation using elV and the empty environment, as given in Section 3.2. In the base case we recurse by unpacking the representation of type U:

```
unpackEnvU : (u : U Zero) → elU0 u → elU u Nil
unpackEnvU NAT         x        = x
unpackEnvU (PAIR u u') (x , x') =
  (unpackEnvU u x , unpackEnvU u' x')
unpackEnvU (u ⇒ u')  f          =
  λ x → unpackEnvU u' (f (packEnvU u x))
unpackEnvU (VAR ())    _
```

The function recurses over the structure of the representation, excluding the branch for variables using an impossible pattern. Due to co- and contravariance in the case of functions, unpacking the representations involved requires a counterpart definition and mutual recursion:

```
packEnvU : (u : U Zero) → elU u Nil → elU0 u
packEnvU NAT         x        = x
packEnvU (PAIR u u') (x , x') =
  (packEnvU u x , packEnvU u' x')
packEnvU (u ⇒ u')  f          =
  λ x → packEnvU u' (f (unpackEnvU u x))
packEnvU (VAR ())    _
```

Again, we recurse the structure of the representations and exclude the branch for variables. In the branch for functions we recurse back into unpackEnvU.

5.2 Strip the quantifiers

Now that we have gained access to an environment, we need to strip the quantifiers from the representation in the dynamic and extend the environment for each variable. This is achieved using the following function:

```
toElU : forall {n} → (v : V n) →
          ((env : Env n) → elV v env) →
          ((env : Env (vars v + n)) → elU (strip v) env)
toElU (BASE _)   f = f
toElU (FORALL v) f =
  toElU v (λ env' → f (tl env') {hd env'})
```

The type of toElU expresses that if we strip the quantifiers from a representation, we can transform a value living in elV to elU, but only if we add an entry to the environment for each bound variable in the representation.

The functions vars and strip are defined as follows:

```
vars : forall {n} → V n → Nat
vars (BASE _)   = Zero
vars (FORALL v) = Succ (vars v)

strip : forall {n} → (v : V n) → U (vars v + n)
strip (BASE u)   = u
strip (FORALL v) = strip v
```

In the base case of toElU there are no quantifiers to strip, hence we can simply return f. In the case of a quantifier, we recurse in the inner representation. The value that we provide in recursion receives an environment with one more entry than the original value requires. Hence, we unfold this environment and feed it only the tail. This leaves us with the head entry, which we provide as an implicit argument as described by the interpretation function. The helper functions on environments are easily defined:

```
hd : forall {n} → Env (Succ n) → U Zero
hd (Cons u _) = u

tl : forall {n} → Env (Succ n) → Env n
tl (Cons _ env) = env
```

Since we know that the environment contains at least one entry, we do not need to pattern match on the empty environment Nil.

5.3 Instantiate the environment

The previous steps have prepared us for the point where we can use the environment as the entrypoint to instantiate variables in the interpretation function. As we have not yet finished the implementation of our framework, we leave the function that captures this behaviour as a postulated lemma:

```
instEnv :
  forall {n} → (u : U n) → (subst : Subst n) →
    ((env : Env n) → elU u (replaceEnv subst env)) →
    ((env : Env (fsts subst)) → elU (apply subst u) env)
```

The function replaceEnv replaces those entries in an environment for which there is a substitution available:

```
replaceEnv :
  forall {n} → (subst : Subst n) → Env n → Env n
```

The lemma instEnv captures the relation between substitutions and environments. Namely, they behave the same: it does not matter whether you instantiate the types to which variables refer in an environment using a substitution, or instantiate variables by applying that substitution to a representation.

5.4 Coerce the value

Next, we have to coerce the value such that its type is represented by the resulting representation, and not by the representation that is contained in the dynamic value. To achieve this, we need a correctness proof of the instance-of algorithm. Again, as our implementation is not yet finished, we leave this as a postulated lemma:

```
iofCorrect :
  forall {n k} → (u1 : U n) → (u2 : U (k + n)) →
    Maybe (Exists (AList (k + n) n)
                  (λ xs → apply (Witness n xs) u2
                                ≡ u1))
```

Other than the function iof from Section 4.2, this lemma operates on representations that possibly have a different number of variables. We assume that a representation quantifies no more variables than it needs; FORALL (BASE NAT) is a perfectly valid representation but does not fulfill this assumption. Then, it is safe

to say beforehand that one representation can only be an instance of the other when it does not use more variables. In fact, the type of this lemma states that first argument uses k less variables than the second argument. Then, given such representations, we know that there exists a substitution from a representation with k + n variables to a representation with n variables, which applied to u2 gives us u1.

We are not completely in the dark about the definition of this lemma. Namely, it weakens the first representation with k variables and then uses the iof function to obtain the substitution. Only the proof on the resulting type of the substitution and its correctness is missing. However, a very similar proof has already been given by McBride on his unification algorithm. Namely, the unification property that applying the resulting substitution to *both* representations gives the same result. We conjecture that our modifications of the unification algorithm imply that the stated property of our instance-of algorithm holds as well.

Given the correctness proof of the instance-of algorithm, we perform the actual coercion of a value using the following function:

```
coerce : forall {n u1 u2} → (u1 ≡ u2) →
           ((env : Env n) → elU u1 env) →
           ((env : Env n) → elU u2 env)
coerce Refl f = f
```

The representations to which the coercion applies are provided as implicit arguments. We deploy the argument proof of their equality by pattern matching on Refl, which performs the coercion and allows us to return f.

5.5 Dress with quantifiers

Having performed the middle of the transformation, we follow the same path backwards. We dress the representation with quantifiers and empty the environment accordingly, obtaining the dual of toElU:

```
toElV :
  forall {n} → (v : V n) →
    ((env : Env (vars v + n)) → elU (strip v) env) →
    ((env : Env n) → elV v env)
toElV (BASE _)    f = f
toElV (FORALL v) f =
  λ env {u} → toElV v f (Cons u env)
```

The transformation function receives the original representation that dictates the quantified variables that need to be introduced. The base case is straightforward since there are no quantifiers to introduce. In the inductive case we have an environment and a quantified implicit variable where we recurse in toElV by appending the variable to the environment. Note the analogy with the elV interpretation function where we introduce an actual quantifier on the type level and also add it to the environment before recursing.

5.6 Pack the empty environment

Finally, we are left with one simple task, that is to pack the empty environment back into the interpretation function. We define the dual of unpackEnvV:

```
packEnvV : (v : V Zero) →
             ((env : Env Zero) → elV v env) →
             elV0 v
packEnvV (BASE u)    f = packEnvU u (f Nil)
packEnvV (FORALL _) f = f Nil
```

In the base case we recurse in the function for packing the universe U from Section 5.1, whereas the inductive case simply provides the empty environment to saturate the argument function.

5.7 The cast function

Now that we have described the individual steps of the framework, we almost come to the point where we combine these steps into the actual cast function.

As mentioned earlier in Section 5.4, our correctness proof of the instance-of algorithm demands that the representation presented to the cast function does not use more variables than the representation contained in the dynamic. Therefore, we first define a function to perform this check:

```
minus : (m n : Nat) →
          Maybe (Exists Nat (λ k → k + n ≡ m))
minus Zero     (Succ _) = Nothing
minus m        Zero     = Just (Witness m Refl)
minus (Succ m) (Succ n) with minus m n
minus (Succ .(k + n)) (Succ n)
    | Just (Witness k Refl) = Just (Witness k Refl)
... | Nothing               = Nothing
```

If the check succeeds, the function gives us the actual difference between the two arguments and a proof[3] that states this fact.

Given this helper function, the instance-of algorithm, and the framework described earlier, the cast function is defined as follows:

```
cast : (v1 : V Zero) → Dyn → Maybe (elV0 v1)
cast v1 (dyn v2 x) with minus (vars v2) (vars v1)
... | Nothing                    = Nothing
... | Just (Witness k p) with iofCorrect (strip v1) (strip v2)
... | Nothing                    = Nothing
... | Just (Witness xs iofLemma) = Just step6
  where
    subst : Subst (vars v2)
    subst = Witness (vars v1) xs

    step1 : (env : Env Zero) → elV v2 env
    step1 = unpackEnvV v2 x

    step2 : (env : Env (vars v2)) → elU (strip v2) env
    step2 = toElU v2 step1

    step3 : (env : Env (vars v1)) →
              elU (apply subst (strip v2)) env
    step3 = instEnv (strip v2) subst
              (λ env' → step2 (replaceEnv subst env'))

    step4 : (env : Env (vars v1)) → elU (strip v1) env
    step4 = coerce iofLemma step3

    step5 : (env : Env Zero) → elV v1 env
    step5 = toElV v1 step4

    step6 : elV0 v1
    step6 = packEnvV v1 step5
```

We verify that the first argument does not use more variables than the second argument[4]. Then, we obtain an associated list (which we turn into a substitution using a local definition) and the correctness proof via iofCorrect, before we follow the six steps of the framework as enumerated in the beginning of this section. Comparing this definition to the earlier definition of the cast function that uses structural equality, we clearly see that using the instance-of algorithm requires a lot more work and careful steps.

[4] The actual implementation of the framework uses the difference k between vars v1 and vars v2, and the proof k + vars v2 ≡ vars v1 to perform uninteresting but obligatory coercions on representations. We omit these here for the sake of presentation.

6. Related work

Dynamic typing in Haskell has been studied by both Baars and Swierstra (2002) and Cheney and Hinze (2002) around the same time. Both approaches only considered monomorphic dynamic typing. Respectively, they state: *"Whether our approach can easily be extended with dynamic polymorphism is as yet unknown and a subject of further research."* and *"We believe our Dynamic also can support making values of closed polymorphic types dynamic, although we have yet to experiment with unifying and pattern-matching polymorphic type representations."*. A similar but weaker research question has been formulated by Sheard et al. (2005) and said to be difficult (Sheard and Pasălić, 2008): *"Is it possible to build [..] singleton types to represent polymorphic types? While we have tried many approaches we are not yet satisfied with the generality of any of them."*. Unfortunately, there has not been any follow up on this work and these research questions have neither been proven nor disproven by the authors. In this paper we show how to define a representation of polymorphic types using generalised algebraic datatypes in Haskell. We also argue that a universe for the representation of types and its interpretation need to be separated to embed polymorphic dynamic typing in a functional language.

A workaround in Haskell to support dynamic typing with polymorphism has been suggested by Pang et al. (2004). The idea is that any polymorphic value can be made monomorphic by wrapping it in a vanilla datatype. While this allows us to move around such dynamic values, we are not able to unwrap it with a less general type by instantiation, like we describe in this paper.

There has also been some work on extending the Haskell library for monomorphic dynamic typing with polymorphism (Stewart, 2010). There it is argued, as we do in this paper, that polymorphism in representations requires their unification. Instead of supporting this via a library, or by extending the language itself, a hook to the compiler is provided to invoke the regular unification mechanism at run time. In our approach we do not follow this path but investigate the embedding in a language itself, thereby also experimenting with and learning about the expressivity of the language and its features.

The combination of dynamic typing and dependently-typed programming is not entirely new. Ou et al. (2004) argue that a programmer needs fine-grained control over the number of type annotations and the level of compile-time safety. A new system is described where pieces of the program are either marked *dependent* or *simple*, where the latter case is verified at run time. However, our goal is different in that we consider the embedding of dynamic typing in a dependently-typed functional language via a universe and its interpretation, instead of completely merging the two idioms by extending the system itself.

We use a dependently-typed functional language mostly for its ability to separate a universe from its interpretation, such that we can compare representations. Crary and Weirich (1999) use the same approach and define interpretation functions on a universe for the representation of polymorphic types, very much like our interpretations. However, their work concerns a system named LX that is completely dedicated to the analysis of types within a programming language, whereas we consider the embedding of such analyses in an already existing language.

A universe of representations and their interpretation functions has been shown to be an effective approach in generic programming in a dependently-typed setting (Altenkirch and McBride, 2003; Oury and Swierstra, 2008). Also, the duality relation between generic programming and dynamic typing has been described earlier (Cheney and Hinze, 2002). Hence, it comes as no surprise that we can use universe construction for dynamic typing as well. However, to our knowledge we are the first to investigate this relation in the context of the embedding of polymorphic dynamic typing.

7. Conclusion

We have explored the embedding of polymorphic dynamic typing in different settings. We argued that an approach in a functional language like Haskell requires generalised algebraic datatypes to relate values to the representation of their types, but in doing so we closed the door on comparing representations that involve the binding of variables. This is because an environment is required to make sure that the occurrences of the same variable reflect the same type. However, we can no longer compare such representations since these environments are existentially quantified.

In essence, we have shown that a universe for the representation of types needs to be separated from its interpretation as a type. While it is possible to perform this separation in Haskell by making heavy use of generalised algebraic datatypes and type families, we believe that a more natural approach is offered by a dependently-typed language such as Agda. There, we are able to elegantly postpone attaching meaning to a representation until after performing any comparison. We first defined a framework for polymorphic dynamic typing in Agda with structural equality of representations. Then, we extended this approach to use an instance-of algorithm based on unification and defined a cast function that required surprisingly intricate steps to coerce values.

We did not describe a complete framework but have two postulated lemmas: one involves transferring information between substitutions and environments, and the other the correctness of the instance-of algorithm. We believe that both can be defined and intend to do so in future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. The authors are indebted to James McKinna for invaluable discussions on the subject and pointing out the advantages of using environments over substitutions in the interpretation functions, and to Stefan Holdermans and Sjoerd Visscher for showing us how to use type families to embed polymorphic dynamic typing in Haskell. This work has been funded by the Technology Foundation STW through its project on "Demand Driven Workflow Systems" (07729).

References

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

Martín Abadi, Luca Cardelli, Benjamin Pierce, Didier Rémy, and Robert Taylor. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):81–110, 1994.

Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming, Dagstuhl, Germany*, pages 1–20. Kluwer Academic Publishers, 2003.

Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the Haskell Symposium, Haskell '09, Edinburgh, UK*, pages 37–48. ACM Press, 2009.

Arthur Baars and Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the International Conference on Functional Programming, ICFP '02, Pittsburgh, PA, USA*, pages 157–166. ACM Press, 2002.

Nicolaas de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the*

Haskell Workshop, Haskell '02, Pittsburgh, PA, USA, pages 90–104. ACM Press, 2002.

Karl Crary and Stephanie Weirich. Flexible type analysis. In Didier Remy, editor, *Proceedings of the International Conference on Functional Programming, ICFP '99, Paris, France*, pages 233–248. ACM Press, 1999.

Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

Thomas van Noort, Peter Achten, and Rinus Plasmeijer. Ad-hoc polymorphism and dynamic typing in a statically typed functional language. In Bruno Oliveira and Marcin Zalewski, editors, *Proceedings of the Workshop on Generic Programming, WGP '10, Baltimore, MD, USA*, pages 73–84. ACM Press, 2010.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Ulf Norell. Dependently typed programming in Agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Revised Lectures of the International Summer School on Advanced Functional Programming, AFP '08, Heijen, The Netherlands*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2008.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Lévy, Ernst Mayr, and John Mitchell, editors, *Proceedings of the International Conference on Theoretical Computer Science, TCS '04, Toulouse, France*, pages 437–450. Kluwer Academic Publishers, 2004.

Nicolas Oury and Wouter Swierstra. The power of Pi. In James Hook and Peter Thiemann, editors, *Proceedings of the International Conference on Functional Programming, ICFP '08, Victoria, BC, Canada*, pages 39–50. ACM Press, 2008.

André Pang, Don Stewart, Sean Seefried, and Manuel Chakravarty. Plugging Haskell in. In Henrik Nilsson, editor, *Proceedings of the Haskell Workshop, Haskell '04, Snowbird, UT, USA*, pages 10–21. ACM Press, 2004.

Emir Pasălić and Nathan Linger. Meta-programming with typed object-language representations. In Gábor Karsai and Eelco Visser, editors, *Proceedings of the International Conference on Generative Programming and Component Engineering, GPCE '04, Vancouver, BC, Canada*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer-Verlag, 2004.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Julia Lawall, editor, *Proceedings of the International Conference on Functional Programming, ICFP '06, Portland, OR, USA*, pages 50–61. ACM Press, 2006.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the Conference on Programming Language Design and Implementation, PLDI '88, Atlanta, GA, USA*, pages 199–208. ACM Press, 1988.

Marco Pil. First class file I/O. In Chris Clack, Kevin Hammond, and Antony Davie, editors, *Selected Papers of the International Workshop on Implementation of Functional Languages, IFL '97, St. Andrews, UK*, volume 1467 of *Lecture Notes in Computer Science*, pages 233–246. Springer-Verlag, 1997.

Rinus Plasmeijer and Arjen van Weelden. A functional shell that operates on typed and compiled applications. In Varmo Vene and Tarmo Uustalu, editors, *Proceedings of the 5th International Summer School on Advanced Functional Programming, AFP '04, Tartu, Estonia*, volume 3622 of *Lecture Notes in Computer Science*, pages 245–272. Springer-Verlag, 2005.

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. iTasks for a change - Type-safe run-time change in dynamically evolving workflows. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*, pages 151–160. ACM Press, 2011.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In Peter Thiemann and James Hook, editors, *Proceedings of the International Conference on Functional Programming, ICFP '08, Victoria, BC, Canada*, pages 51–62. ACM Press, 2008.

Tim Sheard and Emir Pasălić. Meta-programming with built-in type equality. *Electronic Notes in Theoretical Computer Science*, 199:49–65, 2008.

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

Don Stewart. *Dynamic extension of typed functional languages*. PhD thesis, University of New South Wales, 2010.