Witnesses and Open Witnesses

Ashley Yakeley
ashley@semantic.org

Abstract

We review *witnesses*, an emerging Haskell idiom, and suggest some terminology. We then introduce *open witnesses* as a library, and propose an extension to allow the creation of them at top-level. We show how this solves the expression problem, all with relatively little implementation fuss.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Design, Languages

Keywords Haskell, witnesses, open witnesses, expression problem, extensible data types

1. Introduction

Users of object-oriented languages such as Java and C++ will be familiar with extensible types under the name *subclassing*, the ability to create a type (a *derived class*) that extends and existing type (a *base class*). The derived class *inherits* all the data of the base class, and can extend it with additional data of any type. Values of the derived class can be (implicitly) converted to the base class without loss of this additional data: such values of the base class can be examined and recovered as the derived class.

The Haskell equivalent of this is adding new variants to an existing data-type, but despite Haskell's sophisticated type system, this is quite hard to do in the general case. It's straightforward to do if one restricts the type of the additional data to be stored and recovered, at the time "base type" is declared.

A number of solutions have been suggested for this problem (see section 5), most notably the $Data.Typeable$ [1] (and $Data.Dynamic$) modules available in the Haskell standard libraries. But this is unsafe (see section 4.2). The key to the problem is dynamically representing, or *witnessing*, the type to be stored, so that it can be matched up and recovered.

There is already existing work on witnesses. Witnesses (section 2) are values that say something about type-variables. The type of the witness defines exactly what can be said about the type variables. We're particularly interested in *simple witnesses* (section 2.1), witnesses that constrain a type-variable to a single type. Simple witnesses can be compared by value, and if the values match, we can return a proof of type identity, itself an *equality witness*.

$$\begin{aligned}&matchWitness\ ::\\ &\quad Witness\ a\ \rightarrow\ Witness\ b\ \rightarrow\ Maybe\ (EqualType\ a\ b)\end{aligned}$$

Simple witness types for closed systems of types can be defined with GADTs (section 2.2), though it is possible to do so without GADTs (section 2.5). And if we have a witness type for its elements, we can create a witness type for HList-style lists (section 2.7). We can also create witness types that reify class instances (section 2.4), so we can pass them around as values.

Our contribution is *open witnesses* (section 3), simple witnesses that can witness to *any* type, but can only be generated in the IO monad (and another OW monad we define for that purpose).

$$newIOWitness\ ::\ IO\ (IOWitness\ a)$$

With open witnesses one can create *open dictionaries* that are fully heterogenous: the same dictionary can store values of any type, with matching open witnesses as keys (section 3.2). For instance, one can implement the ST monad as a state monad on OW with an open dictionary as its state (section 3.3).

So far, so good; but we're limited in the ways open witnesses can be created. But if we extend Haskell to allow open witnesses to be declared at top level, with each one unique (section 4), we can do a lot more.

$$\langle\text{identifier}\rangle\ ::\ IOWitness\ \langle\text{type}\rangle\ \leftarrow\ newIOWitness$$

The combination of fully heterogenous dictionaries and the top-level declaration of unique strongly-typed keys to those dictionaries gives us many useful things:

- a safe version of $Typeable$ (section 4.2)
- extensible data types (section 4.3) and thus a solution to the expression problem (section 4.4)
- idioms for object oriented programming (sections 4.5, 4.6)
- bindings dictionaries for an extension for thread-local storage (section 4.7)

The purpose of this paper is twofold: to convince Haskell programmers that they want to use open witness declarations, and, relatedly, to convince implementers of Haskell that they want to implement them.

2. Witnesses

A *witness* is a value that *witnesses* some sort of constraint on some list of type variables. The type of such a value might look something like this:

$$MyWitness\ a\ b\ c$$

The constraint on a, b and c might be anything, depending on the value. Perhaps there's a class constraint. Perhaps one of them is restricted to a particular set of types. However, we're mostly interested in these three categories:

- *simple witnesses* constrain a variable to a single type
- *equality witnesses* constrain two type variables to be the same type

- *instance witnesses* constrain type variables to an instance of a type-class.

Type witnesses are not a new idea, but were one of the first uses of GADTs in Haskell. They are used in at least one major Haskell project, darcs [12]. RepLib [17] introduces *representation types* which are both simple witnesses and what we call *representatives* (section 2.6), but here we separate the two notions. This section of the paper is largely a review of existing work.

2.1 Simple Witnesses and Equality Witnesses

Simple witnesses constrain a variable to a single type, and they have a type of this form:

$$\mathit{MySimpleWitness}\ a$$

Here a is the type variable being constrained. Though a might be of any kind, throughout we shall examine only simple witnesses to types of kind $*$.

Sheard et al. [14] show how to match two simple witnesses, to provide (if they are equal) a proof of type equality. The principle is this: to count as a simple witness type, each value must constrain the a parameter to a single type. Accordingly, if two values are identical, then they have the same type (though the converse is not always true, as we shall see). This means that given two simple witnesses of types $w\ a$ and $w\ b$, we can compare them to determine whether a and b are the same type. If they are the same type, we can provide a value of type $\mathit{EqualType}\ a\ b$ (an *equality witness*) that proves the equality of a and b.

This class provides a function $\mathit{matchWitness}$ that accomplishes this:

```
class SimpleWitness w where
  matchWitness ::
    w a → w b → Maybe (EqualType a b)
```

The class comes with a constraint. To be a correct instance of $\mathit{SimpleWitness}$, all values must match themselves:

$$\mathit{matchWitness}\ \mathit{wit}\ \mathit{wit}\ =\ \mathit{Just}\ \mathit{MkEqualType}$$

Thus with the appropriate implementation, $\mathit{MySimpleWitness}$ would be an instance.

```
instance SimpleWitness MySimpleWitness where
  ...
```

$\mathit{EqualType}$ (Equal in Sheard et al. [14]) is another kind of witness type, one that witnesses to the equality of its two type arguments. It can be defined straightforwardly with GADTs:

```
data EqualType a b where
  MkEqualType :: ∀ t. EqualType t t
```

Simply by bringing a $\mathit{MkEqualType}$ into scope, its type parameters a and b can be unified in any type expression. For instance:

```
exampleUnify ::
  ∀ a b c. EqualType a b → (a, b → c) → (b, a → c)
exampleUnify MkEqualType = id
```

One cannot construct a $\mathit{MkEqualType}$ of type $\mathit{EqualType}\ a\ b$ where a and b are different types. One can of course construct $\mathit{undefined}$ of type $\mathit{EqualType}\ a\ b$ for any a and b, but since $\mathit{undefined}$ will not match against $\mathit{MkEqualType}$, it cannot be used to unify a and b.

Note that the type arguments to $\mathit{EqualType}$ have kind $*$. We could create a similar type for representing proofs of equality of type-constructors of any other kind. In practice, we will shoehorn these into $\mathit{EqualType}$ where possible. For instance, a proof that f and f', two type-constructors of kind $* \to *$, are identical, can be represented with a value of type $\mathit{EqualType}\ (f\ ())\ (f'\ ())$.

Simple witnesses constrain their parameter to a single type, rather than just to an open type expression. For instance, using GADTs, one can create a type that includes values that witness only to part of a type:

```
data MyPartialWitness a where
  MPWMaybe :: ∀ p. MyPartialWitness (Maybe p)
```

Here $\mathit{MPWMaybe}$ witnesses the parameter a to $\mathit{Maybe}\ p$, but p is left unwitnessed. $\mathit{MyPartialWitness}$ cannot be made a correct instance of $\mathit{SimpleWitness}$ and is not a simple witness type.

2.2 Witnesses with GADTs

Just as we did for our equality witness type, the easiest way to create witness types of all sorts is with *generalized algebraic data types* (GADTs) [11]. Here's an example of a simple witness type, where the possible types to be witnessed are represented with values:

```
data CharOrInt a where
  IsChar :: CharOrInt Char
  IsInt  :: CharOrInt Int
```

This gives us two witnesses, IsChar and IsInt. They have type $\mathit{CharOrInt}\ \mathit{Char}$ and $\mathit{CharOrInt}\ \mathit{Int}$, but as constructors both pattern-match as $\mathit{CharOrInt}\ a$, unifying a to Char or Int in their consequent expression. For instance:

```
somechar :: ∀ a. CharOrInt a → a
somechar = λ IsChar → 't'
```

The type of the parameter to $\mathit{CharOrInt}$ can be universally quantified ($\forall\ a.$) in the type of $\mathit{somechar}$, but IsChar has the effect of binding that parameter to Char in the consequent of the expression, 't'.

Since each value of $\mathit{CharOrInt}$ constrains the type parameter a to a single type, we can make it an instance of $\mathit{SimpleWitness}$:

```
instance SimpleWitness CharOrInt where
  matchWitness IsChar IsChar = Just MkEqualType
  matchWitness IsInt IsInt   = Just MkEqualType
  matchWitness _ _           = Nothing
```

Our example $\mathit{CharOrInt}$ covers only two types. However, we can create witness types that witness to a system of types:

```
data MyType a where
  IsChar  :: MyType Char
  IsInt   :: MyType Int
  IsMaybe :: ∀ a. MyType a → MyType (Maybe a)
  IsList  :: ∀ a. MyType a → MyType [a]
  IsPair  ::
    ∀ a b. MyType a → MyType b → MyType (a, b)
```

This covers all types creatable from Char and Int and the type constructors [], Maybe and (,). For instance, a witness to the type $[\mathit{Maybe}\ ([\mathit{Int}], \mathit{Char})]$:

```
exampleMyType :: MyType [Maybe ([Int], Char)]
exampleMyType =
  IsList $ IsMaybe $ IsPair (IsList IsInt) IsChar
```

We can write $\mathit{matchWitness}$ for MyType to make it an instance of $\mathit{SimpleWitness}$:

```
instance SimpleWitness MyType where
```

```
matchWitness IsChar IsChar =
    Just MkEqualType
matchWitness IsInt IsInt =
    Just MkEqualType
matchWitness (IsMaybe tp) (IsMaybe tq) = do
    MkEqualType ← matchWitness tp tq
    return MkEqualType
matchWitness (IsList tp) (IsList tq) = do
    MkEqualType ← matchWitness tp tq
    return MkEqualType
matchWitness (IsPair tpa tpb) (IsList tqa tqb) = do
    MkEqualType ← matchWitness tpa tqa
    MkEqualType ← matchWitness tpb tqb
    return MkEqualType
matchWitness _ _ = Nothing
```

2.3 Using Witnesses

The simplest use of witnesses is to witness the type of a value. We provide the type Any to make this easy:

```
data Any w = ∀ a. MkAny (w a) a
```

Thus the type $Any\ CharOrInt$ contains either a $Char$ or an Int: it is isomorphic to $Either\ Char\ Int$.

$$Any\ CharOrInt \cong Either\ Char\ Int$$

But witness types are more general than this: they can witness a type-variable in any expression. We can generalise Any to take a type-constructor:

```
data AnyF w f = ∀ a. MkAnyF (w a) (f a)
```

Thus we have this isomorphism:

$$AnyF\ CharOrInt\ [] \cong Either\ [Char]\ [Int]$$

Notice how the $Char$ vs. Int choice of type is separated from its use as the element type to the [] constructor. This separation is the key purpose of witness types.

2.4 Instance Witnesses

Yakeley [18] introduces *instance witnesses*, that reify an instance of a class. For example, we can define a $NumInst$ type that reifies instances of the Num class:

```
data NumInst a where
    MkNumInst :: ∀ a. Num a ⇒ NumInst a
```

Any value with a type with a Num constraint can be rewritten to take a $NumInst$ argument instead. Here are $(+)$ and $fromInteger$ as defined in the Prelude:

```
(+) :: ∀ a. Num a ⇒ a → a → a
fromInteger :: ∀ a. Num a ⇒ Integer → a
```

And here they are rewritten:

```
plus' :: ∀ a. NumInst a → a → a → a
  — Look, no Num constraint!
plus' MkNumInst = (+)
fromInteger' :: ∀ a. NumInst a → Integer → a
fromInteger' MkNumInst = fromInteger
```

And we can create $NumInst$ values for any instance of Num:

```
intNum :: NumInst Int
intNum = MkNumInst
```

2.5 Witnesses without GADTs

Baars and Swierstra [4] introduced an equality type in 2000, before GADTs were well-known:

```
newtype Equal a b = Equal (∀ f. f a → f b)
```

By "plugging in" the appropriate type constructor as f, they showed how to use it to convert type-variables in any expression. Indeed their $Equal$ is isomorphic to $EqualType$:

```
toEqual :: EqualType a b → Equal a b
toEqual MkEqualType = Equal id
fromEqual :: Equal a b → EqualType a b
fromEqual (Equal p) = p MkEqualType
```

Sulzmann and Wang [15] show how to convert GADTs into GADT-less types by a similar approach of including the general conversion function, which we can use to create witness types without GADTs:

```
data CharOrInt' a
  =IsChar(∀ f. f a → f Char)
  | IsInt  (∀ f. f a → f Int)
```

2.6 Representatives

If two simple witnesses have the same value, then they have the same type. Now we introduce *representatives*: type constructors for which if they have the same type, then they have the same value.

simple witness	*representative*
value → type	type → value

The main benefit of representatives is that we can make them instances of a class, so as to avoid passing them around explicitly.

```
class Is rep a where
  representative :: rep a
```

Simple witnesses defined with GADTs are often also representatives. The $CharOrInt$ type we defined in section 2.2, for instance, is a representative type:

```
instance Is CharOrInt Char where
  representative = IsChar
instance Is CharOrInt Int where
  representative = IsInt
```

The simplest representative is the universal representative, $Type$.

```
data Type a = MkType
```

$Type$ is useful as a parameter to polymorphic functions when one wants to make clear that just the type is being passed to specify some class instance dictionary, and not any value.

```
class Storable a where
  sizeOf :: Type a → Int
```

The definition is met trivially, since there is only one value in $Type$.

```
instance Is Type a where
  representative = MkType
```

2.7 HList-Style List Types

HList [6] provides strongly-typed heterogenous lists using two constructors called $HCons$ and $HNil$. Given a witness type for the elements, we can create a witness type for an HList of those elements.

```
data ListType w a where
  IsHNil :: ListType w HNil
  IsHCons ::
      w e → ListType w l → ListType w (HCons e l)

instance SimpleWitness w ⇒
```

```
    SimpleWitness (ListType w)
  where
    matchWitness IsHNil IsHNil = Just MkEqualType
    matchWitness
       (IsHCons we1 wl1) (IsHCons we2 wl2)
     = do
       MkEqualType ← matchWitness we1 we2
       MkEqualType ← matchWitness wl1 wl2
       return MkEqualType
```

The type may also have representatives:

```
  instance Is (ListType w) HNil where
    representative = IsHNil
  instance (Is w e, Is (ListType w) l) ⇒
    Is (ListType w) (HCons e l)
  where
    representative =
       HCons representative representative
```

As an example, we can use *CharOrInt* from section 2.2 to create a witness for lists of *Char* and *Int* values.

```
  charsInts :: Any (ListType CharOrInt)
  charsInts =
    MkAny representative (HCons 3 (HCons HNil 'a'))
```

The type *Any* (*ListType CharOrInt*) contains *HList* values where each element is either a *Char* or an *Int*.

3. Open Witnesses

As we have shown, using GADTs it is straightforward to create a simple witness type for any given finite set of types and type-constructors. We now introduce a type for a variety of simple witnesses we call *open witnesses*, that can witness any type. However, they cannot be constructed: they can only be generated in certain monads.

We present an interface for open witnesses in the *IO* monad here as the public interface to a *library extension*: while it is safe to use, it requires unsafe functions to implement.

```
  type IOWitness a
  instance SimpleWitness IOWitness
  newIOWitness :: ∀ a. IO (IOWitness a)
```

Unlike the examples of type witnesses we constructed earlier with GADTs, *IOWitness* does not encode the witnessed type. Instead, each has a generated unique value, and the *matchWitness* function matches them by this value. So while two *IOWitness* values from separate calls to *newIOWitness* may have the same type, *matchWitness* on them will return *Nothing*.

However, if two *IOWitness* arguments have the same value, then they must be the result of a single call to *newIOWitness*. They must therefore have the same type, and thus it is safe for *matchWitness* to create a *MkEqualType* for them.

We also introduce a "runnable" *OW* monad in which open witnesses (*OpenWitness*) can be generated and used. For simplicity, we generalise *IOWitness* as *OpenWitness RealWorld*.

```
  data OpenWitness s a
  instance SimpleWitness (OpenWitness s)
  data RealWorld
  type IOWitness = OpenWitness RealWorld
  newIOWitness :: ∀ a. IO (IOWitness a)

  data OW s a
  newOpenWitnessOW :: ∀ s a. OW s (OpenWitness s a)
  runOW :: ∀ a. (∀ s. OW s a) → a
  owToIO :: ∀ a. OW RealWorld a → IO a
```

The *OW* monad follows a scheme similar to the *ST* monad of using a type parameter (*s*) to prevent *OpenWitness* values (or *STRef* values in *ST*) unsafely escaping.

3.1 Implementation

We believe this open witness API cannot be implemented in safe Haskell. However, we can implement it in GHC using various unsafe extensions.

The *OpenWitness* type and the *newIOWitness* function can be implemented similarly to the *Data.Unique* module.[2] *OpenWitness* is a **newtype** of *Integer*:

```
  newtype OpenWitness s a = MkOpenWitness Integer
    deriving Eq
```

We use *unsafePerformIO* and the **NOINLINE** pragma to declare *ioWitnessSource* as an *MVar* at top level. By using *Integer* rather than *Int*, we prevent *newIOWitness* from rolling over and unsafely issuing duplicate values.

```
  ioWitnessSource :: MVar Integer
  {-# NOINLINE ioWitnessSource #-}
  ioWitnessSource = unsafePerformIO (newMVar 0)
```

An *IOWitness* is an *OpenWitness* specialised for *RealWorld*, very similar to the *ST* monad:

```
  data RealWorld
  type IOWitness = OpenWitness RealWorld
```

Our generation function *newIOWitness* uses *ioWitnessSource* to count out unique values.

```
  newIOWitness :: ∀ a. IO (IOWitness a)
  newIOWitness = do
    val ← takeMVar ioWitnessSource
    putMVar ioWitnessSource (val + 1)
    return (MkOpenWitness val)
```

For *matchWitness*, we compare the *Integer* values and return a *MkEqualType* if they are equal. We have to create the *MkEqualType* with *unsafeCoerce*.

```
  instance SimpleWitness (OpenWitness s) where
    matchWitness
       (MkOpenWitness ua)
       (MkOpenWitness ub)
     = if ua == ub
       then Just (unsafeCoerce MkEqualType)
       else Nothing
```

The *OW* monad is a pure state monad with an *Integer* state.

```
  newtype OW s a = MkOW (State Integer a)
    deriving (Functor, Monad, MonadFix)
```

The *newOpenWitnessOW* function creates new *OpenWitness* values with the current state, and then increments the state.

```
  newOpenWitnessOW :: ∀ s a. OW s (OpenWitness s a)
  newOpenWitnessOW = MkOW
    (State (λ val → (MkOpenWitness val, val + 1)))
```

To run computations of the *OW* monad, we simply run the state monad with the initial state of 0. This means that the *OpenWitness* values will be unique only within the run that created them. This is not a problem, as the *s* type parameter ensures that witnesses cannot escape *runOW*.

```
  runOW :: ∀ a. (∀ s. OW s a) → a
  runOW uw = (λ (MkOW st) → evalState st 0) uw
```

We define *owToIO* to run computations of type *OW RealWorld a* in the *IO* monad, and the generated witnesses are thus of type

$IOWitness$. The OW computation modifies the state of our top-level $MVar$, $ioWitnessSource$.

```
owToIO :: OW RealWorld a → IO a
owToIO (MkOW st) =
  modifyMVar ioWitnessSource (λ start →
  let
    (a, count) = runState st start
  in
  return (count, a)
  )
```

3.2 Open Dictionaries

Using $OpenWitness$ it is straightforward to create an open dictionary type. An $OpenDict$ can store values of any type in the same dictionary, indexed by $OpenWitness$ keys. While this can be generalised to any witness type, we present the API here specifically for $OpenWitness$ for simplicity.

```
data OpenDict s
openDictLookup ::
  OpenWitness s a → OpenDict s → Maybe a
emptyOpenDict :: OpenDict s
openDictFromList ::
  [Any (OpenWitness s)] → OpenDict s
openDictAdd ::
  OpenWitness s a → a →
  OpenDict s → OpenDict s
openDictModify ::
  OpenWitness s a → (a → a) →
  OpenDict s → OpenDict s
openDictReplace ::
  OpenWitness s a → a →
  OpenDict s → OpenDict s
type IOOpenDict = OpenDict RealWorld
```

We choose not to expose any ordering on $OpenWitness$, the type of keys of our dictionary, something that will become important in section 4. So the performance of look-up for an $OpenDict$ can be no better than $O(n)$ as we compare a given key with each key in the dictionary in turn. This could perhaps be improved by exposing an ordering privately to the $OpenDict$ implementation, but for simplicity we show an implementation that uses only what is exposed.

The type is simply a list of cells (key-value pairs), each of type Any ($OpenWitness$ s).

```
newtype OpenDict s =
  MkOpenDict [Any (OpenWitness s)]
emptyOpenDict = MkOpenDict []
openDictFromList = MkOpenDict
```

To look up a key, we go through each pair in the dictionary until a key matches.

```
matchAny :: (SimpleWitness w) ⇒
  w a → Any w → Maybe a
matchAny wit (MkAny cwit ca) = do
  MkEqualType ← matchWitness cwit wit
  return ca

openDictLookup wit (MkOpenDict cells) =
  listToMaybe (mapMaybe (matchAny wit) cells)
```

To add an entry, we simply attach it to the head with the Haskell : list construction operator.

```
openDictAdd wit a (MkOpenDict cells) =
  MkOpenDict ((MkAny wit a) : cells)
```

To modify an entry, we again go through each pair until a key matches, and then modify it:

```
replaceFirst :: (a → Maybe a) → [a] → [a]
replaceFirst f (a : aa) = case f a of
  Just newa → (newa : aa)
  _ → a : (replaceFirst f aa)
replaceFirst _ _ = []

openDictModify wit f (MkOpenDict cells) =
  MkOpenDict
    (replaceFirst
      ((fmap ((MkAny wit) . f)) . (matchAny wit))
      cells
    )

openDictReplace wit a =
  openDictModify wit (const a)
```

3.3 OW and ST

Trading one library extension for another, it is possible to build the ST monad together with $STRef$ (except for, of course, the unsafe functions) using the OW monad.

Our ST monad type is simply a state monad nesting OW, with $OpenDict$ as the state.

```
import Control.Monad.State
type ST s = StateT (OpenDict s) (OW s)
```

The basic monad-running functions are straightforward.

```
stToOW :: ST s a → OW s a
stToOW st = evalStateT st emptyWitnessDict
runST :: (∀ s . ST s a) → a
runST st = runOW (stToOW st)
fixST :: (a → ST s a) → ST s a
fixST = mfix
stToIO :: ST RealWorld a → IO a
stToIO = owToIO . stToOW
```

Our reference type, $STRef$, is simply our open witness type.

```
type STRef = OpenWitness
```

To create a new reference given an initial value, we generate it with $newOpenWitnessOW$ and store it with the value in the dictionary.

```
newSTRef :: a → ST s (STRef s a)
newSTRef a = do
  wit ← lift newOpenWitnessOW
  dict ← get
  put (openDictAdd wit a dict)
  return wit
```

To read or write a reference, we find it in the dictionary and perform the appropriate action on the dictionary entry:

```
readSTRef :: STRef s a → ST s a
readSTRef key = do
  dict ← get
  case openDictLookup key dict of
    Just a → return a
    _ → fail "ref not found"

writeSTRef :: ∀ s a. STRef s a → a → ST s ()
writeSTRef key a =
  modify (openDictReplace key a)

modifySTRef ::
```

$\forall\, s\; a.\, STRef\; s\; a \;\rightarrow\; (a \;\rightarrow\; a) \;\rightarrow\; ST\; s\; ()$
$modifySTRef\; key\; f \;=$
$\quad modify\; (openDictModify\; key\; f)$

This is not the most efficient implementation. Since we have chosen not to make *OpenWitness* an instance of *Ord*, we cannot use it as a key to a *Map*. However, it would not be hard to generate such keys (say, *Int*) in the monad and store them in our *STRef* types.

4. Open witness declarations

We now introduce a language extension that would allow the programmer to declare IOWitness values at top level.

$\langle\text{identifier}\rangle \;::\; IOWitness\; \langle\text{type}\rangle \;\leftarrow\; newIOWitness$

This extension might be generalised to allow other top level things such as *MVar*s, but for this paper we restrict ourselves to *newIOWitness*. Simple witnesses declared in this way are guaranteed to be unique, that is to match themselves (with *matchWitness*) but not match witnesses from any other declaration or from any explicit call to *newIOWitness* in the *IO* monad.

In the syntax of the Haskell 98 Report[9], we add a new case to the *topdecl* production:

$|\; pat \;::\; type \;\leftarrow\; \mathbf{newIOWitness}$

The declared *type* must be equal to $IOWitness\; t$, where t is a closed type, i.e., where all type-variables have been quantified. Informally, **forall**s, be they explicit or implicit, are not allowed *outside* the *IOWitness*.

$w1 \;::\; IOWitness\; Int \;\leftarrow\; newIOWitness$
— OK
$w2 \;::\; IOWitness\; (\forall\, a.\; IO\; a) \;\leftarrow\; newIOWitness$
— OK if impredicativity is allowed
$w3 \;::\; \forall\, a.\; IOWitness\; (IO\; a) \;\leftarrow\; newIOWitness$
— prohibited, IOa is not closed
$w4 \;::\; IOWitness\; (IO\; a) \;\leftarrow\; newIOWitness$
— prohibited, this is the same as $w3$

While this could be generalised to allow certain other *IO* functions at top-level (see section 6.2), in this paper we consider only the *newIOWitness* function.

An extension that allows the running of *IO* code at top level runs the risk of breaking various assumptions of Haskell. In particular, we want to prevent the observation of the order in which initialisers are run. The *newIOWitness* function must have no externally-observable side-effects. Furthermore, we cannot allow an *Ord* instance or any ordering of *IOWitness* values.

4.1 Implementation

Open witness declarations, like other top-level initialisers, can be written using *unsafePerformIO*, but care must be taken to ensure that the initialiser (*newIOWitness*) is run only once. In GHC, we can use the **NOINLINE** pragma. Thus

$identifier \;::\; type \;\leftarrow\; newIOWitness$

becomes

$identifier \;::\; type$
$\{-\#\; \mathbf{NOINLINE}\; identifier\; \#-\}$
$identifier \;=\; unsafePerformIO\; newIOWitness$

However, we don't need to actually run *newIOWitness*. We can instead have the compiler create its own static witnesses. For instance, we can hash unique names. For a given package P and module M, the nth witness declaration

$pat \;::\; type \;\leftarrow\; newIOWitness$

becomes

$pat \;::\; type \;=\; MkOpenWitness$
$\quad (toInteger\; (hashString\; \text{"P : M"}) \;+\; n)$

A stronger hash function could also be used if necessary.

4.2 A Safe Typeable

The *Typeable* class in *Data.Typeable* is unsafe: it allows one to create *unsafeCoerce*:

$\mathbf{newtype}\; Thing\; a \;=\; MkThing\; \{unThing \;::\; a\}$
$\mathbf{instance}\; Typeable\; (Thing\; a)\; \mathbf{where}$
$\quad typeOf\; \_ \;=\; typeOf\; ()$

$unsafeCoerce \;::\; a \;\rightarrow\; b$
$unsafeCoerce\; a \;=$
$\quad unThing\; \$\; fromJust\; \$\; cast\; \$\; MkThing\; \$\; a$

This is unavoidable if *Data.Typeable* is to allow its users to create instances of *Typeable* for their own types.

With open witness declarations, however, we can define a safe *Typeable* class. But we only implement the *representative* functionality of *Data.Typeable*: our approach avoids *TyCon* and introspection into the internal structure of types.

A naive approach is to make our *TypeRep* type *IOWitness*, and so require a witness declaration for each instance:

$\mathbf{class}\; NaiveTypeable\; a\; \mathbf{where}$
$\quad naiverep \;::\; IOWitness\; a$

This however requires a new instance declaration for each and every type that one wishes to use. For example, types such as *Int*, $[Char]$, $[Maybe\; [Bool]]$ and so forth would each require a separate instance. What we would prefer is an instance declaration only for each defined type and type constructor: declarations for [] and *Maybe* as well as *Int*, *Char* and *Bool*. So instead we create a *TypeRep* type:

$\mathbf{data}\; TypeRep\; t\; \mathbf{where}$
$\quad SimpleTypeRep \;::\! IOWitness\; t \;\rightarrow\; TypeRep\; t$
$\quad ApplyTypeRep \;::$
$\qquad TypeRep1\; p \;\rightarrow\; TypeRep\; a \;\rightarrow\; TypeRep\; (p\; a)$

And here is our *Typeable* class:

$\mathbf{class}\; Typeable\; a\; \mathbf{where}$
$\quad rep \;::\; TypeRep\; a$

Is *TypeRep* a representative as defined in section 2.6? Actually, no, as we cannot guarantee that two values of the same type have the same value. But it is a simple witness type:

$\mathbf{instance}\; SimpleWitness\; TypeRep\; \mathbf{where}$
$\quad matchWitness$
$\qquad (SimpleTypeRep\; wa)\; (SimpleTypeRep\; wb) \;=$
$\qquad\quad matchWitness\; wa\; wb$
$\quad matchWitness$
$\qquad (ApplyTypeRep\; tfa\; ta)\; (ApplyTypeRep\; tfb\; tb) \;=\; \mathbf{do}$
$\qquad\quad MkEqualType \;\leftarrow\; matchTypeRep1\; tfa\; tfb$
$\qquad\quad MkEqualType \;\leftarrow\; matchWitness\; ta\; tb$
$\qquad\quad return\; MkEqualType$
$\quad matchWitness\; \_\; \_ \;=\; Nothing$

We can use this fact to define the required *cast* and *gcast*:

$cast \;::\; \forall\, a\; b.\, (Typeable\; a, Typeable\; b) \;\Rightarrow$
$\quad a \;\rightarrow\; Maybe\; b$
$cast\; a \;=\; \mathbf{do}$
$\quad MkEqualType \;::\; EqualType\; a\; b \;\leftarrow$
$\qquad matchWitness\; rep\; rep$
$\quad return\; a$

```
gcast :: ∀ a b c. (Typeable a, Typeable b) ⇒
  c a → Maybe (c b)
gcast ca = do
  MkEqualType :: EqualType a b ←
     matchWitness rep rep
  return ca
```

We still have $TypeRep1$ to define, to witness types of kind $* \rightarrow *$. Our scheme obliges us to choose a finite set of kinds, and define a $TypeRep$X type for each one. For simplicity, we'll pick the set $\{*, * \rightarrow *, * \rightarrow * \rightarrow *\}$. We do not include, for instance, $(* \rightarrow *) \rightarrow *$, though this is the kind of our Any type.

```
data TypeRep1 (t :: * → *) where
  SimpleTypeRep1 ::
     IOWitness (t ()) → TypeRep1 t
  ApplyTypeRep1 ::
     TypeRep2 p → TypeRep a → TypeRep1 (p a)
data TypeRep2 (t :: * → * → *) where
  SimpleTypeRep2 ::
     IOWitness (t () ()) → TypeRep2 t
```

We can now create some instances for our types, both of kind $*$, ...

```
witChar :: IOWitness Char ← newIOWitness
instance Typeable Char where
  rep = SimpleTypeRep witChar
witInt :: IOWitness Int ← newIOWitness
instance Typeable Int where
  rep = SimpleTypeRep witInt
  — etc.
```

...and the higher kinds:

```
witList :: IOWitness [()] ← newIOWitness
instance Typeable a ⇒ Typeable [a] where
  rep =
     ApplyTypeRep
        (SimpleTypeRep1 witList)
        rep
witFn :: IOWitness (() → ()) ← newIOWitness
instance (Typeable a, Typeable b) ⇒
  Typeable (a → b)
where
  rep =
     ApplyTypeRep
        (ApplyTypeRep1
           (SimpleTypeRep2 witFn)
           rep
        )
        rep
  — etc.
```

The $Dynamic$ type is easy to define:

```
type Dynamic = Any TypeRep
toDyn :: Typeable a ⇒ a → Dynamic
toDyn a = MkAny representative a
fromDynamic ::
  Typeable a ⇒ Dynamic → Maybe a
fromDynamic (MkAny wit a) = do
  MkEqualType ← matchWitness wit representative
  return a
fromDyn :: Typeable a ⇒ Dynamic → a → a
fromDyn dyn def =
  fromMaybe def (fromDynamic dyn)
```

For $dynApply$, we need to examine the $TypeRep$ in the first argument, and verify, firstly, that it represents a function type; and secondly, that the type of its argument matches the $TypeRep$ of the second argument. The rest just falls into place thanks to the type-checking magic of $MkEqualType$.

```
dynApply ::
  Dynamic → Dynamic → Maybe Dynamic
dynApply
  (MkAny (ApplyTypeRep
     (ApplyTypeRep1 (SimpleTypeRep2 witFn') rx')
     ry
  ) f)
  (MkAny rx x)
 = do
  MkEqualType ← matchWitness witFn witFn'
  MkEqualType ← matchWitness rx rx'
  return (MkAny ry (f x))
dynApply _ _ = Nothing
```

4.3 Extensible Data-Types

The expression problem concerns the ability to extend types by adding new variants, and to create new functions on such types which can then be extended with new equations for the new variants.

The first part of the expression problem is the ability to add variants, and so first we must discuss what we mean by *variants*. For Haskell, a variant is normally considered as a constructor in a data-type. But our modest extension doesn't allow anything so fancy as to declare new constructors to existing data-types.

Instead, we consider *virtual constructors*. A virtual constructor is a pair of functions that do the work of a constructor, more specifically, of a single-argument constructor.

A constructor of a data-type D with a single argument of type T does two things. One is to *construct*, by acting as a function of type $T \rightarrow D$: indeed this is the type of such a constructor when considered as a function. The other is to *match*, that is, to examine whether or not a given D has that constructor, and if so, to obtain the contained T. This we can represent as a function of type $D \rightarrow Maybe\ T$. A virtual constructor, then, is simply a pair of functions we call $construct$ and $match$.

```
construct :: T → D
match :: D → Maybe T
```

We have two constraints on the functions.

- **construction**: a given T constructed as a D matches to the same T:

  $match\ .\ construct = Just$

- **uniqueness**: if a given D matches a given T, it will be constructed as the same D:

  $fmap\ construct\ (match\ d) = Just\ d\ \langle\text{or}\rangle\ Nothing$

What we want is an *extensible data-type*: some type D we can define in module $M1$, and later, given any type T, define a virtual constructor of D for T in module $M2$.

We can do this with open witness declarations. Our D is just a value with a type witnessed by $IOWitness$.

```
module M1 where
  type D = Any IOWitness
```

For $M2$, we declare a witness wit_T for T, and use it to match values inside the D.

```
module M2 where
```

$$
\begin{array}{l}
\quad \mathbf{import}\ M1 \\
\quad \mathbf{import}\ \langle\text{elsewhere}\rangle(T) \\
\quad \mathit{wit}_T\ ::\ \mathit{IOWitness}\ T\ \leftarrow\ \mathit{newIOWitness} \\
\quad \mathit{construct}_T\ ::\ T\ \rightarrow\ D \\
\quad \mathit{construct}_T\ t\ =\ \mathit{MkAny}\ \mathit{wit}_T\ t \\
\quad \mathit{match}_T\ ::\ D\ \rightarrow\ \mathit{Maybe}\ T \\
\quad \mathit{match}_T\ (\mathit{MkAny}\ \mathit{wit}\ x)\ =\ \mathbf{do} \\
\quad\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}\ \mathit{wit}_T \\
\quad\quad \mathit{return}\ x
\end{array}
$$

Let's verify that the constraints are satisfied. Firstly, the construction constraint:

$$
\begin{array}{l}
\mathbf{LHS}\ =\ \mathit{match}_T\ .\ \mathit{construct}_T \\
=\ \lambda\ t\ \rightarrow\ \mathit{match}_T\ (\mathit{construct}_T\ t) \\
=\ \lambda\ t\ \rightarrow\ \mathit{match}_T\ (\mathit{MkAny}\ \mathit{wit}_T\ t) \\
=\ \lambda\ t\ \rightarrow\ \mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}_T\ \mathit{wit}_T \\
\quad \mathit{return}\ t \\
=\ \lambda\ t\ \rightarrow\ \mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{Just}\ \mathit{MkEqualType} \\
\quad \mathit{return}\ t \\
=\ \lambda\ t\ \rightarrow\ \mathit{return}\ t\ =\ \mathit{Just}\ =\ \mathbf{RHS}
\end{array}
$$

And the uniqueness constraint:

$$
\begin{array}{l}
d\ =\ \mathit{MkAny}\ \mathit{wit}\ x \\
\mathbf{LHS}\ =\ \mathit{fmap}\ \mathit{construct}\ (\mathit{match}\ d) \\
=\ \mathit{fmap}\ \mathit{construct}_T\ (\mathit{match}_T\ (\mathit{MkAny}\ \mathit{wit}\ x)) \\
=\ \mathit{fmap}\ \mathit{construct}_T \\
(\mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}\ \mathit{wit}_T \\
\quad \mathit{return}\ x \\
) \\
=\ \mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}\ \mathit{wit}_T \\
\quad \mathit{return}\ (\mathit{construct}_T\ x) \\
=\ \mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}\ \mathit{wit}_T \\
\quad \mathit{return}\ (\mathit{MkAny}\ \mathit{wit}_T\ x) \\
=\ \mathit{Nothing}\ \langle\text{or (if } \mathit{wit}\ =\ \mathit{wit}_T)\rangle\ \mathbf{do} \\
\quad \mathit{MkEqualType}\ \leftarrow\ \mathit{matchWitness}\ \mathit{wit}\ \mathit{wit} \\
\quad \mathit{return}\ (\mathit{MkAny}\ \mathit{wit}\ x) \\
=\ \mathit{Nothing}\ \langle\text{or}\rangle\ \mathit{Just}\ d\ =\ \mathbf{RHS}
\end{array}
$$

4.4 The Expression Problem

We can consider the expression problem as a diamond-shaped pattern of dependency.

1. **define type** D
2. given type T, **extend** D with variant on T:
   $\mathit{construct}_T\ ::\ T\ \rightarrow\ D$
   $\mathit{match}_T\ ::\ D\ \rightarrow\ \mathit{Maybe}\ T$
3. given type R, **declare function** f of type $D\ \rightarrow\ R$
4. given function $f_T\ ::\ T\ \rightarrow\ R$, **define result** of $f\ .\ \mathit{construct}_T$ to be f_T.

Here points 2 and 3 depend on point 1, and point 4 depends on points 2 and 3, forming the diamond shape.

The unit of dependency in Haskell is the *module*, but Haskell has a sensible rule that added modules cannot change the behaviour of existing modules.[3] This means point 4 cannot be effective in a separate module, it must be in the same module as either point 2 or point 3. Let's consider each case.

We can put point 4 with point 3, defining the result when we declare the function, and define modules $M1$, $M2$, $M3$, each importing the previous modules:

- in $M1$, define type D
- given T, in $M2$ define variant $(\mathit{construct}_T, \mathit{match}_T)$ of D on T
- given R and $f_T\ ::\ T\ \rightarrow\ R$, in $M3$ define $f\ ::\ D\ \rightarrow\ R$ with $f\ .\ \mathit{construct}_T\ =\ f_T$.

To solve this with our open witness declarations, with virtual constructors taking the role of variants, we use our the extensible datatypes solution in the previous section for points 1 and 2. For point 3, we define f by applying match_T to its D argument to determine if it is the variant, and then give the appropriate result.

$$
\begin{array}{l}
\mathbf{module}\ M3\ \mathbf{where} \\
\quad \mathbf{import}\ M1 \\
\quad \mathbf{import}\ M2 \\
\quad \mathbf{import}\ \langle\text{elsewhere}\rangle(R, f_T) \\
\quad f\ ::\ D\ \rightarrow\ R \\
\quad f\ d\ =\ \mathbf{case}\ \mathit{match}_T\ d\ \mathbf{of} \\
\quad\quad \mathit{Just}\ t\ \rightarrow\ f_T\ t \\
\quad\quad \mathit{Nothing}\ \rightarrow\ \mathit{undefined}
\end{array}
$$

Alternatively, we can put point 4 with point 2, defining the application when we declare the variant. Again, each module imports the previous modules:

- in $MM1$, define type D
- given R, in $MM2$ define $f\ ::\ D\ \rightarrow\ R$
- given T and $f_T\ ::\ T\ \rightarrow\ R$, in $MM3$ define
  $\mathit{construct}_T\ ::\ T\ \rightarrow\ D$
  $\mathit{match}_T\ ::\ D\ \rightarrow\ \mathit{Maybe}\ T$
  such that $f\ .\ \mathit{construct}_T\ =\ f_T$.

If $MM1$ and $MM2$ were joined into a single module, so that we knew about the function f when defining our open type D, the obvious approach would be to include f directly in D:

$$
\begin{array}{l}
\mathbf{data}\ D\ =\ \mathit{MkD} \\
\{ \\
\quad \mathit{variant}\ ::\ \mathit{Any}\ \mathit{IOWitness}, \\
\quad f\ ::\ R \\
\}
\end{array}
$$

Here the result of f on the D is stored in it directly. This approach is very similar to the *virtual method table* in C++, where objects carry pointers to tables of functions, known as *methods*.

But since $MM1$ and $MM2$ are separate, we need a way of adding arbitrary functions of different types to D. The solution is, essentially, an *open* method table.

$$
\begin{array}{l}
\mathbf{module}\ MM1\ \mathbf{where} \\
\quad \mathbf{type}\ D\ =\ \mathit{IOOpenDict}
\end{array}
$$

In $MM2$, we create a witness wit_f for f, and define f to look up the witness in its D argument's method table. We don't care if it returns $\mathit{undefined}$ if the witness isn't there.

$$
\begin{array}{l}
\mathbf{module}\ MM2\ \mathbf{where} \\
\quad \mathbf{import}\ MM1 \\
\quad \mathbf{import}\ \langle\text{elsewhere}\rangle(R) \\
\quad \mathit{wit}_f\ ::\ \mathit{IOWitness}\ R\ \leftarrow\ \mathit{newIOWitness} \\
\quad f\ ::\ D\ \rightarrow\ R \\
\quad f\ d\ =\ \mathit{unJust}\ (\mathit{openDictLookup}\ \mathit{wit}_f\ d)
\end{array}
$$

For $MM3$, we're given a type T and a method function f_T of type $T\ \rightarrow\ R$. Our $\mathit{construct}_T$ function creates a D with a single entry in its method table, that is, $f_t\ t$ for key wit_f.

$$
\mathbf{module}\ MM3\ (\mathit{construct}_T, \mathit{match}_T)\ \mathbf{where}
$$

$$
\begin{array}{l}
\textbf{import } MM1 \\
\textbf{import } MM2 \\
\textbf{import } \langle\text{elsewhere}\rangle(T, f_T) \\
wit_T :: IOWitness\ T \leftarrow newIOWitness \\
construct_T :: T \rightarrow D \\
construct_T\ t = openDictFromList \\
\quad [MkAny\ wit_T\ t, MkAny\ wit_f\ (f_T\ t)] \\
match_T :: D \rightarrow Maybe\ T \\
match_T\ d = openDictLookup\ wit_T\ d
\end{array}
$$

Let's check that $f\ .\ construct_T = f_T$:

$$
\begin{array}{l}
\textbf{LHS} = f\ .\ construct_T = \lambda\ t \rightarrow f\ (construct_T\ t) \\
= \lambda\ t \rightarrow unJust\ (openDictLookup\ wit_f\ (\\
\quad openDictFromList\ [MkAny\ wit_T\ t, MkAny\ wit_f\ (f_T\ t)] \\
)) \\
= \lambda\ t \rightarrow unJust\ (Just\ (f_T\ t)) = f_T = \textbf{RHS}
\end{array}
$$

We also need to check that $(construct_T, match_T)$ is a virtual constructor as defined in the previous section. The construction constraint holds straightforwardly:

$$
\begin{array}{l}
\textbf{LHS} = match_T\ .\ construct_T \\
= \lambda\ t \rightarrow match_T\ (construct_T\ t) \\
= \lambda\ t \rightarrow openDictLookup\ wit_T\ (openDictFromList \\
\quad [MkAny\ wit_T\ t, MkAny\ wit_f\ (f_T\ t)] \\
) \\
= \lambda\ t \rightarrow Just\ t = Just = \textbf{RHS}
\end{array}
$$

The uniqueness constraint also holds, but only because we cleverly hid wit_T inside $MM3$. For $match_T\ d$ to match, $d :: D$ must contain an entry for wit_T. But since wit_T is hidden, the only way to create such a D is by using $construct_T$.

$$
\textbf{LHS} = fmap\ construct_T\ (match_T\ d)
$$

If d does not have a wit_T:

$$
\begin{array}{l}
= fmap\ construct_T\ (openDictLookup\ wit_T\ d) \\
= fmap\ construct_T\ Nothing = Nothing = \textbf{RHS}
\end{array}
$$

If d does have a wit_T, then we must have been created by $construct_T$. So there must be some t such that $d = construct_T\ t$.

$$
\begin{array}{l}
= fmap\ construct_T\ (match_T\ (construct_T\ t)) \\
= fmap\ construct_T\ (Just\ t)) \\
= Just\ (construct_T\ t) = Just\ d = \textbf{RHS}
\end{array}
$$

4.5 COM-Style Interfaces

We can use open witness declarations to implement a style of OO programming similar to Microsoft's Component Object Model:

- there's a single type that any object can be given
- objects can be defined to implement *interfaces* (set of functions)
- given such an object, one can *query* it to find out whether it supports a given interface
- new interfaces can be defined

For a Haskell implementation, an interface might typically be a datatype with a list of *member functions*. However, we will allow any type to be an interface.

$$
\begin{array}{l}
\textbf{data } IDrawable = MkIDrawable \\
\quad iDrawableBoundsRect :: IORef\ (Int, Int, Int, Int), \\
\quad iDrawableDraw :: Graphics \rightarrow IO\ ()
\end{array}
$$

Our strategy will be to declare a witness for each interface definition.

$$
\begin{array}{l}
iDrawableWitness :: \\
\quad IOWitness\ IDrawable \leftarrow newIOWitness
\end{array}
$$

We want to present a corresponding $queryInterface$ function to query objects for interfaces. One difference from COM is that our interface types are purely that: they provide no access to any underlying object and so cannot be used as an argument to $queryInterface$. Since our base object type is not an interface, we call it $Unknown$ instead of $IUnknown$. If we wanted to match COM behaviour more closely, we could correspond the COM interface "IWidget" to the Haskell type $(IWidget, Unknown)$, but here we'll leave that.

This is straightforward to implement: $Unknown$ is simply $IOOpenDict$:

$$
\begin{array}{l}
\textbf{type } Unknown = IOOpenDict \\
queryInterface :: \\
\quad IOWitness\ i \rightarrow Unknown \rightarrow Maybe\ i \\
queryInterface = openDictLookup
\end{array}
$$

We shall also need a function to construct objects from interfaces:

$$
\begin{array}{l}
newUnknown :: [Any\ IOWitness] \rightarrow Unknown \\
newUnknown = openDictFromList
\end{array}
$$

For example, consider a checkbox control for a user interface, for which we want to provide three interfaces.

- $IDrawable$
- $IClickable$
- $IBooleanState$ (checkbox is either checked or unchecked)

Those interfaces come with corresponding witnesses:

- $iDrawableWitness$
- $iClickableWitness$
- $iBooleanStateWitness$

We should already know how to implement the interfaces for our object, and we can package them together into an $Unknown$ using $newUnknown$:

$$
\begin{array}{l}
newCheckBox :: IO\ Unknown \\
newCheckBox = \textbf{do} \\
\quad return\ \$\ newUnknown \\
\qquad [MkAny\ iDrawableWitness\ drawable, \\
\qquad MkAny\ iClickableWitness\ clickable, \\
\qquad MkAny\ iBooleanState\ booleanState]
\end{array}
$$

4.6 Prototype-Based OO

Prototypes are an approach to object-oriented programming that erases the boundary between classes and objects. Instead of classes, any object can act as a *prototype* for creating similar objects. It's a very dynamic sort of typing, so we'll have to do most everything in the IO monad.

- A single *clone* operation replaces these operations from class-based OO idioms:
  - creating a new instance (class → object)
  - subtyping (class → class)
  - cloning (object → object).
- New fields and methods can be added to existing objects, which can be looked up by name.
- New empty objects can be created.

For our implementation of prototypes in Haskell, we don't distinguish fields and methods: they are both simply *members*, a method being just a member that happens to have a function type. Members of objects are referred to by *member name*, and these are typed to match the type of the member. Our objects are mutable dictionaries.

```
type PObject = IORef IOOpenDict
```

We use $IOWitness$ values for member names, each declared with a top-level call to $newIOWitness$. When applied to an object, member names act as keys to a dictionary holding the state of the object.

```
type PName = IOWitness
```

Since objects in prototype-based programming are mutable, regardless of implementation our Haskell equivalents cannot be constructed, they can only by created within our execution monad. Creating new empty objects is straightforward, we simply create a reference containing an empty dictionary.

```
newPObject :: IO PObject
newPObject = newIORef emptyOpenDict
```

Likewise, cloning an object is no more than copying its state:

```
clonePObject :: PObject → IO PObject
clonePObject pobj = do
  state ← readIORef pobj
  newIORef state
```

Reading and writing member is also straightforward. $lookupMember$ looks up the name in the dictionary. $readMember$ does the same thing, but fails if the method is not found.

```
lookupMember ::
  ∀ a. PName a → PObject → IO (Maybe a)
lookupMember member object = do
  dict ← readIORef object
  return (openDictLookup member dict)

readMember ::
  ∀ a. PName a → PObject → IO a
readMember member object = do
  ma ← lookupMember member object
  case ma of
     Just a → return a
     Nothing → fail "member not found"
```

Our $writeMember$ function is also used to add new members to objects.

```
writeMember ::
  ∀ a. PName a → a → PObject → IO ()
writeMember member val object = do
  dict ← readIORef object
  writeIORef (openDictAdd member val dict)
```

Invoking member functions must be done in the IO monad, since members are mutable in objects, and we run the risk that the member isn't in the object. Member functions must generally include an argument for the object itself, so that when an object is cloned, the method is used with the new object rather than the old. To simplify method invocation, we can create an idiom for methods, that their types should have a particular form:

```
type PMethod a r = PObject → a → IO r
```

We provide a function to make invocation slightly simpler:

```
invoke :: ∀ a r. PName (PMethod a r) → PMethod a r
invoke name object args = do
  m ← readMember name object
  m object args
```

While the hierarchies of class-based idioms model strict IS-A relationships, prototypes are good for more vague IS-LIKE-A relationships. For instance, an ellipse *is like a* rectangle. Here we first create a prototype rectangle:

```
wit_bounds :: PName (Int, Int, Int, Int) ←
  newIOWitness
wit_draw :: PName (PMethod Drawing.Graphics ()) ←
  newIOWitness

rectangleDraw :: PMethod Drawing.Graphics ()
rectangleDraw obj graphics = do
  (left, top, right, bottom) ← readMember wit_bounds obj
  Drawing.drawRect graphics left top right bottom

makeRectanglePrototype :: IO PObject
makeRectanglePrototype = do
  rectangleProt ← newPObject
  writeMember wit_bounds (0, 0, 100, 100) rectangleProt
  writeMember wit_draw rectangleDraw rectangleProt
  return rectangleProt
```

Then we clone it and modify the clone to make a prototype ellipse.

```
ellipseDraw :: PMethod Drawing.Graphics ()
ellipseDraw obj graphics = do
  (left, top, right, bottom) ← readMember wit_bounds obj
  Drawing.drawEllipse graphics left top right bottom

makeEllipsePrototype :: PObject → IO PObject
makeEllipsePrototype rectangleProt = do
  ellipseProt ← clonePObject rectanglePrototype
  writeMember wit_draw ellipseDraw ellipseProt
  return ellipseProt
```

Finally we can use these prototypes to create instances (which are, in fact, just clones):

```
makeShape ::
  PObject → (Int, Int, Int, Int) → IO PObject
makeShape prototype bounds = do
  shape ← clonePObject bounds
  writeMember wit_bounds bounds shape
  return shape

main = do
  rectangleProt ← makeRectanglePrototype
  ellipseProt ← makeEllipsePrototype rectangleProt
  myCircle ← makeShape ellipseProt (50, 200, 30, 30)
  myRectangle ←
     makeShape rectangleProt (80, 200, 60, 30)
  graphics ← Drawing.newWindow
  invoke wit_draw graphics myCircle
  invoke wit_draw graphics myRectangle
  ...
```

4.7 Thread-Local Storage

Peyton-Jones [10] suggests a language extension for *thread-local storage*. It consists of a new top-level declaration, **newkey**, and two functions, $withBinding$ and $lookupBinding$.

```
newkey ⟨identifier⟩ :: Key ⟨type⟩
withBinding :: Key a → a → IO b → IO b
lookupBinding :: Key a → IO a
```

Our open witnesses extension cannot do thread-local storage by itself, but by doing the Key work of dynamic typing, it can reduce the necessary API to a single function that gets a single thread-local object, an $IORef$ to an $IOOpenDict$.

$$lookupDict :: IO\ (IORef\ IOOpenDict)$$

This $IORef$ is initialised at thread creation with an empty dictionary:

$$newIORef\ emptyOpenDict$$

We can then implement the suggested thread-local extension. A Key is simply an $IOWitness$.

$$\textbf{type}\ Key = IOWitness$$

And top-level **newkey** declarations become top-level $IOWitness$ declarations. Thus

$$\textbf{newkey}\ identifier :: Key\ type$$

becomes

$$identifier :: Key\ type \leftarrow newIOWitness$$

The $lookupBinding$ function calls $lookupDict$ to fetch the key:

```
lookupBinding key = do
  dictref ← lookupDict
  dict ← readIORef dictref
  return (unJust (openDictLookup key dict))
```

The $withBinding$ function executes a function with a new binding added to the binding dictionary. It then restores the old dictionary when it's finished.

```
withBinding key a foo = do
  dictref ← lookupDict
  bracket
    (do
      dict ← readIORef
      writeIORef dictref (openDictAdd key a dict)
      return dict
    )
    (writeIORef dictref)
    (const foo)
```

5. Related work

Several different approaches have been proposed to solve the expression problem in Haskell:

- $Data.Typeable$ [1] is a popular solution to this problem, available in the standard libraries. But it is unsafe (section 4.2), and therefore ugly: by writing an instance of the $Typeable$ class, it's easy to write $unsafeCoerce$.
- Weirich [17] presents RepLib, a library for representing the internal structure of types. But that means breaking their encapsulation.
- Löh and Hinze [7] offer an extension to Haskell of open data types and open functions. This is clean and intuitive to use, powerful, and safe, but it involves a translation from their extended Haskell to existing Haskell that requires modules to be compiled together.
- Seefried and Chakravarty [13] also offer an extension to Haskell of open data types and open functions, that allows separate compilation, but with a translation that is considerably more complex.
- Swierstra [16] has a scheme where types can be easily constructed from a given list of variants. But new variants cannot be added to existing monomorphic types.

6. Further work

6.1 Multiple Dispatch

In section 4.4, we showed how to do *single dispatch*, that is, create a function on a single open type that can be defined for new variants.

The programming language Dylan allows *multiple dispatch*, that is, functions that dispatch to particular methods based on the type of more than one argument. This is also a notable feature of the Haskell extension proposed by Löh and Hinze [7]. Can this be done in Haskell with open witness declarations?

6.2 Top-Level Declarations

The mechanism we proposed to declare open witnesses at top level is to call one particular IO function ($newIOWitness$) as a static initialiser. This could be generalised to declare top-level $MVar$s, $IORef$s, and so on. Care needs to be taken, however, to prevent observation of the order in which initialisers are executed.

On extending Haskell with static initialisers there has been extensive discussion on the Haskell mailing list since at least October 2004 [8], mostly in the context of global variables. Hey et al. [5] summarise this on the HaskellWiki web site.

Acknowledgments

References

[1] Data.Typeable. Haskell standard library, . URL `http://haskell.org/ghc/docs/latest/html/libraries/base/src/Data-Typeable.html`.

[2] Data.Unique. Haskell standard library, . URL `http://haskell.org/ghc/docs/latest/html/libraries/base/src/Data-Unique.html`.

[3] Language qualities. Haskell-Prime trac wiki. URL `http://hackage.haskell.org/trac/haskell-prime/wiki/LanguageQualities`.

[4] A. I. Baars and S. D. Swierstra. Typing dynamic typing. *SIGPLAN Not.*, 37(9):157–166, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/583852.581494.

[5] A. Hey et al. Top level mutable state. HaskellWiki web site. URL `http://haskell.org/haskellwiki/Top_level_mutable_state`.

[6] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488.

[7] A. Löh and R. Hinze. Open data types and open functions. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 133–144, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3. doi: http://doi.acm.org/10.1145/1140335.1140352.

[8] J. Meacham. Global variables and IO initializers: A proposal and semantics. Haskell mailing list, 2004. URL `http://www.haskell.org/pipermail/haskell/2004-October/014618.html`.

[9] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. September 2002. URL `http://haskell.org/definition/haskell98-report.pdf`.

[10] S. Peyton-Jones. thread-local variables. Haskell mailing list, 2006. URL `http://www.haskell.org/pipermail/haskell/2006-August/018343.html`.

[11] S. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1160074.1159811.

[12] D. Roundy. darcs. source code. URL `http://darcs.net/`.

[13] S. Seefried and M. M. T. Chakravarty. Solving the expression problem in Haskell with true separate compilation. Technical report, University of New South Wales, Sydney, Australia, June 2007.

[14] T. Sheard, J. Hook, and N. Linger. GADTs + extensible kinds = dependent programming. submitted to ICFP, 2005. URL `http://www.cs.pdx.edu/~sheard/papers/GADT+ExtKinds.ps`.

[15] M. Sulzmann and M. Wang. Translating generalized algebraic data types to System F. Manuscript, 2005. URL `http://www.comp.nus.edu.sg/~sulzmann/manuscript/simple-translate-gadts.ps`.

[16] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 2008. doi: 10.1017/S0956796808006758. URL `http://www.cs.nott.ac.uk/~wss/Publications/DataTypesALaCarte.pdf`.

[17] S. Weirich. RepLib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-489-8. doi: http://doi.acm.org/10.1145/1159842.1159844.

[18] A. Yakeley. Two fun things with GADTs. Haskell-cafe mailing list, 2005. URL `http://www.haskell.org/pipermail/haskell-cafe/2005-January/008326.html`.