Polymorphic Subtyping in O'Haskell

Johan Nordlander

Department of Computing Science
Chalmers University of Technology
S - 412 96 Göteborg, Sweden
Email: nordland@cs.chalmers.se

June 1, 2000

Abstract

O'Haskell is a programming language derived from Haskell by the addition of concurrent reactive objects and subtyping. Since Haskell already is a polymorphic language with overloading, the resulting type system is much richer than what is the norm in almost any widespread object-oriented or functional language. Yet there is strong evidence that O'Haskell is not a complex language to use, and that both Java and Haskell programmers easily can find their way with its polymorphic subtyping system. This paper describes the type system of O'Haskell from a programmer's point of view, with the aid of an illustrative, real world programming example: a strongly typed interface to the graphical toolkit Tk.

1 Introduction

The programming language O'Haskell is the result of a finely tuned combination of ideas and concepts from functional, object-oriented, and concurrent programming, that has its origin in the purely functional language Haskell [Pet97]. The name *O'Haskell* is a tribute to this parentage, where the *O* should be read as an indication of *Objects*, as well as a reactive breach with the tradition that views *Input* as an active operation on par with *Output*. The main design ideas underlying O'Haskell are:

- A minimalistic approach to object-based concurrency, which collapses the notions of an object and a process into a *reactive object*. Reactive objects are asynchronous, state-encapsulating servers whose purpose is to react to input messages; they cannot actively block execution, nor can they selectively filter their sources of input.
- A monadic integration of computational effects, which clearly separates stateful objects from stateless *values*. Apart from higher-order functions and recursive data structures, the latter notion also includes first-class commands, object templates, and methods.
- A safe, polymorphic type system with declared record and variant subtyping, supported by a powerful partial type inference algorithm.

This paper will give an practically oriented overview of this last issue – the O'Haskell type system. The other points above, as well as the full theoretical development of this paper's topic, are covered in the author's thesis [Nor99]. The reader is implicitly referred to that source wherever this paper falls short in its presentation or level of detail.

As an illustration of the merits of the polymorphic subtyping system of O'Haskell, we will use a real, implemented program library as our main example: a statically typed interface to Ousterhout's graphical toolkit *Tk* [Ous94]. The Tk interface is part of the distribution of the O'Haskell interpreter *O'Hugs*, which can be downloaded freely from

```
http:/www.cs.chalmers.se/~nordland/ohugs/
```

It is our intention that this concrete implementation will show that polymorphic subtyping in O'Haskell is not an artificial construct, but a real, practically useful tool that provides valuable expressive power to such complex tasks as the design of fullscale GUIs and APIs.

The rest of the paper is organized as follows. Section 2 gives an informal but relatively thorough overview of the type system of O'Haskell, which is what is at focus in this paper. For the purpose of the subsequent example code, though, section 3 presents the basic dynamic concepts of O'Haskell as well. The actual Tk interface is discussed in section 4, followed by the description and analysis of a typical Tk client in section 5. In section 6, finally, the paper rounds off with a discussion on the experience we have had with the algorithm so far.

2 Type system

2.1 Records and datatypes

In O'Haskell, 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, O'Haskell treats both record selectors and datatype constructors as *global* constants — an absolutely normal choice for what datatypes are concerned, but not so for records (see e.g. [MTH90, Gas97]). 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.

Just as a datatype, a record type is defined in O'Haskell by a global declaration. 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.

Record selection is performed by means of the familiar *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.[1]

```
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.

2.2 Subtyping by declaration

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:

[1] Strictly speaking, this record type is not legal since its name coincides with that of a predefined Haskell *type class*. Type classes form the basis of the *overloading* system of Haskell, whose ins and outs are beyond the scope of this survey. 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. See further the section on future work in [Nor99].

```
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
```

2.3 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 [Lev83], 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.

2.4 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 standard 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'` [CW85]. 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.*

2.5 Partial type inference

As is well known, polymorphic subtyping systems need types qualified by *subtype constraints* in order to preserve a notion of principal types [Mit84, FM90, Hen96]. 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 `|`[2], 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

[2] The syntax is inspired by the way patterns with Boolean guards are expressed in Haskell.

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 inferring subtyping steps at application nodes when the involved types are known, as in

```
addColor cpt
```

for example. However, as soon as an application node involves types that are instantiation variables (as in the body of `twice` above), the algorithm resorts to ordinary unification in order to avoid accumulating constraints on these yet unknown types.

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. See [Nor99] for a formal development of the algorithm as well as these claims.

3 The dynamics of O'Haskell

The central dynamic notion in O'Haskell is the state-encapsulating, concurrently executing reactive object. 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.

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. On the type level, templates, actions, and requests are matched by the predefined constants `Template a`, `Action`, and `Request a`, respectively, which are related to an upper bound `Cmd a` according to the following built-in axioms:

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

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.

As an example of what these steps concretely look like, here follows the main procedure of a program that creates a counter object, increments it, and then prints the resulting counter value on standard output.

```
main env = do cnt <- counter
              cnt.inc
              res <- c.read
              env.putStr ("The value is " ++ show res)
```

Notice that in contrast to Haskell and most other languages, an O'Haskell program takes the interface to its computing environment as an explicit parameter; there are no global procedures or system routines accessible within the language.

The template, action and request constructs are defined by a syntactic translation into a core language that is essentially Haskell augmented with seven mondaic constants, whose semantics, in turn, is given by a non-deterministic transition system. This is all done in such a way that the purely functional semantics of the core is preserved. However, an explanation of the dynamic aspects of O'Haskell is outside the scope of this paper; the reader is referred to [Nor99, NC97] for more information on the issue.

4 The Tk environment

We are now ready to illustrate and concretize the power of polymorphic subtyping in O'Haskell, by showing the central parts of a statically typed interface to the graphical toolkit Tk [Ous94]. Tk is a particularly suitable candidate for having an O'Haskell interface, since it is an example of a tried and tested computing environment that has been consciously designed around an event-driven structure. The fact that Tk is originally untyped makes the task of giving it a flexible interface in O'Haskell even more challenging. The full Tk interface is part of the O'Hugs distribution, which can be obtained from `http://www.cs.chalmers.se/ nordland/ohugs/`.

4.1 Widgets

Every designer of a graphical toolkit is faced with the problem of how to express layout in a convenient manner, and how to make the parameters that control the appearance of the toolkit easily configurable. These problems are not trivial, since the naive solution of having the programmer supply coordinates as well as values for every configurable parameter makes programming with the toolkit an unduly exercise in details.

Tk addresses the first problem with a *packing algorithm* that is able to compute coordinates and sizes for widgets on basis of abstract information like whether stacking direction should be horizontal or vertical, or whether stretchable background "glue" should be inserted, etc. The packing algorithm is however quite complicated, and its interface has an unnecessarily imperative formulation, even for an algorithm written in an imperative language.

The latter problem above is solved in Tk by a calling convention that allows widget creating functions to take a variable number of configuration *options*, just like commands on the standard Unix command line. However, few statically typed languages would allow a direct adaption of this idea, and a translation that attempts to utilize subtyping is furthermore complicated by the fact that options for different widgets often overlap in a way that contradicts the intuitive relationships that one might want to express (e.g. that every button is a widget).

Our O'Haskell interface to Tk makes good use of polymorphism as well as both record and datatype subtyping to circumvent these problems. The basic widget operations are defined in a hierarchy of record types, where each widget type is parameterized w.r.t. the type of the configuration options it takes. We only show the significant parts of the type declarations here, and let incomplete declarations be denoted by trailing dots.

```
struct Widget a =
   focus     :: Action
   bind      :: [Event] -> Action
   set       :: [a] -> Action
   ...

struct WWidget a < Widget a, Packable

struct Button < WWidget ButtonOpt =
   flash     :: Action
```

```
      invoke    :: Action

   struct Canvas < WWidget CanvasOpt =
      line      :: [Pos] -> [LineOpt] -> Request Line
      clear     :: Action
      ...

   struct CWidget a < Widget a =
      getCoords :: Request [Pos]
      setCoords :: [Pos] -> Action
      move      :: Pos -> Action

   struct Line < CWidget LineOpt

   struct Window < Widget WindowOpt =
      button    :: [ButtonOpt] -> Request Button
      canvas    :: [CanvasOpt] -> Request Canvas
      pack      :: Packable -> Action
      ...
```

This hierarchy particularly expresses that every widget subject to automatic layout is a subtype of the type `Packable`. Any object that has this type is a valid target for the packing algorithm of a top-level window. The various layout forms are expressed in terms of layout functions that express vertical and horizontal combination and – for finetuning when necessary – ability or nonability to occupy surplus space in a window.

```
struct Packable =
   packIn :: PathName -> Dir -> PFill -> Expansion -> Cmd ()

row, col  :: [Packable] -> Packable

p1 <<< p2 = row [p1,p2]
p1 ^^^ p2 = col [p1,p2]

data Stretch = None | XFill | YFill | XYFill

expand    :: Packable -> Packable
noExpand  :: Packable -> Packable
stretch   :: Stretch  -> Packable -> Packable

rigid = noExpand . stretch None
```

The listed parameters of the `packIn` method are governed by Tk-specific details that are of little use to the programmer; the packing algorithm is instead initiated by invoking the `pack` method of some top-level window. The convenient use of combinators to express layout is a loan from the Fudget graphics library [CH93].

4.2 Configuration options

The various configuration options are defined as constructors in a hierarchy of datatypes. Since the widget types are parametric in their option type, this option hierarchy can directly mirror any subset relationship Tk defines, without causing any unwanted effects on the widget hierarchy. Moreover, the subtype axiom `WWidget a < Packable` effectively ensures that any pair of packable widgets can be combined by the layout combinators, irrespective of their defined options. A sketch of some of the different widget options available in Tk follows below.

```
data WidthOpt =
      Width Int

data CanvasOpt > WidthOpt =
      Background Color
    | Borderwidth Int
    | Height Int

data WindowOpt > CanvasOpt =
      Title String

data LabelOpt > CanvasOpt =
      Foreground Color
    | Font String
    | Text String
    | Anchor AnchorType

data ButtonOpt > LabelOpt =
      Command Action
    | Enabled Bool

data LineOpt > WidthOpt =
      Fill Color
```

4.3 Modelling events

Datatype subtyping is also put to good use in the definition of mouse events, as shown below.

```
data ButtonPress =
      ButtonPress Int (Pos -> Action)
    | AnyButtonPress (Int -> Pos -> Action)

data MouseEvent > ButtonPress =
      ButtonRelease Int (Pos -> Action)
    | AnyButtonRelease (Int -> Pos -> Action)
    | Motion (Pos -> Action)
    | Double ButtonPress
    | Triple ButtonPress

data Event > MouseEvent, KeyEvent, WindowEvent
```

Notice how the use of two distinct datatypes for mouse-related events makes it possible to model the restriction that double- or triple-clicking are event modifications that are only relevant to button presses; i.e. there are no double or triple button release events.

4.4 The Tk runtime environment

The root type of the Tk interface, finally, is a record type defined as follows:

```
struct TkEnv < StdEnv =
   window    :: [WindowOpt] -> Request Window
   periodic  :: Int -> Cmd () -> Request Runnable
   delay     :: Int -> (String -> Cmd ()) -> Request String
   dialog    :: String -> [(String,Action)] -> Action
   bell      :: Action
```

```
   quit      :: Action
   ...

struct Runnable =
   start :: Action
   stop  :: Action
```

Here `delay`, `periodic`, and `dialog` are true reactive variants of some of the few predefined Tk scripts that bypass the event-driven principle and involve indefinite blocking. Our signatures should particularly be noted for the callbacks they take as arguments (the `String` component in the signature for `delay` is used to distinguish between different timeouts). A nice consequence of our reactive dialog-box implementation is that all objects are alive as usual while the box is open, thus the responsibility of screening out irrelevant mouse events is put where it belongs, i.e. on the window manager.

Since the `main` procedure can be assumed to take a `TkEnv` as an argument, the whole O'Haskell interface to Tk can actually be described in terms of a set of type declarations, plus a set of purely functional layout operators. We believe that this clarity and explicit structure represents an immediate advantage over other strongly typed Tk interfaces, notably the one presented by TkGofer [CMV97].

5 A Tk client example

As an example of how the Tk interface is intended to be used, we give an example of a typical graphics intensive, interactive Tk client in this section. Our client example is a small drawing program, *Scribbler*, that is able to log the movements by the mouse in a canvas window. The code for Scribbler is listed below, as well as in the TkExamples folder in the O'Hugs distribution.

```
struct Scribbler =
   penUp   :: Int -> Pos -> Action
   penDown :: Int -> Pos -> Action
   movePen :: Pos -> Action

scribbler  :: Canvas -> Template Scribbler
scribbler canv =
   template
      pos := Nothing
      col := Red
   in struct
      penUp   _   _ = action pos := Nothing
      penDown but p = action (pos,col) := (Just p,color but)
         where
            color 1 = Red
            color 2 = Green
            color _ = Blue
      movePen     p = action
         case pos of
            Nothing -> done
            Just p' -> canv.line [p',p] [Width 3, Fill col]
                       pos := Just p

main env = do
   win   <- env.window [Title "Scribbler"]
   canv  <- win.canvas [Background White]
   quit  <- win.button [Text "Quit", Command env.quit]
```

```
clear <- win.button [Text "Clear", Command canv.clear]
s     <- scribbler canv
win.pack (canv ^^^ (clear <<< quit))
canv.bind [AnyButtonPress   s.penDown,
           AnyButtonRelease s.penUp,
           Motion           s.movePen]
```

The procedure **main** does a number of things in this example: it creates a window and a number of other graphical *widgets*, it creates the main object of the application, it arranges the widgets according to a specific layout (the **win.pack** method call), and it installs the desired event-handlers (by invoking **canv.bind**). By installing the handlers specifically with the canvas widget, it is understood that only events generated when the mouse is over the canvas will be reported.

Subtyping and subtype inference is utilized at several places in this example:

1. In the argument to **win.button**, the constructors **Text** and **Command** belong to different types, yet the inference algorithm is able to find their least upper bound (LUB): **ButtonOpt**.
2. A similar situation is found in the argument to **canv.bind**, although this time the LUB **MouseEvent** is further promoted to **Event** due to the argument type expected by **canv.bind**.
3. The last example actually makes use of covariant depth subtyping for lists when a subtyping step from **[MouseEvent]** to **[Event]** is inferred on basis of the axiom **MouseEvent < Event**.
4. The widgets **canv**, **clear**, and **quit** are all assigned the type **Packable** when fed to the layout combinators **^^^** and **<<<**. Note the rank-2 polymorphism in use when widgets with different configuration options are given a common LUB.
5. The selection **.window** from **env** leads to an inferred type **TkEnv** for the argument to **main**, which in turn determines that **scribbler** will be linked to the Tk run-time environment, and not just the standard one (**StdEnv**).

Note that all these subtyping steps are automatically inferred by the partial inference algorithm. It can likewise be noted that the type signature for **scribbler** is given here for documentation purposes only; if it is removed the type inference algorithm will recover exactly the same information on basis of the record type declarations that are in scope.

6 Discussion

The positive result regarding partial subtype inference experienced with the Tk example of this paper is not just a lucky coincidence. On the contrary, it reinforces an impression we have from both teaching and doing extended programming with the language: the O'Haskell subtype inference algorithm actually works in practice, and it works remarkably well. In fact, as we have stated previously, an example of where the algorithm mistakenly fails on a type correct program has yet to be found in practical programming.

Considering that this is a result obtained with an algorithm that still lacks a simple declarative specification, we cannot help wondering to what extent such a property is a prerequisite for building up an working understanding of a type inference algorithm. Arguably, most pople feel comfortable with ordinary ML style type inference, not because they always know what to expect in terms of full typing derivations, but because the algorithm is predictable and "makes sense" on a much more intuitive level. We believe that this informal property holds for our algorithm as well, and for that reason we have good hope that there actually exists a simple declarative specification for it. Finding such a specification is an area of ongoing research.

It is also instructive to compare with Pierce & Turner's work on local type inference here [PT98]. They have an algorithm that has a relatively simple declarative specification, although it is not a conservative extension of ML style type inference. Still, it is clear from the examples the authors provide that their algorithm sometimes will exhibit a quite surprising behaviour, simply because it fails to infer typing derivations for many simple programs that are intuitively correct to anyone with prior experience from ML. Indeed, when the limitations of local type inference is discussed, it is noticeable that the provided examples are exclusively chosen among the category of programs that would be typeable in plain ML [HP99]. Having a declarative specification obviously does not help much if the programmer intuitively is using a different kind of system.

We believe our algorithm works because it is complete w.r.t. both the Hindley/Milner system, and core Java; i.e. the purely polymorphic and the purely subtyping systems that people are likely to be experienced with, and which probably define their understanding of the concepts. A different question is of course to what extent programmers will learn to expect polymorphism *and* subtyping to be applied at the same time (like when the body of `twice` is typed with a subtyping step even though the code is polymorphic). It is in scenarios like this that a simple declarative specification of our algorithm would make a most valuable contribution.

References

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets – A Graphical User Interface in a Functional Language. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993.

[CMV97] K. Claessen, E. Meijer, and T. Vullinghs. Implementing Graphical Paradigms in TkGofer. In *Proceedings of the 1997 International Conference on Functional Programming*, Amsterdam, 1997. ACM.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.

[FM90] Y. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73, 1990.

[Gas97] Benedict R. Gaster. Polymorphic Extensible Records for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Hen96] Fritz Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, May 1996.

[HP99] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[Lev83] Daniel Leviant. Polymorphic type inference. In *Conference Record of POPL '83: The 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 88–98, 1983.

[Mit84] J. Mitchell. Coercion and type inference. In *ACM Principles of Programming Languages*, 1984.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NC97] Johan Nordlander and Magnus Carlsson. Reactive Objects in a Functional Language – An escape from the evil "I". In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Nor99] Johan Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Department of Computing Science, Chalmers Technical University, Göteborg, Sweden, May 1999.

[Ous94] J. K. Ousterhout. *Tcl and Tk Toolkit.* Addison-Wesley, 1994.

[Pet97] J. Peterson. The Haskell Home Page. `http://haskell.org`, 1997.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.