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 classes. Users of object-oriented languages such as Java and C++ will be familiar with extensible types under the name classes. 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 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.

A number of solutions have been suggested for this problem (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.

matchWitness ::
  Witness a → Witness b → Maybe (EqualType a b)

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).

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

Copyright notice will appear here once ‘preprint’ option is removed.
• **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 single 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:

```
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 \( \text{EqualType} a b \) (an equality witness) that proves the equality of \( a \) and \( b \).

This class provides a function `matchWitness` that accomplishes this:

```
class SimpleWitness w where
  matchWitness :: w a \rightarrow w b \rightarrow \text{Maybe} \text{(EqualType} a b)
```

The class comes with a constraint. To be a correct instance of `SimpleWitness`, all values must match themselves:

```
matchWitness wit wit = \text{Just} \text{MkEqualType}
```

Thus with the appropriate implementation, `MySimpleWitness` would be an instance.

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

\( \text{EqualType} \) (\( \text{Equal} \) 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 \text{EqualType} a b where
  \text{MkEqualType} :: \forall t. \text{EqualType} t t
```

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

```
exampleUnify ::
  \forall a b c. \text{EqualType} a b \rightarrow (a, b \rightarrow c) \rightarrow (b, a \rightarrow c)
```

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

Note that the type arguments to `EqualType` have kind \( \ast \). 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 `EqualType` where possible. For instance, a proof that \( f \) and \( f' \), two type-constructors of kind \( \ast \rightarrow \ast \), are identical, can be represented with a value of type `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 \text{MyPartialWitness} a where
  \text{MPWMaybe} :: \forall p. \text{MyPartialWitness} (\text{Maybe} p)
```

Here `MPWMaybe` witnesses the parameter \( a \) to `Maybe p`, but `p` is left unwitnessed. `MyPartialWitness` cannot be made a correct instance of `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 \text{CharOrInt} a where
  \text{IsChar} :: \text{CharOrInt} \text{Char}
  \text{IsInt} :: \text{CharOrInt} \text{Int}
```

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

```
somechar :: \forall a. \text{CharOrInt} a \rightarrow a
somechar = \lambda \text{Char} \rightarrow \text{‘t’}
```

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

Since each value of `CharOrInt` constrains the type parameter \( a \) to a single type, we can make it an instance of `SimpleWitness`:

```
instance SimpleWitness \text{CharOrInt} where
  matchWitness \text{IsChar} \text{IsChar} = \text{Just} \text{MkEqualType}
  matchWitness \text{IsInt} \text{IsInt} = \text{Just} \text{MkEqualType}
  ...
```

Our example `CharOrInt` covers only two types. However, we can create witness types that witness to a system of types:

```
data \text{MyType} a where
  \text{IsChar} :: \text{MyType} \text{Char}
  \text{IsInt} :: \text{MyType} \text{Int}
  \text{IsMaybe} :: \forall a. \text{MyType} a \rightarrow \text{MyType} (\text{Maybe} a)
  \text{IsList} :: \forall a. \text{MyType} a \rightarrow \text{MyType} [a]
  \text{IsPair} ::
    \forall a. \text{MyType} a \rightarrow \text{MyType} b \rightarrow \text{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 \([\text{Maybe} ([\text{Int}, \text{Char}])]\):

```
exampleMyType :: \text{MyType} [\text{Maybe} ([\text{Int}, \text{Char}])]
exampleMyType = \text{IsList} \text{IsMaybe} \text{IsPair} \text{IsList} \text{IsInt} \text{IsChar}
```

We can write `matchWitness` for `MyType` to make it an instance of `SimpleWitness`:

```
instance SimpleWitness \text{MyType} where
```

---

**Witnesses and Open Witnesses** 2 2008/8/9
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
  return MkEqualType
matchWitness IsChar OrInt = 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:

\[
data \text{Any } w = \forall a. \text{MkAny } (w a) a
\]

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:

\[
data \text{AnyF } w f = \forall a. \text{MkAnyF } (w a) (f a)
\]

Thus we have this isomorphism:

\[
\text{AnyF CharOrInt} :: \equiv \text{Either Char Int}
\]

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:

\[
data \text{NumInst } a where
  \text{MkNumInst } :: \forall a. \text{Num } a \Rightarrow \text{NumInst } a
\]

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:

\[
(+): \forall a. \text{Num } a \Rightarrow a \Rightarrow a \
\text{fromInteger } :: \forall a. \text{Num } a \Rightarrow \text{Integer } \rightarrow a
\]

And here they are rewritten:

\[
\text{plus}' :: \forall a. \text{NumInst } a \Rightarrow a \Rightarrow a \
\text{fromInteger}' :: \forall a. \text{NumInst } a \Rightarrow \text{Integer } \rightarrow a
\]

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

\[
\text{intNum } :: \text{NumInst Int} \
\text{intNum } = \text{MkNumInst}
\]

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 (\forall f. f a \rightarrow 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:

\[
\text{toEqual } :: \text{EqualType } a b \rightarrow \text{Equal } a b \
\text{toEqual } \text{MkEqualType } = \text{Equal id} \
\text{fromEqual } :: \text{Equal a b } \rightarrow \text{EqualType } a b
\]

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:

\[
data \text{CharOrInt' } a =\] IsChar (\forall f. f a \rightarrow f \text{ Char}) \
| IsInt (\forall f. f a \rightarrow f \text{ 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.

\[
\begin{array}{ll}
\text{simple witness} & \text{representative} \\
\text{value } \rightarrow \text{type} & \text{type } \rightarrow \text{value}
\end{array}
\]

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

\[
\text{class Is rep a where} \\
\text{representative } :: \text{rep a}
\]

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

\[
\begin{array}{ll}
\text{instance IsCharOrInt Char where} \\
\text{representative } = \text{IsChar} \\
\text{instance IsCharOrInt Int where} \\
\text{representative } = \text{IsInt}
\end{array}
\]

The simplest representative is the universal representative, Type.

\[
data \text{Type } a = \text{MkType}
\]

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.

\[
\text{class Storable } a \text{ where} \\
\text{sizeOf } :: \text{Type } a \rightarrow \text{Int}
\]

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

\[
\begin{array}{ll}
\text{instance IsType } a \text{ where} \\
\text{representative } = \text{MkType}
\end{array}
\]

2.7 HList-Style List Types

HList [6] provides strongly-typed heterogeneous 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.

\[
data \text{ListType } w a \text{ where} \\
\text{IsHNil } :: \text{ListType } w \text{ HNil} \\
\text{IsHCons } :: \text{w e } \rightarrow \text{ListType } w l \rightarrow \text{ListType } w (\text{HCons } e l)
\]

\[
\text{instance SimpleWitness } w \Rightarrow
\]

Witnesses and Open Witnesses
The type may also have representatives:

\[
\text{instance } \text{Is} \ (\text{ListType } w) \ \text{HNil} \ \text{where} \\
\text{representative } = \ \text{IsHNil} \\
\text{instance } (\text{Is } w e, \text{Is} \ (\text{ListType } w) \ l) \Rightarrow \\
\text{Is} \ (\text{ListType } w) (\text{HCons } e \ l) \\
\text{where} \\
\text{representative } = \\
\text{HCons representative representative}
\]

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{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, that can witness any type. However, they cannot be constructed: they can only be generated in certain constructors. We now introduce a type for a variety of simple witnesses.

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

\[
\text{charsInts} :: \ \text{Any} \ (\text{ListType} \ \text{CharOrInt}) \\
\text{charsInts} = \\
\text{MkAny representative} (\text{HCons 3 (HCons HNil ‘a’)})
\]

The \text{type Any} (\text{ListType} \ \text{CharOrInt}) contains \text{HList} values where each element is either a \text{Char} or an \text{Int}.
Witnesses and Open Witnesses.

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

\[
\begin{align*}
\text{owToIO} & : \text{OW} \text{ RealWorld } a \rightarrow \text{IO} \ a \\
\text{owToIO} \ (\text{MkOW } st) &= \\
\ & \text{modifyMVar } \text{mioWitnessSource } (\lambda \ a \rightarrow \\
& \text{let } \ (a, \text{count}) = \text{runState } st \text{ start } \\
& \text{in } \\
& \return \ (\text{count}, \ a) \\
\end{align*}
\]

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
  openDictAdd :: OpenWitness s a → OpenDict s → Maybe a
  emptyOpenDict :: OpenDict s
  openDictFromList :: [Any (OpenWitness s)] → OpenDict s
  openDictAdd :: OpenWitness s a → a → OpenDict s
  openDictModify :: OpenWitness s a → (a → a) → OpenDict s
  openDictReplace :: OpenWitness s a → a → 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 → a → 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 (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 (aa : aa) = case f a of
  Just newa → (newa : aa)
  _ → a : (replaceFirst f aa)
replaceFirst aa = []

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

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 OW emptyWitnessDict
runST :: (forall 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 ::forall s a. STRef s a → a → ST s ()
writeSTRef key a =
  modify (openDictReplace key a)
```

```
∀ s a. STRef s a → (a → a) → ST s ()
modifySTRef key f = (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.

(identifier) :: IOWitness (type) ← newIOWitness

This extension might be generalised to allow other top level things such as MVars, 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 ← 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, foralls, be they explicit or implicit, are not allowed outside the IOWitness.

w1 :: IOWitness Int ← newIOWitness
     — OK
w2 :: IOWitness (forall a. IO a) ← newIOWitness
     — OK if impredicativity is allowed
w3 :: (forall a. IOWitness (IO a) ← newIOWitness
     — prohibited, IOa is not closed
w4 :: IOWitness (IO a) ← 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 ← newIOWitness

becomes

identifier :: type
{¬ # 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 ← newIOWitness

becomes

pat :: type = MkOpenWitness ((toInteger (hashString "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:

newtype Thing a = MkThing {unThing :: a}

instance Typeable (Thing a) where
typeOf _ = typeOf ()

unsafeCoerce :: a → b
unsafeCoerce a = 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:

class Typeable a where
    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:

data TypeRep t where
    SimpleTypeRep :: IOWitness t → TypeRep t
    TypeRep1 p → TypeRep a → TypeRep (p a)

And here is our Typeable class:

class Typeable a where
    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. It is a simple witness type:

instance SimpleWitness TypeRep where
    matchWitness
    (SimpleTypeRep wa) (SimpleTypeRep wb) = matchWitness wa wb
    (ApplyTypeRep tfa ta) (ApplyTypeRep tfb tb) = do
        MkEqualType ← matchTypeRep1 tfa tfb
        MkEqualType ← matchWitness ta tb
        return MkEqualType
    matchWitness _ _ = Nothing

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

cast :: ∀ a b. (Typeable a, Typeable b) ⇒ a → Maybe b
cast a = do
    MkEqualType ← EqualType a b ← matchWitness rep rep
               return a
We still have TypeRep1 to define, to witness types of kind \( \star \rightarrow \star \). Our scheme obliges us to choose a finite set of kinds, and define a TypeRepX type for each one. For simplicity, we’ll pick the set \{ \*, \star \rightarrow \*, \star \rightarrow \star \}. We do not include, for instance, \( \star \rightarrow \star \), though this is the kind of our Any type.

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

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

witChar :: IOWitness Char \leftarrow newIOWitness
instance Typeable a \Rightarrow Typeable [a] where
  rep = ApplyTypeRep
         (SimpleTypeRep1 witChar)

witInt :: IOWitness Int \leftarrow newIOWitness
instance Typeable Int where
  rep = SimpleTypeRep witInt
— etc.

...and the higher kinds:

witList :: IOWitness [()] \leftarrow newIOWitness
instance Typeable a \Rightarrow Typeable [a] where
  rep = ApplyTypeRep
         (SimpleTypeRep1 witList)

witFn :: IOWitness (() () ) \leftarrow newIOWitness
instance (Typeable a, Typeable b) \Rightarrow
  Typeable (a -> b) where
  rep = ApplyTypeRep
         (ApplyTypeRep1
          (SimpleTypeRep2 witFn)
           rep)
— etc.

The Dynamic type is easy to define:

type Dynamic = Any TypeRep

toDyn :: Typeable a \Rightarrow Dynamic
toDyn a = MkAny representative a
fromDynamic :: Typeable a \Rightarrow Dynamic \rightarrow Maybe a
fromDynamic (MkAny wit a) = do
  MkEqualType \leftarrow matchWitness wit representative
  return a
fromDyn :: Typeable a \Rightarrow Dynamic \rightarrow 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 \rightarrow Dynamic \rightarrow Maybe Dynamic
dynApply
  (MkAny (ApplyTypeRep
            (ApplyTypeRep1 (SimpleTypeRep2 witFn') rx')
             ry)) f)
  (MkAny rx x)
= do
  MkEqualType \leftarrow matchWitness witFn witFn'
  MkEqualType \leftarrow 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 \( D \rightarrow \text{Maybe} \ T \). A virtual constructor, then, is simply a pair of functions we call construct and match.

construct :: T \rightarrow D
match :: D \rightarrow \text{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 (or) Nothing

What we want is an extended 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 \( D \).
import M1
import (elsewhere)(T)

\[
\text{wit}_T :: IOWitness T \leftarrow \text{newIOWitness} \\
\text{construct}_T :: T \rightarrow D \\
\text{construct}_T \equiv \text{ MkAny wit}_T t \\
\text{match}_T :: D \rightarrow \text{Maybe T} \\
\text{match}_T (\text{MkAny wit } x) = \text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } t
\]

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

\[
\text{LHS} = \text{match}_T, \text{construct}_T \\
= \lambda t \rightarrow \text{match}_T (\text{construct}_T t) \\
= \lambda t \rightarrow \text{match}_T (\text{MkAny wit}_T t) \\
= \lambda t \rightarrow \text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } t
\]

And the uniqueness constraint:

\[
d = \text{MkAny wit } x \\
\text{LHS} = \text{fmap construct (match d)} \\
= \text{fmap construct}_T (\text{match}_T (\text{MkAny wit } x)) \\
= \text{fmap construct}_T \\
(\text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } x) \\
= \text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } (\text{construct}_T x) \\
= \text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } (\text{MkAny wit}_T x) \\
= \text{Nothing (or if wit = wit}_T) \text{do} \\
\text{MkEqualType} \leftarrow \text{matchWitness wit}_T t \\
\text{return } (\text{MkAny wit } x) \\
= \text{Nothing (or) Just } d = \text{RHS}
\]

4.4 The Expression Problem

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

1. \text{define type } D
2. \text{given type } T, \text{extend } D \text{ with variant on } T: \\
\text{construct}_T :: T \rightarrow D \\
\text{match}_T :: D \rightarrow \text{Maybe T}
3. \text{given type } R, \text{declare function } f \text{ of type } D \rightarrow R
4. \text{given function } f_T :: T \rightarrow R, \text{define result of } f . \text{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 2, defining the result when we define the function, and define modules \( M1, M2, M3 \) each importing the previous modules:

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

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

\[
\text{module } M3 \text{ where} \\
\text{import } M1 \\
\text{import } M2 \\
\text{import (elsewhere)}(R, f_T) \\
f :: D \rightarrow R \\
f d = \text{case } \text{match}_T d \text{ of} \\
\text{Just } t \rightarrow f_T t \\
\text{Nothing } \rightarrow \text{undefined}
\]

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 \( f :: D \rightarrow R \) such that \( f . \text{construct}_T = f_T \).

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 \( D \):

\[
\text{data } D = \text{MkD} \\
\{ \\
\text{variant :: Any IOWitness,} \\
\text{f :: R} \\
\}\n\]

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.

\[
\text{module } MM1 \text{ where} \\
\text{type } D = \text{IOOpenDict} \\
\]

In \( MM2 \), we create a witness \( \text{wit}_f \) for \( f \), and define \( f \) to look up the witness in its \( D \) argument’s method table. We don’t care if it returns \text{undefined} if the witness isn’t there.

\[
\text{module } MM2 \text{ where} \\
\text{import } MM1 \\
\text{import (elsewhere)}(R) \\
\text{wit}_f :: IOWitness R \leftarrow \text{newIOWitness} \\
f :: D \rightarrow R \\
f d = \text{unJust (openDictLookup wit}_f d)
\]

For \( MM3 \), we’re given a type \( T \) and a method function \( f_T \) of type \( T \rightarrow R \). Our \( \text{construct}_T \) function creates a \( D \) with a single entry in its method table, that is, \( f_t \) for key \( \text{wit}_f \).

\[
\text{module } MM3 (\text{construct}_T, \text{match}_T) \text{ where}
\]
import MM1
import MM2
import (elsewhere)(T, fT)

witT :: IOWitness T ← newIOWitness
constructT :: T → D
constructT t = openDictFromList
   [ MkAny witT t, MkAny witT (fT t)]
matchT :: D → Maybe T
matchT d = openDictLookup witT d

Let’s check that \( f \cdot \text{construct}_T = f_T \):
\[
LHS = f \cdot \text{construct}_T = \lambda t \to f (\text{construct}_T t)
= \lambda t \to \text{unJust} (\text{openDictLookup} \ \text{witT} (\text{openDictFromList} [\text{MkAny witT} t, \text{MkAny witT} (fT t)])
)
= \lambda t \to \text{unJust} (\text{Just} (f_T t)) = f_T = \text{RHS}
\]

We also need to check that \((\text{construct}_T, \text{match}_T)\) is a virtual constructor as defined in the previous section. The construction constraint holds straightforwardly:
\[
\begin{align*}
\text{LHS} &= \text{match}_T \cdot \text{construct}_T \\
&= \lambda t \to \text{match}_T (\text{construct}_T t) \\
&= \lambda t \to \text{openDictLookup} \ \text{witT} (\text{openDictFromList} [\text{MkAny witT} t, \text{MkAny witT} (fT t)])
\end{align*}
\]
\[
= \lambda t \to \text{Just} \ t = \text{Just} = \text{RHS}
\]

The uniqueness constraint also holds, but only because we cleverly hid \( \text{witT} \) inside \( \text{MM3} \). For \( \text{match}_T d \) to match, \( d :: D \) must contain an entry for \( \text{witT} \). But since \( \text{witT} \) is hidden, the only way to create such a \( D \) is by using \( \text{construct}_T \).
\[
\text{LHS} = f \cdot \text{match}_T \cdot \text{construct}_T\]
\[
\text{If } d \text{ does not have a } \text{witT}: \\
= \text{fmap} \ \text{construct}_T (\text{openDictLookup} \ \text{witT} \ d) \\
= \text{fmap} \ \text{construct}_T \ \text{Nothing} = \text{Nothing} = \text{RHS}
\]
\[
\text{If } d \text{ does have a } \text{witT}, \text{then we must have been created by } \text{construct}_T. \text{So there must be some } t \text{ such that } d = \text{construct}_T t.
\]
\[
= \text{fmap} \ \text{construct}_T (\text{match}_T (\text{construct}_T t)) \\
= \text{fmap} \ \text{construct}_T (\text{Just} t) \\
= \text{Just} (\text{construct}_T t) = \text{Just} d = \text{RHS}
\]

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.

\[
data \text{IDrawable} = \text{MkIDrawable} \text{idDrawableBoundRect :: IORef (Int, Int, Int, Int), idDrawableDraw :: Graphics } → \text{IO} ()
\]

Our strategy will be to declare a witness for each interface definition.
\[
\text{idDrawableWitness ::}
\]

We want to present a corresponding \text{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 \text{queryInterface}. Since our base object type is not an interface, we call it Unknown instead of \text{IUnknown}. If we wanted to match COM behaviour more closely, we could correspond the COM interface “IWidget” to the Haskell type \text{(IWidget, Unknown)}, but here we’ll leave that.

This is straightforward to implement: Unknown is simply \text{IOpenDict}:
\[
\text{type Unknown} = \text{IOpenDict} \\
\text{queryInterface ::} \\
\text{IOWitness i → Unknown → Maybe i} \\
\text{queryInterface} = \text{openDictLookup}
\]

We shall also need a function to construct objects from interfaces:
\[
\text{newUnknown :: \lbrack Any IOWitness\rbrack → Unknown} \\
\text{newUnknown} = \text{openDictFromList}
\]

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

- \text{IDrawable}
- \text{IClickable}
- \text{IBoolBooleanStateWitness} (checkbox is either checked or unchecked)

Those interfaces come with corresponding witnesses:

- \text{idDrawableWitness}
- \text{iClickableWitness}
- \text{iBooleanStateWitness}

We should already know how to implement the interfaces for our object, and we can package them together into an Unknown using \text{newUnknown}:
\[
\text{newCheckBox :: IO Unknown} \\
\text{newCheckBox} = \text{do} \\
\text{return } \text{Just} \ \text{newUnknown} \\
\text{[MkAny idDrawableWitness drawable,} \\
\text{MkAny iClickableWitness clickable,} \\
\text{MkAny iBooleanStateWitness booleanState]}
\]

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 \text{IO} monad.

- A single \text{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 \text{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.

\[
\text{type } \text{PObj} = \text{IORef } \text{IOOpenDict}
\]

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

\[
\text{type } \text{Name} = \text{IOWitness}
\]

Since objects in prototype-based programming are mutable, regard-
less of implementation our Haskell equivalents cannot be con-
structed, they can only be created within our execution monad. Cre-
ating new empty objects is straightforward, we simply create a ref-

\[
\text{new} \text{PObj} :: \text{PObj}
\]

Structures, they can only be created within our execution monad. Cre-
ating new empty objects is straightforward, we simply create a ref-

\[
\text{new} \text{PObj} = \text{newIORef } \text{emptyOpenDict}
\]

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

\[
\text{clone} \text{PObj} :: \text{PObj} \rightarrow \text{IO } \text{PObj}
\]

\[
\text{clone} \text{PObj } \text{pobj} = \text{do}
\]

\[
\text{state } \leftarrow \text{readIORef } \text{pobj}
\]

\[
\text{new} \text{IORef } \text{state}
\]

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

\[
\text{lookupMember } ::
\forall \ a. \ \text{Name} \ a \rightarrow \text{PObj} \rightarrow \text{IO } \text{(Maybe } a)\]

\[
\text{lookupMember } \text{member } \text{object } = \text{do}
\]

\[
\text{dict } \leftarrow \text{readIORef } \text{object}
\]

\[
\text{return } \{ \text{openDictLookup } \text{member } \text{dict} \}
\]

\[
\text{readMember } ::
\forall \ a. \ \text{Name} \ a \rightarrow \text{PObj} \rightarrow \text{IO } a\]

\[
\text{readMember } \text{member } \text{object } = \text{do}
\]

\[
\text{ma } \leftarrow \text{lookupMember } \text{member } \text{object}
\]

\[
\text{case } \text{ma } \text{of}
\]

\[
\text{Just } a \rightarrow \text{return } a
\]

\[
\text{Nothing } \rightarrow \text{fail } \text{"member not found"}
\]

Our \text{writeMember} function is also used to add new members to
objects.

\[
\text{writeMember } ::
\forall \ a. \ \text{Name} \ a \rightarrow \ a \rightarrow \text{PObj} \rightarrow \text{IO } ()\]

\[
\text{writeMember } \text{member } \text{val } \text{object } = \text{do}
\]

\[
\text{dict } \leftarrow \text{readIORef } \text{object}
\]

\[
\text{writeIORef } \{ \text{openDictAdd } \text{member } \text{val } \text{dict} \}
\]

Invoking member functions must be done in the \text{IO} monad, since
members are mutable in objects, and we run the risk that the mem-
ber 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:

\[
\text{type } \text{Method } a \rightarrow r \rightarrow \text{PObj} \rightarrow a ightarrow \text{IO } r
\]

We provide a function to make invocation slightly simpler:

\[
\text{invoke } :: \forall \ a. \ r. \ \text{Name } (\text{Method } a \rightarrow r) \rightarrow \text{Method } a \rightarrow r
\]

\[
\text{invoke } \text{name } \text{object } \text{args } = \text{do}
\]

\[
\text{m } \leftarrow \text{readMember } \text{name } \text{object}
\]

\[
\text{m } \text{object } \text{args}
\]

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

\[
\text{withRectangles } :: \text{Name } (\text{Int}, \text{Int}, \text{Int}, \text{Int}) \rightarrow \text{IOWitness}
\]

\[
\text{withRectangles } = \text{new} \text{IOWitness}
\]

\[
\text{rectangleDraw } :: \text{Name } (\text{Method } \text{Drawing.Graphics }) \rightarrow (\text{left}, \text{top}, \text{right}, \text{bottom}) \rightarrow \text{readMember } \text{withRectangles } \text{object}
\]

\[
\text{Drawing.drawRect } \text{graphics } \text{left } \text{top } \text{right } \text{bottom}
\]

\[
\text{makeRectanglePrototype } :: \text{IO } \text{PObj}
\]

\[
\text{makeRectanglePrototype } = \text{do}
\]

\[
\text{rectangleProt } \leftarrow \text{new} \text{PObj}
\]

\[
\text{writeMember } \text{withRectangles } (0, 0, 100, 100) \text{rectangleProt}
\]

\[
\text{writeMember } \text{withRectangles } \text{rectangleDraw } \text{rectangleProt}
\]

\[
\text{return } \text{rectangleProt}
\]

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

\[
\text{ellipseDraw } :: \text{Name } (\text{Method } \text{Drawing.Graphics }) \rightarrow \text{IO Witness}
\]

\[
\text{ellipseDraw } \text{obj } \text{graphics } = \text{do}
\]

\[
(\text{left}, \text{top}, \text{right}, \text{bottom}) \rightarrow \text{readMember } \text{withRectangles } \text{obj}
\]

\[
\text{Drawing.drawEllipse } \text{graphics } \text{left } \text{top } \text{right } \text{bottom}
\]

\[
\text{makeEllipsePrototype } :: \text{IO } \text{PObj}
\]

\[
\text{makeEllipsePrototype } = \text{do}
\]

\[
\text{ellipseProt } \leftarrow \text{clone} \text{PObj } \text{rectanglePrototype}
\]

\[
\text{writeMember } \text{withRectangles } \text{ellipseDraw } \text{ellipseProt}
\]

\[
\text{return } \text{ellipseProt}
\]

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

\[
\text{makeShape } ::
\text{PObj } \rightarrow (\text{Int}, \text{Int}, \text{Int}, \text{Int}) \rightarrow \text{IO } \text{PObj}
\]

\[
\text{makeShape } \text{prototype } \text{bounds } = \text{do}
\]

\[
\text{shape } \leftarrow \text{clone} \text{PObj } \text{prototype}
\]

\[
\text{writeMember } \text{withRectangles } \text{bounds } \text{shape}
\]

\[
\text{return } \text{shape}
\]

\[
\text{main } = \text{do}
\]

\[
\text{rectangleProt } \leftarrow \text{makeRectanglePrototype}
\]

\[
\text{ellipseProt } \leftarrow \text{makeEllipsePrototype } \text{rectangleProt}
\]

\[
\text{myCircle } \leftarrow \text{makeShape } \text{ellipseProt } (50, 200, 30, 30)
\]

\[
\text{myRectangle } \leftarrow \text{makeShape } \text{rectangleProt } (80, 200, 60, 30)
\]

\[
\text{graphics } \leftarrow \text{Drawing.newWindow}
\]

\[
\text{invoke } \text{withDraw } \text{graphics } \text{myCircle}
\]

\[
\text{invoke } \text{withDraw } \text{graphics } \text{myRectangle}
\]

4.7 Thread-Local Storage

Peyton-Jones [10] suggests a language extension for thread-local
storage. It consists of a new top-level declaration, \text{newkey}, and
two functions, with\text{Binding} and \text{lookup\text{Binding}}.

\[
\text{newkey } \text{(identifier)} : = \text{key } \text{(type)}
\]

\[
\text{withBinding } : = \text{key } \rightarrow \rightarrow \rightarrow \text{IO } b \rightarrow \text{IO } b
\]

\[
\text{lookupBinding } : = \text{key } \rightarrow \text{IO } a
\]

Our open witnesses extension cannot do thread-local storage by
itself, but by doing the \text{Key} work of dynamic typing, it can reduce
the necessary API to a single function that gets a single thread-local object, an IORref to an IOWOpenDict.

\[\text{lookupDict} :: \text{IO} (\text{IORref IOWOpenDict})\]

This IORref is initialised at thread creation with an empty dictionary:

\[\text{newIORref emptyOpenDict}\]

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

\[\text{type} \text{Key} = \text{IOWitness}\]

And top-level newKey declarations become top-level IOWitness declarations. Thus

\[\text{newKey identifier :: Key type}\]

becomes

\[\text{identifier :: Key type} \leftarrow \text{newIOWitness}\]

The \text{lookupBinding} function calls \text{lookupDict} to fetch the key:

\[\text{lookupBinding key = do}\]
\[\text{dictref ← lookupDict}\]
\[\text{dict ← readIORref dictref}\]
\[\text{return (unJust (openDictLookup key dict))}\]

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.

\[\text{withBinding key a foo = do}\]
\[\text{dictref ← lookupDict}\]
\[\text{dict ← readIORref}\]
\[\text{do}\]
\[\text{writeIORref dictref (openDictAdd key a dict)}\]
\[\text{return dict}\]
\[\text{(const foo)}\]

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 \text{(newIOWitness)} as a static initialiser. This could be generalised to declare top-level MVars, IORrefs, 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


Witnesses and Open Witnesses

11

2008/69


