A survey of O'Haskell

Introduction

This document provide a detailed, but informal survey of O'Haskell and its characteristic features. A formal semantic treatment of the language can be found in [1]; in this survey the exposition will instead be based on short code fragments and discussions around these.

The rest of this document is organized as follows. Section 2 continues with a brief overview of the base language Haskell and its syntax, before we introduce the major type system additions of O'Haskell: records and subtyping (sections 3 and 4). In section 5 our approach to automatic type inference in O'Haskell is presented. Reactive objects, concurrency, and encapsulated state are the issues discussed in section 6. Section 7 presents some additional syntactic enhancements that O'Haskell provides, before the survey ends with a section discussing the role of *inheritance* in O'Haskell programming (section 8).

Haskell

Haskell is the quintessential non-strict, purely functional language, and the base upon which O'Haskell is built. While we suspect that some previous acquaintance with Haskell might be necessary in order to get the most out of the present survey, we will nevertheless take some steps to familiarize the reader with the syntax of Haskell in this section.

Functions

Functions are the central concept in Haskell. Applying a function to its arguments is written as a simple juxtaposition; that is, if `f` is a function taking three integer arguments, then

```
f 7 13 0
```

is an expression denoting the result of evaluating `f` applied to the arguments `7`, `13`, and `0`. If an argument itself is a non-atomic expression, parentheses must be used as delimiters, as in

```
f 7 (g 55) 0
```

Operators like + (addition) and == (test for equality) are also functions, but written between their first two arguments. An ordinary function application always binds more tightly than an operator, thus

```
a b+c d
```

should actually be read as

```
(a b) + (c d)
```

Laziness

Non-strict semantics means that function arguments are only evaluated when absolutely needed by a function (hence the epithet *lazy* is often used to describe non-strict languages). So, even if

```
g 55
```

is a non-terminating or erroneous computation (including for example an attempt to divide by zero), the computation

```
f 7 (g 55) 0
```

will nevertheless succeed in Haskell, provided that `f` happens to be a function which ignores its second argument (whenever the first one is `7`, say). This kind of flexibility can be very useful, not least in the encoding and manipulation of infinite data structures. That said, we would also like to make it clear at this early stage that none of the extensions to Haskell that we put forward here relies on the semantics being non-strict. Thus it is perfectly reasonable to judge the merits of our extensions as if they were intended for an ordinary, strict programming language.

Function definitions

Functions can be defined by equations on the top-level of a program, or locally within a single expression, as in

```
f x y z = let sq i = i * i
          in  sq x * sq y * sq z
```

Note that the single equality symbol denotes *definitional* equality in Haskell (i.e. = is neither a destructive assignment, nor an equality test). Local definitions within other definitions are also possible, as in

```
f x y z = sq x * sq y * sq z where
   sq v = v * v
```

Anonymous functions can furthermore be introduced, with the so-called *lambda*-expression

```
\x y z -> x*y*z
```

being identical to

```
let h x y z = x*y*z in h
```

Type inference

In general, the programmer does not have to supply a type when introducing a new variable. Instead the Hindley-Milner-style type inference algorithm employed in Haskell is able to find the most general type for each expression, which often results in *polymorphic types* being

inferred (i.e. type expressions that include variables standing for arbitrary types). The obvious example of a polymorphic type is given by the identity function,

```
id x = x
```

which has the most general type `a -> a` in Haskell. However, should the programmer so decide, an explicit type annotation can also be used to indicate a more specific type, as in

```
iid :: Int -> Int
iid x = x
```

Partial application

A function like `f` above that takes three integer arguments and delivers an integer result has the type

```
Int -> Int -> Int -> Int
```

However, this does not mean that such a function must always appear before exactly three arguments in an expression. Instead, a function applied to fewer arguments is treated as an expression denoting an anonymous function, which in turn is applicable to the missing arguments. This means that `(f 7)` is a valid expression of type `Int -> Int -> Int`, and that `(f 7 13)` denotes a function of type `Int -> Int`. Note that this treatment is consistent with parsing an expression like `f 7 13 0` as `(((f 7) 13) 0)`.

Pattern-matching

Haskell functions are often defined by a sequence of equations that *pattern-match* on their arguments, like in the following example:

```
fac 0 = 1
fac n = n * fac (n-1)
```

An equivalent, but arguably less elegant definition of the same function would be

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Notice also the use of recursion in these two definitions, which is the prescribed (and only!) way of expressing iteration in a purely functional language. Later in this survey we will see examples of more traditional *loop* constructs being defined for monadic commands. It should be noted, though, that the semantics of these constructs are given in terms of recursion (and that an optimizing compiler, in turn, is likely to implement many forms of recursion in terms of loops on the machine language level).

Another form of pattern-matching, using Boolean *guard* expressions, can also be used, although this form might appear a bit contrived in this simple example.

```
fac n | n==0      = 1
      | otherwise = n * fac (n-1)
```

Moreover, explicit **`case`** expressions are also available in Haskell, as in this fourth variant of the factorial function:

```
fac n = case n of
           0 -> 1
           m -> m * fac (m-1)
```

Algebraic datatypes

User-defined types take the form of *algebraic datatypes* in Haskell, which is a kind of labeled union types with name equality and recursive scope. Here is an example of a type for binary trees:

```
data BTree a = Leaf a
             | Node (BTree a) (BTree a)
```

The type argument `a` is used to make the binary tree type polymorphic in the contents of its leaves; thus a binary tree of integers has the type `BTree Int`. The identifiers `Leaf` and `Node` implicitly defined above are called the *constructors* of the datatype. Constructors, which have global scope in Haskell, can be used both as function identifiers and in patterns, as the following example illustrates:

```
swap v@(Leaf _) = v
swap (Node l r) = Node (swap r) (swap l)
```

This function (of type `BTree a -> BTree a`) takes any binary tree and returns a mirror image of the tree obtained by recursively swapping its left and right branches. The first defining equation also illustrates the use of two special Haskell patterns: the *as*-pattern `v@...` which binds an additional variable to a pattern, and the wildcard _ which matches anything without binding any variables.

Predefined types

In addition to the integers, Haskell's primitive types include characters (`Char`) as well as floating-point numbers (`Float` and `Double`). The type of Boolean values (`Bool`) is a predefined, but still ordinary algebraic datatype.

Lists and tuples are also essentially predefined datatypes, but they are supported by some special syntax. The empty list is written `[]`, and a non-empty list with head `x` and tail `xs` is written `x:xs`. A list known in its entirety can be expressed as `[x1,x2,x3]`, or equivalently `x1:x2:x3:[]`. Moreover, a pair of elements `a` and `b` is written `(a,b)`, and a triple also containing `c` is written `(a,b,c)`, etc. As an illustration to these issues, here follows a function which ``zips'' two lists into a list of pairs:

```
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

The type names for lists and tuples are formed in accordance with the term syntax: `[a]` is the type of lists containing elements of type `a`, and `(a,b)` denotes the type of pairs formed by elements of types `a` and `b`. Thus the type of the function `zip` above is `[a] -> [b] -> [(a,b)]`. There is also degenerate tuple type `()`, called *unit*, which only contains the single element `()`.

Strings are just lists of characters in Haskell, although for this specific type ordinary string syntax can also be used, with `"abc"` being equivalent to `['a','b','c']`. The type name `String` is just a *type abbreviation*, defined as:

```
type String = [Char]
```

String concatenation is thus an instance of general list concatenation in Haskell, for which there exists a standard operator `++`, defined as

```
[]     ++ bs = bs
(a:as) ++ bs = a : (as ++ bs)
```

Haskell also provides a primitive type `Array`, with an indexing operator `!` and an ``update'' operator `//`. However, this type suffers from the fact that updates must be implemented in the purely functional way, which often amounts creating fresh copies of an array each time it is modified. We will see later in this survey how monads and stateful objects may enable us to support the `Array` type in a more intuitive, as well as a more efficient manner.

Higher-order functions

Functions are first-class values in Haskell, so it is quite common that functions are defined to take other functions as parameters. A typical example of this is the standard function `map`, which maps a given list into a new list by applying some unspecified function to each element. `map` is defined as follows:

```
map f []     = []
map f (x:xs) = f x : map f x
```

The higher-orderedness of `map` is exposed in its type,

```
map :: (a -> b) -> [a] -> [b]
```

where it should be noted that the parentheses are essential. As an example of how `map` can be used, we construct a capitalizing function for strings by defining

```
cap = map toUpper
```

where `toUpper :: Char -> Char` is a predefined function that capitalizes characters. The type of `cap` must accordingly be

```
cap :: [Char] -> [Char]
```

or, equivalently,

```
cap :: String -> String
```

Overloading

One of the most prominent features of Haskell is a quite advanced type system for systematic overloading of identifiers and operators. The basic idea is that an overloaded symbol is defined by giving it a polymorphic type signature within a *type class* declaration, after which implementations of the symbol for various types can be supplied by means of so called *instance* declarations. The standard type class that declares the overloaded equality operators is shown below. Enclosing an infix operator in parentheses is the Haskell way of syntactically transforming it into an ordinary prefix identifier.

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
```

When an overloaded symbol is used in a definition, a particular implementation for the symbol can usually be determined by the type inference algorithm on basis of the inferred type, as in

```
f x = if x == 'A' then True else False
```

In other cases, the intended implementation must be abstracted out as a *dictionary parameter*, which shows up in the inferred type as a *type class context* to the left of the actual type. For example, the function

```
elem a []      = False
elem a (x:xs)
   | a == x    = True
   | otherwise = elem a xs
```

is polymorphic in the type that is compared, hence its inferred type becomes

```
elem :: Eq a => a -> [a] -> Bool
```

One reasonable reading of such a type is that the function actually takes three arguments, of which the first is a dictionary value of ``type'' `Eq a`, that contains the overloaded operations used in the function body.

Haskell provides a number of predefined type classes, that group together overloaded symbols implementing for example equality (`Eq`), ordered comparison (`Ord`), basic arithmetic (`Num`), text conversion (`Show`), monad operations (`Monad`), etc. However, overloaded symbols will be used very sparingly in this survey. And even when such symbols are used, the use of type classes will mostly be fully transparent, which means that overloaded symbols like == can actually be considered to possess exactly the type that is required in each particular case.

Layout

Haskell makes extensive use of two-dimensional layout in order to convey information that would otherwise have to be supplied using delimiter symbols. The intended meaning of this convention should hopefully be obvious, and we will continue to use it throughout in our examples. It is occasionally convenient to override the layout rules with a more explicit syntax, though, and hence it may be good to keep in mind that the following two-dimensional code fragment

```
let f x y = e1
    g i j = e2
in g
```

is actually a syntactic shorthand for

```
let { f x y = e1 ; g i j = e2 } in g
```

Records

The first O'Haskell contribution we encounter in this survey is a system for programming with first-class records. Although Haskell already provides some support for records, we have chosen to replace this feature with the present system, partly because the Haskell proposal is a somewhat ad-hoc adaption of the datatype syntax, and partly because Haskell records do not fit too well with the subtyping extension we will describe in the next section.

The distinguishing feature of our record proposal is that the treatment of records and datatypes is perfectly *symmetric*; that is, there is a close correspondence between record selectors and datatype constructors, between record construction and datatype selection (i.e.

pattern-matching over constructors), and between the corresponding forms of type extension, which yields subtypes for records and supertypes for datatypes.

Along this line, we treat both record selectors and datatype constructors as *global* constants - an absolutely normal choice for what datatypes are concerned, but not so for records. Still, we think that a symmetric treatment like the present one has some interesting merits in itself, and that the ability to form hierarchies of record types alleviates most of the problems of having a common scope for all selector names. We also note that overloaded names in Haskell are given very much the same treatment, without much hassle in practice.

A record type is defined in O'Haskell by a global declaration on par with the datatype declarations previously encountered. The following example shows a record type describing two-dimensional points, defined by two selector identifiers of type `Float`.

```
struct Point =
   x,y :: Float
```

The **`struct`** keyword is reused in the term syntax for record construction. We will generally rely on Haskell's layout-rule to avoid cluttering up our record expressions, as in the following example:

```
pt = struct
        x = 0.0
        y = 0.0
```

Since the selector identifiers are global and unique, there is no need to indicate which record type a record term belongs to. It is a static error to construct a record term where the set of selector equations is not exhaustive for some record type.

Record selection is performed by means of the standard *dot*-syntax. O'Haskell distinguishes record selection from the predefined Haskell operator `.` by requiring that the latter expression is followed by some amount of white space before any subsequent identifier. Record selection also binds more tightly than function and operator application, as the following example indicates.

```
dist p = sqrt (sq p.x + sq p.y) where
  sq i = i * i
```

A selector can moreover be turned into an ordinary prefix function if needed, by enclosing it in parentheses, as in

```
xs = map (.x) some_list_of_points
```

Just as algebraic datatypes may take type arguments, so may record types. The following example shows a record type that captures the signatures of the standard equality operators. (Strictly speaking, this record type is not legal since its name coincides with that of a predefined Haskell *type class*. The name `Eq` has a deliberate purpose, though - it makes the example connect on to a known Haskell concept, and in addition it indicates the potential possibility of reducing the number of constructs in O'Haskell by eliminating type class declarations in favour of record types.)

```
struct Eq a =
   eq, ne :: a -> a -> Bool
```

A record term of type `Eq Point` is defined below.

```
pdict = struct
           eq     = eq
           ne a b = not (eq a b)
        where
           eq a b = a.x==b.x && a.y==b.y
```
This example also illustrates three minor points about records: (1) record expressions are not recursive, (2) record selectors possess their own namespace (the equation `eq = eq` above is *not* recursive), and (3) selectors may be implemented using the familiar function definition syntax if so desired.

Subtyping

The subtyping system of O'Haskell is based on *name inequality*. This means that a possible subtype relationship between (say) two record types is determined solely by the names of the involved types, and not by consideration to whether the record types in question might have matching definitions in some sense. Thus, name inequality is just a logical generalization of the name *equality* principle used in Haskell for determining whether two types are equal.

The subtype relation between user-supplied types is defined by explicit declaration. This makes record subtyping in O'Haskell look quite similar to interface extension in Java, as the following type declaration exemplifies:

```
struct CPoint < Point =
   color :: Color
```
The record type `CPoint` is here both introduced, and declared to be a subtype of `Point` by means of the *subtype axiom* `CPoint < Point`. A consequence of this axiom is that the type `CPoint` is considered to possess the selectors `x` and `y` as well, in addition to its own contributed selector `color`. This must be observed when constructing `CPoint` terms, as is done in the following function:

```
addColor p = struct x = p.x; y = p.y; color = Black

cpt = addColor pt
```
Notice here that leaving out the equation `color = Black` would not make the definition invalid, since the function result would then be a value of type `Point` instead of `CPoint`. On the other hand, the selectors `x` and `y` are vital, because without them the record term would not exhaustively implement any record type.

Subtyping can also be defined for algebraic datatypes. Consider the following type modeling the black and white colors:

```
data BW = Black
        | White
```
This type can now be used as the basis for an extended color type:

```
data Color > BW =
   Red | Orange | Yellow | Green | Blue | Violet
```
Since its set of possible values is larger, the new type `Color` defined here must necessarily be a *supertype* of `BW` (hence we use the symbol > instead of < when extending a datatype). The

subtype axiom introduced by the previous type declaration is accordingly `BW < Color`. And analogously to the case for record types formed by extension, the extended type `Color` is considered to possess all the constructors of its base type `BW`, in addition to those explicitly mentioned for `Color`.

Haskell allows pattern-matching to be incomplete, so there is no datatype counterpart to the static exhaustiveness requirement that exists for record types. However, the set of constructors associated with each datatype still influences the static analysis of O'Haskell programs, in the sense that the type inference algorithm approximates the domain of a pattern-matching construct to the smallest such set that contains all enumerated constructors. The two following functions, whose domains become `BW` and `Color`, respectively, illustrate this point.

```
f Black = 0
f _     = 1

g Black = 0
g Red   = 1
g _     = 2
```

Polymorphic subtype axioms

Subtype axioms may be polymorphic, as in the following example where a record type capturing the standard set of comparison operators is formed by extending the type `Eq` defined in a previous section.

```
struct Ord a < Eq a =
   lt, le, ge, gt :: a -> a -> Bool
```

The subtype axiom declared here states that for all types `a`, a value of type `Ord a` also supports the operations of `Eq a`.

Polymorphic subtyping works just as well for datatypes. Consider the following example, which provides an alternative definition of the standard Haskell type `Either`.

```
data Left a =
   Left a

data Right a =
   Right a

data Either a b > Left a, Right b
```

Apart from showing that a datatype declaration need not necessarily declare any new value constructors, the last type declaration above is also an example of type extension with multiple basetypes. It effectively introduces *two* polymorphic subtype axioms; one which says that for all `a` and `b`, a value in `Left a` also belongs to `Either a b`, and one which reads: for all `a` and `b`, the values of type `Right b` form a subset of the values of type `Either a b`.

So, for some fixed `a`, a value of type `Left a` can also be considered to be of type `Either a b`, *for all* `b`. This typing flexibility is actually a form of rank-2 polymorphism, which is put to good use in the following example.

```
f v@(Left _) = v
f (Right 0)  = Right False
f (Right _)  = Right True
```

The interesting part here is the first defining equation. Thanks to subtyping, `v` becomes bound to a value of type `Left a` instead of `Either a Int`. Hence `v` can be reused on the right-hand side, in a context where a value of some supertype to `Right Bool` is expected. The O'Haskell type of `f` thus becomes

```
f :: Either a Int -> Either a Bool
```

Notice also that although both syntactically and operationally a valid Haskell function, `f` is not typeable under Haskell's type regime.

Depth subtyping

Subtyping is a reflexive and transitive relation; i.e. we have that any type is a subtype of itself, and that `S < T` and `T < U` implies `S < U` for all types `S`, `T`, and `U`. The fact that type constructors may be parameterized makes these issues a bit more complicated, though. For example, under what circumstances should we be able to conclude that `Eq S` is a subtype of `Eq T`?

O'Haskell incorporates a quite flexible rule that allows *depth subtyping* within a type constructor application, by taking the *variance* of a type constructor's parameters into account. By variance we mean the role a type variable has in the set of type expressions in its scope -- does it occur in a function argument position, in a result position, in both these positions, or perhaps not at all?

In the definition of record type `Eq` above, all occurrences of the parameter `a` are to the left of a function arrow. For these cases O'Haskell prescribes *contravariant* subtyping, which means that `Eq S` is a subtype of `Eq T` only if `T` is a subtype of `S`. Thus we have that `Eq Point` is a subtype of `Eq CPoint`, i.e. an equality test developed for points can also be used for partitioning colored points into equivalence classes.

The parameter of the datatype `Left`, on the other hand, only occurs as a top-level type expression (that is, in a result position). In this case subtyping is *covariant*, which means for example that `Left CPoint` is a subtype of `Left Point`.

As an example of *invariant* subtyping, consider the record type

```
struct Box a =
   in  :: a -> Box a
   out :: a
```

Here the type parameter `a` takes the role of a function argument as well as a result, so both the co- and contravariant rules apply at the same time. The net result is that `Box S` is a subtype of `Box T` only if `S` and `T` are identical types.

There is also the unlikely case where a parameter is not used at all in the definition of a record or datatype:

```
data Contrived a = Unit
```

Clearly a value of type `Contrived S` also has the type `Contrived T` for any choice of `S` and `T`, thus depth subtyping for this *nonvariant* type constructor can be allowed without any further preconditions.

The motivation behind these rules is of course the classical rule for depth subtyping at the function type, which says that `S -> T` is a subtype of `S' -> T'` only if `S'` is a subtype of `S`, and `T` is a subtype of `T'`. O'Haskell naturally supports this rule, as well as covariant subtyping for the built-in container types lists, tuples, and arrays.

Depth subtyping may now be transitively combined with subtype axioms to infer intuitively correct, but perhaps not immediately obvious subtype relationships. Some illustrative examples are:

Relation:	**Interpretation:**
`Left CPoint < Either Point Int`	*If either some kind of point or some integer is expected, a colored point will certainly do.*
`Ord Point < Eq CPoint`	*If an equivalence test for colored points is expected, a complete set of comparison operations for arbitrary points definitely meets the goal.*

Restrictions on subtype axioms

The general rule when defining subtype axioms is that a defined sub-/supertype (a *base* type) may be any kind of type expression that is not a variable. This means for example that arguments to parameterized base types may be constant type expressions if so desired, they are not limited to just variables.

This general rule is not without restrictions, though. The following examples illustrate the kind of subtype axioms that O'Haskell is forced to reject:

`struct S a < Bool`	*Supertype is not a record type*
`struct S a < S Int`	*Axiom interferes with depth subtyping*
`struct S a < Eq Char, Eq Int`	*Axioms are ambiguous*
`struct S a < Eq (S a)`	*Axiom is recursive*

All but the last of these restrictions are motivated by the type soundness criterion - allowing such axioms easily leads to typings that would break type safety, or at least require yet unknown implementation techniques for both records and datatypes. Recursive axioms, on the other hand, are probably operationally sound, but the current inference algorithm is unable to handle them.

The above restrictions also apply to the implicit subtyping relationships that can be derived by transitivity. Thus, assuming that we have

```
struct Num a < Eq a
   ...
```

the declaration

```
struct A < Ord Char, Num Int
```
is illegal, since its axioms also implicitly declare that `A < Eq Char` and `A < Eq Int`. Note, though, that a declaration like

```
struct A < Ord Char, Num Char
```
is OK, since all transitively generated rules that relate `A` to `Eq` now collapse into one: `A < Eq Char`.

A corresponding set of restrictions exists for subtype axioms relating algebraic datatypes.

Automatic type inference

As is well known, polymorphic subtyping systems need types qualified by *subtype constraints* in order to preserve a notion of principal types. This is easily demonstrated by the following archetypical polymorphic function:

```
twice f x = f (f x)
```
In Haskell, `twice` has the principal type

```
(a -> a) -> a -> a
```
from which every other valid type for `twice` (e.g. `(Point -> Point) -> Point -> Point`) can be obtained as a substitution instance. But if we now allow subtyping, and assume `CPoint < Point`, `twice` can also have the type

```
(Point -> CPoint) -> Point -> CPoint
```
which is not an instance of the principal Haskell type. In fact, there can be no simple type for `twice` that has both `(Point -> Point) -> Point -> Point` and `(Point -> CPoint) -> Point -> CPoint` as substitution instances, since the greatest common anti-instance of these types, `(a -> b) -> a -> b`, is not a valid type for `twice`. So to obtain a notion of principality in this case, we must restrict the possible instances of `a` and `b` to those types that allow a subtyping step from `b` to `a`; that is, to associate the subtype constraint `b < a` with the typing of `twice`. In O'Haskell subtype constraints are attached to types by means of the symbol `|` (the syntax is inspired by the way patterns with Boolean guards are expressed in Haskell), so the principal type for `twice` thus becomes

```
(a -> b) -> a -> b | b < a
```
This type has two major drawbacks compared to the principal Haskell type: (1) it is syntactically longer than most of its useful instances because of the subtype constraint, and (2) it is no longer unique modulo renaming, since it can be shown that, for example,

```
(a -> b) -> c -> d | b < a, c < a, b < d
```
is also a principal type for `twice`. In this simple example the added complexity that results from these drawbacks is of course manageable, but even just slightly more involved examples soon get out of hand, since, in effect, *every application node* in the abstract syntax tree can give rise to a new type variable and a new subtype constraint. Known complete inference algorithms tend to illustrate this point very well, and even if simplification algorithms have been proposed that alleviate the problem to some extent, the general simplification problem is at least NP-hard. Apart from that, it is also an inevitable fact that no conservative

simplification strategies in the world can ever give us back the attractive type for `twice` we know from Haskell.

For these reasons, O'Haskell relinquishes the goal of complete type inference, and employs a *partial* type inference algorithm that trades in generality for a consistently readable output. The basic idea of this approach is to let functions like `twice` retain their original Haskell type, and, in the spirit of monomorphic object-oriented languages, infer subtyping steps only when both the inferred and the expected type of an expression are known. This choice can be justified on the grounds that `(a -> a) -> a -> a` is still likely to be a sufficiently general type for `twice` in most situations, and that the benefit of a consistently readable output from the inference algorithm will arguably outweigh the inconvenience of having to supply a type annotation when this is not the case. We certainly do not want to prohibit exploration of the more elaborate areas of polymorphic subtyping that need constraints, but considering the cost involved, we think it is reasonable to expect the programmer to supply the type information in these cases.

As an example of where the lack of inferred subtype constraints might seem more unfortunate than in the typing of `twice`, consider the function

```
min x y = if less x y then x else y
```

which, assuming `less` is a relation on type `Point`, will be assigned the type

```
Point -> Point -> Point
```

by our algorithm. A more useful choice would probably have been

```
a -> a -> a | a < Point
```

here, but as we have indicated, such a constrained type can only be attained by means of an explicit type annotation in O'Haskell. On the other hand, note that the principal type for `min`,

```
a -> b -> c | a < Point, b < Point, a < c, b < c
```

is a yet more complicated type, and presumably an overkill in any realistic context.

An informal characterization of our inference algorithm is that it improves on ordinary polymorphic type inference by allowing subtyping steps at application nodes when the types are known, as in

```
addColor cpt
```

for example. In addition, the algorithm computes least upper bounds for instantiation variables when required, so that e.g. the list

```
[cpt, pt]
```

will receive the type

```
[Point]
```

Greatest lower bounds for function arguments will also be found, resulting in the inferred type

```
CPoint -> (Int,Bool)
```

for the term

```
\p -> (p.x, p.color == Black)
```

Notice, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, hence a compound expression might receive a less general type, even though its principal type has no constraints. One example of this is

```
let twice f x = f (f x) in twice addColor pt
```

which is assigned the type `Point`, not the principal type `CPoint`.

Unfortunately, a declarative specification of the set of programs that are amenable to this kind of partial type inference is still an open problem. Completeness relative to a system that lacks constraints is also not a realistic property to strive for, due to the absence of principal types in such a system. However, experience strongly suggests that the algorithm is able to find solutions to most constraint-free typing problems that occur in practice - in fact, an example of where it mistakenly fails has yet to be found in practical O'Haskell programming. Moreover, the algorithm is provably complete w.r.t. the Haskell type system, hence it possesses another very important property: programs typeable in Haskell retain their inferred types when considered as O'Haskell programs. On top of this, the algorithm can also be shown to accept all programs typeable in the core type system of Java.

Reactive objects

The central dynamic notion in O'Haskell is the state-encapsulating, concurrently executing *reactive object*. In this section we will survey this dynamic part of the language as it is seen by the programmer. However, the amount of new concepts introduced here is quite extensive, and some of these might not seem immediately compatible with the idea of a purely functional language. Thus some readers might find it helpful to know right from the outset that all constructs introduced in this section are indeed syntactically transformable into a language consisting of only the Haskell kernel and a set of primitive monadic constants. This ``naked'' view of O'Haskell will not be further pursued in this survey, though; the reader is instead referred to [1] and [3] for more information on the subject.

Templates and methods

Objects are instantiated from a **`template`** construct, which defines the *initial state* of an object together with a *communication interface*. A communication interface can be a value of any type, but in order to be useful it must contain at least one *method* that will allow the object to react to method invocations (we will also use the metaphor *sending a message* as a synonym for invoking a method).

A method in turn can be of two forms: either an asynchronous **`action`**, that lets an invoking sender continue immediately and thus introduces concurrency, or a synchronous **`request`**, which allows a value to be passed back to the waiting sender. The body of a method, finally, is a sequence of *commands*, which can basically do three things: update the local state, create new objects, and invoke methods of other objects.

The following code fragment defines a template for a simple counter object:

```
counter = template
             val := 0
          in struct
                inc  = action
                          val := val + 1
                read = request
                          return val
```

Instantiating this template creates a new object with its own unique state variable `val`, and returns an interface that is a record containing two methods: one asynchronous action `inc`, and one synchronous request `read`. Invoking `inc` means sending an asynchronous message to the counter object behind the interface, which will respond by updating its state. Invoking `read`, in turn, will perform a rendezvous with the counter, and return its current value.

Procedures and commands

Actions, requests and templates are all examples of expressions that denote commands in the monad `Cmd`. `Cmd` is actually a type constructor, with `Cmd a` denoting the type of reactive commands that may perform side-effects before returning a value of type `a`. The `Cmd` type is visible in the type of `counter`,

```
counter :: Cmd Counter
```

where `Counter` is assumed to be a record type defined as

```
struct Counter =
   inc  :: Cmd ()
   read :: Cmd Int
```

The result returned from the execution of a monadic command may be bound to a variable by means of the *generator* notation. For example

```
do c <- counter
   ...
```

means that the command `counter` is executed (i.e. the counter template is instantiated), and `c` becomes bound to the value returned (the interface of the new counter object).

Executing a command and discarding its result is simply written without the left-arrow. For example, the result of invoking an asynchronous method is always the uninteresting value `()`, so a suitable way of incrementing counter `c` is therefore

```
do c <- counter
   c.inc
```

The **do**-construct above is by itself a full-blown expression, representing a command sequence, or an anonymous *procedure*, that *when executed* first creates an object instance of the template `counter`, and then invokes the method `inc` of this object. The value returned by a procedure is the value returned by its last command, so the type of the above expression thus becomes `Cmd ()`.

Since the `read` method of `c` is a synchronous method that returns an `Int`, we can write

```
do c <- counter
   c.inc
   c.read
```

and obtain a procedure with the type `Cmd Int`.

Just as for other commands, the result of executing the `read` method can be captured by means of the generator notation:

```
do c <- counter
   c.inc
   v <- c.read
   return v
```

This procedure is actually equivalent to the previous one. The identifier `return` is the trivial built-in command which produces a result value (in this case simply `v`) without performing any effects. Unlike most imperative languages, however, `return` is not a branching construct in O'Haskell - a `return` in the middle of a command sequence means just that a command without effects is executed and its result is discarded. For example

```
do ...
   return (v+1)
   return v
```

is simply identical to

```
do ...
   return v
```

Naming a procedure is moreover just like naming any other expression:

```
testCounter = do c <- counter
                 c.inc
                 c.read
```

`testCounter` is thus the name of a simple procedure which *whenever executed* creates a new counter object and returns its value after it has been incremented once. The counter itself is then simply forgotten, which means that its space requirements eventually will be reclaimed by the garbage collector.

A very useful procedure with a predefined name is

```
done = do return ()
```

which takes the role of a *null* command in O'Haskell.

A word about overloading

Sequencing by means of the **do**-construct, and command injection (via `return`), is not just limited to the `Cmd` monad. Indeed, just as in Haskell, these fundamental operations are *overloaded* and available for any type constructor that is an instance of the *type class* `Monad`. We will not stress this dependance on the overloading system in this survey, though, partly because overloading feature constitutes a virtually orthogonal complement to the subtyping system we put forward, and partly because we do not intend to capitalize on overloading in any essential way in our presentation.

Still, one more monad will be introduced, that is related to `Cmd` by means of subtyping. We will therefore take the liberty of reusing `return` and the **do**-syntax for this new type constructor, even though strictly speaking this means that the overloading system must come into play behind the scenes. And while we are on the subject, the same trick is actually tacitly employed for the equality operator == at a few places as well. However, the overloadings that occur in this survey are all statically resolvable, so our naive presentation of the matter is

intuitively quite correct. It is our intention that this slight toning down of Haskell's most conspicuous type system feature will avoid far more confusion about overloading in O'Haskell than it gives rise to.

Assignable local state

The method-forming constructs **`action`** and **`request`** are syntactically similar to procedures, but with different operational behaviours. Whereas calling a procedure means that its commands are executed by the calling thread, invoking a method triggers execution of commands within the object to which the method belongs. For this reason, methods have no meaning in isolation, and the method syntax is accordingly not available outside the scope of a template.

In actions and requests, as well as in procedures that occur within the scope of a template, two additional forms of commands are available: commands that refer to local state variables (for example the command `return val` in the counter template), and commands that assign new values to these variables (e.g. `val := val + 1`).

As can be expected, state variables are surrounded by several restrictions in order to preserve purely functional semantics of expression evaluation. Firstly they must only occur within commands. For example

```
template
   x := 1
in x
```

is statically illegal, since the state variable `x` is not visible outside the commands of a method or a procedure. Secondly, there are no aliases in O'Haskell, which means that state variables are not first-class values. Thus the procedure declaration

```
someProc r = do r := 1
```

is illegal even if only applied to integer-typed state variables, because `r` is not syntactically a state variable. Parameterization over some unknown state can instead be achieved in O'Haskell by turning the possible parameter candidates into full-fledged objects.

Thirdly, the visibility of a state variable does not reach beyond the innermost enclosing template. This makes the following example ill-formed:

```
template
   x := 1
in
   template
      y := 2
   in
      do x := 0
```

And fourthly, there is a restriction which prevents other local bindings from shadowing a state variable. An expression like the following one is thus disallowed:

```
template
   x := 1
in \x -> ...
```

While not strictly necessary for preserving the purity of the language, this last restriction has the merit of making the question of assignability a simple syntactical matter (see [1]), at the same time as it puts the focus on the very special status that state variables enjoy in O'Haskell.

The `O` monad

Commands that refer or assign to the local state of an object belong to a richer monad `O s`, where the `s` component captures the type of the local state, and where `O s a` accordingly is the type of state-sensitive commands that return results of type `a`. An assignment command always returns `()`, whereas a state-referencing command can return any type. Due to subtyping (more on that below), procedures that contain at least one such command also become lifted to the monad `O s`.

The local state type of an object with more than one state variable is an anonymous tuple type; i.e. there is no information regarding the name of state variables encoded in the state type. For example, the templates

```
a = template
       x := 1
       f := True
    in ...
```

and

```
b = template
       count := 0
       enable := False
    in ...
```

both generate objects with local states of type `(Int,Bool)`. In fact, the anonymity of state variables effectively illustrates that procedures are *parametric* in the actual state they work on - there exists no connection at run-time between a value of some `O` type and the object in which it is declared. Thus, as long as the state types match, a procedure declared within one template may very well work as a local procedure within another template. This particularly means that the possibility of exporting state-dependent procedures does not constitute a loophole in the encapsulation mechanism that O'Haskell provides - still the only way by which an object may affect the state of another object is by invoking an exported *method*.

Self

Closely connected to the concept of a local state is the special variable `self`, which implicitly is in scope inside every template expression, and which may not be shadowed. All occurrences of `self` have type `Ref s`, where `s` is the type of the current local state. The values of type `Ref s` are the actual object references that uniquely identify a particular object at run-time. References are thus very similar to the *PIDs* (process identifiers) that most operating systems generate; they can also be thought of as a form of *pointers* to storage of type `s`. However, since actions and requests implicitly refer to the reference value bound to `self`, most programs do not need to explicitly access this variable. Some further details concerning the `Ref` type are provided below.

It should also be noted that the variable `self` in O'Haskell has nothing to do with the *interface* of an object (in contrast to, for example, `this` in C++ and Java). This is a natural consequence

of the fact that an O'Haskell object may have multiple interfaces - some objects may even generate new interfaces on demand (recall that an interface is simply a value that contains at least one method).

Expressions vs. commands

Although commands are full-fledged values in O'Haskell, there is a sharp distinction between the *execution* of a command, and the *evaluation* of a command considered as a functional value. The following examples illustrate the point.

```
f :: Counter -> (Cmd (), Cmd Int)
f cnt = (cnt.inc, cnt.read)
```

The identifier `f` defined here is a function, not a procedure; it cannot be *executed* in any sense, only applied to arguments of type `Counter`. The fact that the returned pair has command-valued components does not change the status of `f`. In particular, the occurrence of subexpressions `cnt.inc` and `cnt.read` in the right-hand side of `f` does *not* imply that the methods of some counter object get invoked when evaluating applications of `f`.

Extracting the first component of a pair returned by `f` is also a pure evaluation with no side-effects. However, the result in this case is a command value, which has the specific property of being *executable*. By placing a command value in the the command sequence of a procedure, the command becomes subject to execution, *whenever the procedure itself is called*. Such a procedure is shown below.

```
do c <- counter
   fst (f c)
```

The separation between *evaluation* and *execution* of command values can be made more explicit by introducing a name for the evaluated command. This is achieved by the **`let`**-command, which is a purely declarative construct (the equality sign really denotes equality).

```
do c <- counter
   let a = fst (f c)
   a
```

Hence the two preceding examples are actually equivalent, and the counter created will be incremented just once. The following fragment is yet another equivalent example,

```
do c <- counter
   let a = fst (f c)
       b = fst (f c)
   a
```

whereas the next procedure has a different operational behaviour (here the `inc` method of `c` will actually be invoked twice).

```
do c <- counter
   let a = fst (f c)
   a
   a
```

A computation that behaves like `f` above, but which also as a side-effect increments the counter it receives as an argument, must be expressed as a procedure.

```
g :: Counter -> Cmd (Cmd (), Cmd Int)
g cnt = do c.inc
           return (c.inc, c.read)
```

Note that the type system clearly separates the side-effecting computation from the pure one, by applying the `Cmd` type constructor to the range type if the computation happens to be a procedure.

Likewise, the type system demands that computations which depend on the current state of some object be implemented as procedures. For example,

```
h :: Counter -> Int
h cnt = cnt.read * 10
```

is not type correct, since `cnt.read` is not an integer - it is a *command* which *when executed* returns an integer. If we really want to compute the result of multiplying the counter value with 10 we must write

```
h :: Counter -> Cmd Int
h cnt = do v <- cnt.read
           return (v * 10)
```

Subtyping in the `O` monad

We have already indicated that the `Cmd` and `O s` monads are related by subtyping. This is formally expressed as a built-in subtype axiom:

```
Cmd a < O s a
```

This axiom can preferably be read as a higher-order relation: ``all commands in the monad `Cmd` are also commands in the monad `O s`, for any `s`''. Note also that the definition above is another example of the rank-2 polymorphism made possible by a subtype relation based on polymorphic subtype axioms.

One way of characterizing the `Cmd` monad is as a refinement of the `O` monad, that captures the set of all commands that are independent of the current local state. O'Haskell actually takes this idea even further, by providing three more primitive command types, that are related to the `Cmd` monad via the following built-in axioms:

```
Template a < Cmd a
Action     < Cmd ()
Request a  < Cmd a
```

The intention here is of course to provide even more precise typings of the **`template`**, **`action`**, and **`request`** constructs. Thus, the type inferred for the template `counter` defined at the beginning of this section is actually `Template Counter` (instead of `Cmd Counter`), and the types of its two methods are accordingly `Action` and `Request Int`. The record type `Counter` can of course be updated to take advantage of this increased precision:

```
struct Counter =
   inc  :: Action
   read :: Request Int
```

Unlike the refinement step of going from `O s` to the `Cmd` monad (which actually makes more programs typeable because of the rank-2 polymorphism), the distinction between `Cmd` and its subtypes has mostly a documentary value. However, by turning a documentation practice into a type system matter, the type system can additionally be relied on for guaranteeing certain operational properties. For example, a command of type `Template a` can be relied on not to change the state of any existing objects when executed, since a template instantiation only

adds components to the system state. Moreover, commands of type `Action` or `Template a` are guaranteed to be deadlock-free, since a synchronous method can never possess any of these types. Note that none of these properties hold for a general command of type `Cmd a`.

To facilitate straightforward comparison of arbitrary object reference values, O'Haskell provides yet another primitive type with a built-in subtype axiom:

```
Ref a < UniRef
```

By means of this axiom, all object references can be compared for equality (by the overloaded primitive ==) when considered as values of the supertype `UniRef`. O'Haskell moreover provides a predefined record type `Object` for this purpose, which forms a convenient base from which interface types supporting a notion of object identity can be built.

```
struct Object =
   self :: UniRef
```

Of the type constructors mentioned here, `Cmd`, `Template`, and `Request` are all covariant in their single argument. This also holds for the `O` type in case of its second argument, whereas the same constructor, like all types that support both dereferencing and assignment, must be invariant in its state component. Along this line, `Ref` is also forced to be an invariant type constructor.

The `main` procedure

Like in many other languages, the procedure name `main` is special in O'Haskell. This procedure, which must be defined in every program, is implicitly invoked whenever an O'Haskell program is run. The following example shows a program which calls the procedure `testCounter` to obtain an integer value, and then sends this result (converted to a string by `show`) to the standard printing routine of the computing environment.

```
main env = do v <- testCounter
              env.putStr (show v)
```

In contrast to Haskell, `main` takes the computing environment as an argument. This means that the possibility of performing external effects is controllable by the programmer, by appropriate use of parameterization. More details regarding the type of the environment parameter will be further discussed under the heading *Reactivity* below.

Concurrency

An object is an implicit critical section, that is, at most one of its methods can be active at any time. The following example creates a situation with two contending clients sending messages to a counter object. Mutual exclusion between method invocations guarantees that the classical problem of simultaneous updates to a state variable still does not exist.

```
proc cnt = template
              --
           in action
                 cnt.inc

main env = do c <- counter
              p <- proc c
              p
              c.inc
```

```
                v <- c.read
                env.putStr (show v)
```
Methods are furthermore guaranteed to be executed in the order they are invoked (although concurrent execution of multiple objects can make the actual invocation order nondeterministic). This is also illustrated by the previous example. In the `main` procedure it is safe to assume that the value `v` read back from the counter is at least 1, since even though `inc` is an asynchronous method, it must have been processed before the subsequent `read` method call returns. On the other hand, nothing can be said about whether the concurrently executing action of object `p` will be scheduled to make its `inc` call before, in between, or after the two invocations in procedure `main`.

Reactivity

Objects alternate between phases of indefinitely long inactivity and finite method execution. For many applications, the active phases may even be considered momentary, given a sufficiently fast processor. The existence of value-returning synchronous methods does not change that fact, since, assuming that the system is not in deadlock, there are no *other* commands that may block indefinitely, and hence invoking a request cannot do so either. Thus it is important that the computing environment also adheres to this reactive view, by *not* providing any indefinitely blocking operations.

Instead, interactive O'Haskell programs install *callback methods* in the computing environment, with the intention that they will be invoked whenever the event occurs that they are set to handle. As a consequence, O'Haskell programs do not generally terminate when the main routine returns; they are rather considered to be alive as long as there is at least one active object or one installed callback method in the system (or, alternatively, execution may terminate when an object invokes the `quit` method of the environment).

The actual shape of the interface to the computing environment must of course be allowed to vary with the type of application being constructed. The current O'Haskell implementation supports three different environment types, `TkEnv`, `XEnv`, and `StdEnv`, which model the computing environments offered by a Tk server, an X server, and the *stdio* fragment of a Unix operating system, respectively. Several other suggestively named interfaces types are of course also conceivable, e.g. `AppletEnv`, `COMEnv`, `SysEnv`, etc.

A skeleton for a program running under the `StdEnv` environment follows below.

```
prog env = template
              ...
           in struct
                 getChar c = action
                                ...
                 signal n  = action
                                ...

main env = do p <- prog env
              env.interactive p
```
We only show a minimal set of the operations offered by `StdEnv` here, but the main idea behind text-based, interactive programming in O'Haskell should still be evident.

```
struct StdEnv =
   putStr      :: String -> Action
```

```
   interactive :: StdProg -> Action
   quit        :: Action
   ...

struct StdProg =
    getChar :: Char -> Action
    signal  :: Int -> Action
```

The type `StdProg` groups the main event-handlers in a text-oriented program together in a common structure, which is installed in the environment in a single step. Note specifically that `getChar` is a method of the *application*, not the environment, in this reactive formulation. Also note that unless the interactive application performs excessively long computations in response to each input character, signal handling will be almost momentary.

As an example of a more elaborate environment interface, `TkEnv` will be discussed as part of a programming example found here.

Additional extensions

O'Haskell also provides a number of minor, mostly syntactic extensions to the Haskell base, which we will briefly review in this section.

Extended `do`-syntax

The **`do`**-syntax of Haskell already contains an example of an expression construct lifted to a corresponding role as a command: the **`let`**-command, illustrated in section 6. O'Haskell defines commands corresponding to the **`if`**- and **`case`**-expressions as well, using the following quite intuitive syntax:

```
do if e then
      cmds
   else
      cmds
   if e then
      cmds
   case e of
      p1 -> cmds
      p2 -> cmds
```

In addition, O'Haskell provides syntactic support for recursive generator bindings, iteration, and exception handling:

```
do fix x <- cmd y
         y <- cmd x
   forall i <- e do
      cmds
   while e do
      cmds
handle
   exception1 -> cmds
   exception2 -> cmds
```

The **`handle`**-clause is an optional appendix to the **`do`**-construct, with the implied semantics that it handles the listed exceptions within the whole proper command sequence. Like the rest

of the **do**-construct, the translation of these syntactic forms just relies on the existence of some (fairly) standard monadic constants (see [1]). The new command-forms are also overloaded in the full language, although the **while**-command is not of very much use outside the `O` monad, with its support for state variables.

Array updates

To simplify programming with the primitive `Array` type, O'Haskell supports a special array-update syntax for arrays declared as state variables. Assuming `a` is such an array, an update to `a` at index `i` with expression `e` can be done as follows (recall that the array indexing operator in Haskell is `!`):

```
a!i := e
```

Semantically, this form of assignment is equivalent to

```
a := a // [(i,e)]
```

where `//` is Haskell's pure array update operator. But apart from being intuitively simpler, the former syntax has the merit of making it clear that normal use of an encapsulated array is likely to be single-threaded; i.e. the rare cases where `a` is mentioned for another purpose than indexing become easily identifiable. Hence conservative copying of the array can be reserved for these occasions, and ordinary updates to `a` performed in place, which is also exactly what the refined syntax above suggests.

See further the discussion on implementation details in [1].

Record stuffing

Record expressions may optionally be terminated by a type constructor name, as in the following examples:

```
struct ..S

struct a = exp; b = exp; ..S
```

These expressions utilize *record stuffing*, a syntactic device for completing record definitions with equations that just map a selector name to an identical variable already in scope. The missing selectors in such an expression are determined by the appended type constructor (which must stand for a record type), on condition that corresponding variables are defined in the enclosing scope. So if `S` is a (possibly parameterized) record type with selectors `a`, `b`, and `c`, the two record values above are actually

```
struct a = a; b = b; c = c
```

and

```
struct a = exp; b = exp; c = c
```

where `c`, and in the first case even `a` and `b`, must already be bound. Record stuffing is most useful in conjunction with **let**-expressions, as is well illustrated in the programming examples document.

What about inheritance?

Object-oriented modeling is deeply associated with a *classification paradigm*, which attempts to explore behavioral relations between objects according to a biologically inspired *inheritance*-metaphor. One side of such a classification scheme clearly concerns purely syntactical similarities between object interfaces, an aspect of inheritance that is well provided for in O'Haskell in terms of its subtyping system.

Behavioural inheritance in general, though, is a much more complex issue. In most object-oriented languages, inheritance is approximated as reuse of method implementations. This allows a new class of objects (a `template` in O'Haskell terminology) to be incrementally defined on basis of old ones, just as if the methods of the old classes had been syntactically copied directly into the definition of the new class. It also means, though, that the binding of method names to implementations cannot in general be statically known, but must be handled at runtime through a mechanism termed *dynamic binding*.

Dynamic binding can actually be understood as two separate issues:

1. Selection of code for an external method invocation, based not on the static type of the interface, but on the actual class associated with the object behind it (external dispatch).
2. A similar openness to the dynamic class even when the method call is originating from a *sibling method in the receptor object itself* (internal dispatch).

From a higher-order point of view, feature (1) is not very special; it comes for free once methods are elevated to first-class status. For example, an interface to an O'Haskell object of type `Counter` need not necessarily be generated by the previous template `counter`. An alternative origin could just as well be

```
counter2 = template
           in struct
              inc  = action
                        c.inc
                        c.inc
              read = request
                        c.read
```

where `c` might be an interface to a counter of the original kind.

Feature (2) on the other hand, is arguably the source of much of the semantic complexity associated with object-oriented languages. In O'Haskell terms it would mean that the template

```
counter3 = template
              val := 0
           in let
              inc  = action
                        val := val + 1
                        if val == 100 then doit
              read = request return val
              doit = done
           in struct ..ProtectedCounter

struct ProtectedCounter < Counter =
   doit :: Cmd ()
```

cannot be understood independently of its context, since its meaning must be kept sensitive to remote ``redefinitions'' of the exported procedure `doit`.

We have chosen not to support property (2) in O'Haskell, on basis of the following compelling chain of arguments:

> Most uses of dynamic binding in current object-oriented programs are examples of external dispatch.
> Of the remaining cases, all but a few are arguably concerned with overriding of empty (virtual) methods *that were foreseen to be overridden*. Such uses are better expressed by ordinary parameterization in O'Haskell, as in
>
> ```
> counter4 doit = template
>                    val := 0
>                 in struct
>                    inc  = action
>                              val := val + 1
>                              if val == 100 then doit
>                    read = request return val
> ```
>
> Of the remaining cases, all but a few are arguably overridings of default implementations of methods that were still foreseen to be overridden. This can also be captured in O'Haskell:
>
> ```
> counter5 = counter4 done
> ```
>
> Of the remaining cases, many are likely to be quick-and-dirty overridings that result in erroneous code. O'Haskell does not actively support these cases.
> Of the remaining cases, many are likely to depend on access to the full source code of the overridden methods. If source is available, corrections can be made in place. Alternatively, the code can be modified to support explicit parameterization as above. The cases that still remain are arguably too few to justify provision of a radically new semantics of recursive binding just for methods.

Hence it is justifiable to say that O'Haskell enforces a *closed-world* view of objects. This means that inheritance must become identical to simple *delegation*; i.e. the creation of private ``parent'' instances together with the task of plugging the right method values into the right interfaces. Overriding is likewise reduced to a matter of ordinary abstraction and application. We do however foresee that further experimentation with the language will identify the need for some special syntax to make this practice straightforward.

[Back to the O'Haskell homepage]
