Polymorphic subtyping in O'Haskell

Johan Nordlander

Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 20000 NW Walker Road, Beaverton, OR 97006, USA

Abstract

O'Haskell is a programming language derived from Haskell by the addition of concurrent reactive objects and subtyping. Because Haskell already encompasses an advanced type system with polymorphism and overloading, the type system of O'Haskell 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 can easily find their way with its polymorphic subtyping system. This paper describes the type system of O'Haskell both formally and from a programmer's point of view; the latter task is accomplished with the aid of an illustrative, real-world programming example: a strongly typed interface to the graphical toolkit Tk. ⓒ 2002 Elsevier Science B.V. All rights reserved.

Keywords: Type inference; Subtyping; Polymorphism; Haskell; Graphical toolkit

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 [36]. 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 request input by means of blocking commands, nor can they selectively filter their sources of input.

- A monadic integration of computational effects, which clearly separates stateful objects from stateless *values*. In addition to 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 gives a practical as well as formal overview of this last issue—the O'Haskell type system. The other points above, reactivity and the monadic object semantics, are fully covered elsewhere [29,31]. The reader is implicitly referred to these sources wherever this paper falls short in its presentation of O'Haskell in general.

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* [35]. 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. In Section 2 we give an informal but relatively thorough overview of the type system of O'Haskell. Although slightly out of scope in this paper, the dynamic aspects of O'Haskell are then briefly reviewed in Section 3, for the benefit of the subsequent sections. The actual Tk interface is discussed in Section 4, followed by the description and analysis of a typical Tk client in Section 5. Section 6 provides a formal treatment of the inference algorithm and its underlying type system; here we also state our main technical results. Section 7 surveys related work, before we round off in Section 8 with a discussion on the practical 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—a standard choice where datatypes are concerned, but not so for records (see e.g. [19,26]). 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 subject to the same restriction, 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*-notation. 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)
```

---

[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 paper. The name Eq has a deliberate purpose, though—it makes the example connect on to a concept known to most Haskell programmers.

where
```
    eq a b = a.x==b.x && a.y==b.y
```

This example also illustrates two minor points about records: record selectors possess their own namespace (the binding eq = eq above is *not* recursive), and 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 the subtype relationship between (say) two record types is determined solely by the names of the involved types, and not by checking 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 illustrates:

```
  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. The consequence of this axiom is that the type CPoint will possess the selectors x and y as well, besides its own contributed selector color. This must be observed when constructing CPoint terms, as 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, because 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
```

Because 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. The following example 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 that says the same thing but for Right b instead of Left a.

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 [22], 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` as would be the case in Haskell. 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
```

Note 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; that is, 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`; that is, 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' [6]. 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 [13,14,27]. 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 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

---

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

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, for example, 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)
```

Note, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, so 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, finding 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 or unexpectedly fails has yet to be found in practical O'Haskell programming. Moreover, the algorithm is provably complete w.r.t. the Hindley–Milner type system, hence it possesses the very appealing property that 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. In Section 6 we will develop these claims in a more formal manner.

3. The dynamics of O'Haskell

In this section we will briefly summarize the new dynamic concepts that O'Haskell introduces. This is for the purpose of the subsequent presentation only; for a full coverage of the O'Haskell dynamics we refer the reader to more appropriate sources [29,31].

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 defines a way of reacting to external messages.

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 an interface and a template for a simple counter object:

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

counter :: Template Counter
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, the next definition shows 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 stateful 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 monadic 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. A detailed description of how this is achieved is however outside the scope of this paper.

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 [35]. Tk is a particularly suitable candidate for an O'Haskell interface, because 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; the naive solution of having the programmer supply coordinates as well as values for every configurable parameter makes programming with the toolkit a painful 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 fine tuning when necessary—ability or inability 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 an idea borrowed from the Fudget graphics library [7].

4.2. Configuration options

The various configuration options are defined as constructors in a hierarchy of datatypes. Because 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. Modeling 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
```

Note 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; there are no double or triple button release events.

4.4. The Tk runtime environment

The top-level interface to the Tk library 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 pre-defined 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, that is, on the window manager.

   Because 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 [8].

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:

```
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 from the record type declarations that are in scope.

6. Formal development

In this section we will give a formal definition of the O'Haskell type system and its inference algorithm, and state our main technical results. To simplify the presentation we choose to follow established practice and concentrate on a core calculus instead of the full language. Specifically, the core calculus will not mention general pattern matching and the reactive monad syntax; nor will it cover Haskell style overloading; as this extension is virtually orthogonal to the subtyping system we study here. In addition, the presentation also deviates from the implemented algorithm (i.e., the O'Hugs interpreter) in the following ways:

- The implementation fully supports higher-order polymorphism as well as a limited form of higher-order subtyping, whereas the current presentation is first order.
- The implemented algorithm utilizes the folklore trick of top-down propagation of expected types (including type annotations).
- The notion of Γ-constraints introduced below has been slightly generalized in the implementation.

Each of these extensions actually makes the implemented algorithm strictly more powerful than the one presented here; however, they do not contribute more than marginally to the usefulness results claimed elsewhere in this paper. Thus, we believe the current, simplified, formulation of the algorithm adequately captures the central underpinnings of polymorphic subtype inference in O'Haskell.

6.1. Basic definitions

The basis for our formal treatment is the familiar let-polymorphic lambda-calculus:

$$
\begin{array}{llll}
e & ::= & x & \text{variable} \\
  & | & e\, e' & \text{application} \\
  & | & \lambda x \to e & \text{abstraction} \\
  & | & \textbf{let } x = e \textbf{ in } e' & \text{local definition}
\end{array}
$$

Types, type schemes, and subtype constraints are defined as follows:

$$
\begin{array}{llll}
\tau, \rho & ::= & \alpha \mid t\, \tau_1 \ldots \tau_{n_t} & \text{types} \\
\sigma & ::= & \forall \bar{\alpha}.(\tau \mid C) & \text{type schemes} \\
C, D & ::= & \{\tau_i \leqslant \rho_i\}^i & \text{constraint sets}
\end{array}
$$

Type variables are ranged over by α and β, while type constants are ranged over by t and s. The *arity* of a type constant t is denoted n_t. Type constants are considered drawn from some finite set that contains at least the function type constructor ($\to$) with arity 2. Terms and type schemes are considered equivalent up to α-conversion, and we will assume that terms have been renamed so that all bound variables are distinct and not equal to any free variables. For notational convenience, we will also make use of

Typing rules:

$$\frac{}{C,\Gamma\cup\{x:\sigma\}\vdash_P x:\sigma}\ \text{Var}\qquad\frac{C,\Gamma\vdash_P e:\tau'\to\tau\quad C,\Gamma\vdash_P e':\tau'}{C,\Gamma\vdash_P e\,e':\tau}\ \text{App}$$

$$\frac{C,\Gamma\cup\{x:\tau'\}\vdash_P e:\tau}{C,\Gamma\vdash_P \lambda x\to e:\tau'\to\tau}\ \text{Abs}\qquad\frac{C,\Gamma\vdash_P e:\tau\quad C\vdash_P\tau\leqslant\tau'}{C,\Gamma\vdash_P e:\tau'}\ \text{Sub}$$

$$\frac{C,\Gamma\vdash_P e:\sigma\quad C,\Gamma\cup\{x:\sigma\}\vdash_P e':\tau}{C,\Gamma\vdash_P \textbf{let }x=e\textbf{ in }e':\tau}\ \text{Let}$$

$$\frac{C\cup D,\Gamma\vdash_P e:\tau\quad \bar\alpha\notin fv(C,\Gamma)}{C,\Gamma\vdash_P e:\forall\bar\alpha.\tau\,|\,D}\ \text{Gen}$$

$$\frac{C,\Gamma\vdash_P e:\forall\bar\alpha.\tau\,|\,D\quad C\vdash_P[\bar\tau/\bar\alpha]D}{C,\Gamma\vdash_P e:[\bar\tau/\bar\alpha]\tau}\ \text{Inst}$$

Subtype relation:

$$\frac{}{C\cup\{\tau\leqslant\rho\}\vdash_P\tau\leqslant\rho}\ \text{Hyp}\qquad\frac{}{C\vdash_P\tau\leqslant\tau}\ \text{Refl}$$

$$\frac{C\vdash_P\tau\leqslant\rho\quad C\vdash_P\rho\leqslant\tau'}{C\vdash_P\tau\leqslant\tau'}\ \text{Trans}\qquad\frac{\tau<_P\rho}{C\vdash_P\Phi\tau\leqslant\Phi\rho}\ \text{Const}$$

$$\frac{(C\vdash_P\tau_i\leqslant\rho_i)^{i\in t^+}\quad(C\vdash_P\rho_j\leqslant\tau_j)^{j\in t^-}}{C\vdash_P t\,\tau_1\ldots\tau_{n_t}\leqslant t\,\rho_1\ldots\rho_{n_t}}\ \text{Depth}$$

Fig. 1. Basic type system.

the following abbreviations:

$$\begin{aligned}\tau\to\rho&\equiv(\to)\,\tau\,\rho\\\tau\,|\,D&\equiv\forall\emptyset.\tau\,|\,D\\\tau&\equiv\tau\,|\,\emptyset\\C\vdash_P D&\equiv C\vdash_P\tau\leqslant\rho\quad\text{for all }\tau\leqslant\rho\in D\end{aligned}$$

6.2. Typing rules

Fig. 1 shows the inference system for typing judgments in our calculus. With the possible exception of rules Depth and Const, this system is standard in the subtyping literature; in particular, it directly corresponds to a system studied by Henglein [14].

Rule Depth depends on the given *variance* of a type constant, as captured by the following notation: For every type constant t, let the sets $t^+, t^-\subseteq\{1\ldots n_t\}$ represent the argument indices that should be restricted according to the rules of co- and contravariance, respectively (see Section 2.4).

For a built-in type, these choices must, of course, be consistent with the dynamic semantics of terms of that type, which for the function symbol means that $(\to)^+=\{2\}$, and $(\to)^-=\{1\}$ (see for example Cardelli and Wegner [6]). Thus, rule Depth replaces and generalizes the standard rule for functions (called Arrow by Henglein [14]). As

regards types defined by the programmer, Section 6.3 will describe how variance information can be extracted from record and datatype declarations.

Our basic subtyping theory is defined in terms of polymorphic *subtype axioms* that may be instantiated by substitution, as witnessed by rule CONST in Fig. 1.

Definition 6.1 (*Subtype axiom*). If $t \ \bar{\tau}$ and $s \ \bar{\rho}$ are type expressions, $t \neq s$, and $\bar{\tau}$ and $\bar{\rho}$ contain no occurrences of t and s, then $(t \ \bar{\tau} < s \ \bar{\rho})$ is a *subtype axiom* relating t and s.

We consider subtype axioms equivalent up to renaming of variables. To assert the existence of some subtype axiom, we write $(\tau < \rho) \in P$, or more conveniently $\tau <_P \rho$, where P is some given set.

The interaction between multiple subtype axioms is controlled by the following definition:

Definition 6.2 (*Subtyping theory*). A *subtyping theory* P is a finite set of subtype axioms such that

(1) All axioms in P relate distinct pairs of type constants.
(2) If $t \ \bar{\tau} <_P s \ \bar{\rho}$ and $s \ \bar{\rho}' <_P t' \ \bar{\tau}'$, then $\bar{\rho}$ and $\bar{\rho}'$ can be unified and we have $\Phi(t \ \bar{\tau} <_P t' \ \bar{\tau}')$, where Φ is a most general unifier of $\bar{\rho}$ and $\bar{\rho}'$.

Because the set of valid subtyping judgments depends on a particular subtyping theory P, we let the inference system in Fig. 1 be parametric in P. This P has a role analogous to the partial order on types that is assumed in [14]. To see that our definition is an instance of the latter, note that, for each P, the relation

$$\{\Phi(\tau, \rho) \mid \tau <_P \rho \wedge \Phi \text{ is a substitution}\}$$

contains all relationships derivable by our CONST rule, and that its reflexive closure defines a partial order on ground type expressions as required. It is also trivial to verify that our formulation preserves the property that subtyping judgments are closed under substitution.

6.3. Records and datatypes

We now extend our core calculus to include programmer-defined records and datatypes. Because we have chosen to work with name inequivalence, no new subtyping rules need to be introduced here. The focus in this subsection will instead be on the kind of subtyping theory that a particular set of type declarations defines.

We extend our term language with the following constructs:

$$
\begin{array}{llll}
e & ::= & k & \text{datatype constructor} \\
  & \mid & \{k_i \rightarrow e_i\}^i & \text{datatype selection} \\
  & \mid & l & \text{record selector} \\
  & \mid & \{l_i = e_i\}^i & \text{record construction} \\
  & \mid & \ldots
\end{array}
$$

The datatype selection syntax might seem a little unusual, since it does not contain any expression to scrutinize. We have chosen the given formulation to emphasize the symmetry between datatype selection and record construction. The standard syntax for datatype selection, **case** e **of** $\{k_i \rightarrow e_i\}^i$, can be seen as syntactic sugar for the application $\{k_i \rightarrow e_i\}^i \, e$ in this calculus. A similar argument applies to the dot-notation used for record selection, i.e. $e.l \equiv l \, e$.

A *program* is defined as an expression within the scope of a set of top-level record and datatype declarations:

$$
\begin{aligned}
p & ::= \{d_i\}^i \, e \\
d & ::= \textbf{struct } t \, \bar{\alpha} < \{s_i \, \bar{\rho}_i\}^i = \{l_j : \tau_j\}^j \\
  & \quad | \quad \textbf{data } t \, \bar{\alpha} > \{s_i \, \bar{\rho}_i\}^i = \{k_j \, \tau_j\}^j
\end{aligned}
$$

Here the s_i are called *base types*, which should be read as *supertypes* in the case of records, and *subtypes* in datatype declarations. We impose the natural restriction that the former types be only record types, and the latter types include only datatypes. The arity n_t of a declared type constant t is the length of the corresponding argument vector $\bar{\alpha}$. It is tacitly assumed that all type expressions respect arities. We, furthermore, make it an implicit precondition that all declared type constants, as well as all introduced record selectors and datatype constructors, are globally unique.

Now, for every declared record type **struct** $t \, \bar{\alpha} < \{s_i \, \bar{\rho}_i\}^i = \{l_j : \tau_j\}^j$, we define the following:

(1) A type scheme for each selector

$$
\sigma_{l_j} = \forall \bar{\alpha}.t \, \bar{\alpha} \rightarrow \tau_j.
$$

(2) A set of subtype axioms

$$
P_t = \{t \, \bar{\alpha} < s_i \, \bar{\rho}_i\}^i \cup \bigcup_i \{t \, \bar{\alpha} < [\bar{\rho}_i/\bar{\beta}](s' \, \bar{\tau}) \mid (s_i \, \bar{\beta} < s' \, \bar{\tau}) \in P_{s_i}\}.
$$

(3) A selector closure

$$
\hat{t} = \{l_j\}^j \cup \bigcup_i \widehat{s_i}.
$$

(4) A selector environment

$$
\Pi_{\hat{t}} = \{l_j : \sigma_{l_j}\}^j \cup \bigcup_i \{l : \forall \bar{\alpha}.t \, \bar{\alpha} \rightarrow [\bar{\rho}_i/\bar{\beta}]\tau \mid (l : \forall \bar{\beta}.s_i \, \bar{\beta} \rightarrow \tau) \in \Pi_{\widehat{s_i}}\}.
$$

For a datatype declaration **data** $t \, \bar{\alpha} > \{s_i \, \bar{\rho}_i\}^i = \{k_j \, \tau_j\}^j$ the corresponding definitions become

$$
\begin{aligned}
\sigma_{k_j} &= \forall \bar{\alpha}.\tau_j \rightarrow t \, \bar{\alpha}, \\
P_t &= \{s_i \, \bar{\rho}_i < t \, \bar{\alpha}\}^i \cup \bigcup_i \{[\bar{\rho}_i/\bar{\beta}](s' \, \bar{\tau}) < t \, \bar{\alpha} \mid (s' \, \bar{\tau} < s_i \, \bar{\beta}) \in P_{s_i}\},
\end{aligned}
$$

$$\frac{}{C, \Gamma \vdash l : \sigma_l} \;\; \text{S}_{\text{EL}} \qquad \frac{}{C, \Gamma \vdash k : \sigma_k} \;\; \text{C}_{\text{ON}}$$

$$\frac{\Pi_{\{l_i\}^i} = \{l_i : \forall \bar{\alpha}.\tau \to \tau'_i\}^i \quad C, \Gamma \vdash_P e_i : [\bar{\rho}/\bar{\alpha}]\tau'_i}{C, \Gamma \vdash \{l_i = e_i\}^i : [\bar{\rho}/\bar{\alpha}]\tau} \;\; \text{S}_{\text{TRUCT}}$$

$$\frac{\Pi_{\{k_i\}^i} = \{k_i : \forall \bar{\alpha}.\tau'_i \to \tau\}^i \quad C, \Gamma \vdash_P e_i : [\bar{\rho}/\bar{\alpha}]\tau'_i \to \tau'}{C, \Gamma \vdash_P \{k_i \to e_i\}^i : [\bar{\rho}/\bar{\alpha}]\tau \to \tau'} \;\; \text{A}_{\text{LTS}}$$

Fig. 2. Records and datatypes.

$$\hat{t} = \{k_j\}^j \cup \bigcup_i \widehat{s_i},$$

$$\Pi_{\hat{t}} = \{k_j : \sigma_{k_j}\}^j \cup \bigcup_i \{k : \forall \bar{\alpha}.[\bar{\rho}_i/\bar{\beta}]\tau \to t\ \bar{\alpha} \mid (k : \forall \bar{\beta}.\tau \to s_i\ \bar{\beta}) \in \Pi_{\widehat{s_i}}\}.$$

We say that a set of declarations introducing type constants t_i is *well-formed* iff

- $\bigcup_i P_{t_i}$ is a valid subtyping theory.
- All $\hat{t_i}$ are globally unique.

In the sequel, we will tacitly assume that all expressions are typed in the scope of a well-formed set of type declarations, and that $P \supseteq \bigcup_i P_{t_i}$.

For both a record type **struct** $t\ \bar{\alpha} < \{s_i\ \bar{\rho}_i\}^i = \{l_j : \tau_j\}^j$ and a datatype **data** $t\ \bar{\alpha} > \{s_i \bar{\rho}_i\}^i = \{k_j\ \tau_j\}^j$, we let the variance information t^+ and t^- be the smallest sets of type parameter indices such that

$$C \vdash_P \Phi(t\ \bar{\alpha}) \leqslant \Phi'(t\ \bar{\alpha})$$

implies

$$C \vdash_P \Phi(s_i\ \bar{\rho}_i) \leqslant \Phi'(s_i\ \bar{\rho}_i)$$

and

$$C \vdash_P \Phi\tau_j \leqslant \Phi'\tau_j$$

for all i, j, Φ, Φ', and C. Computing these variance sets on basis of variances for the predefined type constants is straightforward; the only complication being that type declarations may be mutually recursive, and hence an iterative procedure will generally be called for. The problem is however easy to formulate in terms of *abstract interpretation* of the given declarations, using an abstract domain that consists of the four subsets of $\{+, -\}$.

The typing rules for records and datatypes are given in Fig. 2. Here (in rules S$_{\text{TRUCT}}$ and A$_{\text{LTS}}$), as well as in the following development, the premises $\Pi_{\{l_i\}^i} = \cdots$ and $\Pi_{\{k_i\}^i} = \cdots$ should be read as an assertion that the corresponding record/selection expression is *well-formed*, that is, that $\{l_i\}^i = \hat{t}$ or $\{k_i\}^i = \hat{t}$ for some t.

We have assumed here that both kinds of type declarations only mention variables that appear in the argument list $\bar{\alpha}$. As has been described by Läufer [21], Jones [17,18],

and others, lifting this restriction for a constructor type naturally leads to a system with support for local existential quantification, while the corresponding generalization for selector types is best interpreted as local universal quantification. The specifics of such a system is beyond the scope of this paper, though.[3]

6.4. Properties of typing judgments

In this section we state the main technical properties of our type system. This exposition will essentially be a recapitulation of results that are proven in general in [14], and which continue to hold for our basic system in particular. Furthermore, our record and datatype extension only trivially affects most of the original proofs. We refer to [31] for all proof details.

Lemma 6.3. *If* $C, \Gamma \vdash_P e : \sigma$ *then* $\Phi C, \Phi \Gamma \vdash_P e : \Phi \sigma$ *for all substitutions* Φ.

Lemma 6.4. *If* $C, \Gamma \vdash_P e : \sigma$ *and* $D \vdash_P C$ *then* $D, \Gamma \vdash_P e : \sigma$.

Proposition 6.5. *Let* $\bar{\alpha} \notin fv(C, \Gamma)$. *Then* $C, \Gamma \vdash_P e : \forall \alpha. \tau \mid D$ *iff* $C \cup D, \Gamma \vdash_P e : \tau$.

For the purpose of the next statement, the subtype relation in Fig. 1 is extended to type schemes as follows:

(1) $C \vdash_P \sigma \leqslant \tau$ if $C, \{x : \sigma\} \vdash_P x : \tau$.
(2) $C \vdash_P \sigma \leqslant \sigma'$ if for all D and τ such that $D \vdash_P C$ and $D \vdash_P \sigma' \leqslant \tau$ we have $D \vdash_P \sigma \leqslant \tau$.

Proposition 6.6 (Invariance under subtyping). *Let* $C \vdash_P \sigma \leqslant \sigma'$. *Then*

(1) *if* $C, \Gamma \vdash_P e : \sigma$ *then* $C, \Gamma \vdash_P e : \sigma'$;
(2) *if* $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma''$ *then* $C, \Gamma \cup \{x : \sigma\} \vdash_P e : \sigma''$.

Theorem 6.7 (Principal types). *Let* $fv(e) \subseteq dom(\Gamma)$, *and let all record and datatype selection expressions in* e *be well-formed. Then there exists a* σ *such that*

(1) $\emptyset, \Gamma \vdash_P e : \sigma;$[4]
(2) *for all* σ', *if* $\emptyset, \Gamma \vdash_P e : \sigma'$ *then* $\emptyset \vdash_P \sigma \leqslant \sigma'$.

Note that this result clearly illustrates that the incompleteness of our preferred inference algorithm for O'Haskell is deliberate, since an algorithm for computing principal types obviously can be defined.

---

[3] Taking this idea further, by also letting the declared sub-/supertypes contain free variables, opens up some interesting possibilities to explore (recall that the generated subtype axioms remain valid in spite of this change). For one thing, a term can now be assigned an existentially quantified type simply by means of a type annotation, or even just by using the term in the right context.

[4] Note that this statement does not say anything about the *satisfiability* of the constraints that are generally contained in σ.

Reduction axioms:

$$
\begin{aligned}
(\lambda x \rightarrow e)\, e' &\;\mapsto\; [e'/x]e & \text{RB\scriptsize ETA} \\
\mathbf{let}\; x = e \;\mathbf{in}\; e' &\;\mapsto\; [e/x]e' & \text{RU\scriptsize NFOLD} \\
l_j\, \{l_i = e_i\}^i &\;\mapsto\; e_j & \text{RS\scriptsize EL} \\
\{k_i \rightarrow e_i\}^i\, (k_j\; e) &\;\mapsto\; e_j\; e & \text{RC\scriptsize ASE} \\
e &\;\mapsto\; e & \text{RR\scriptsize EFL} \\
\lambda x \rightarrow e\; x &\;\mapsto\; e & \text{RE\scriptsize TA}
\end{aligned}
$$

Reduction rules:

$$
\frac{e_1 \mapsto e_1'\quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1'\, e_2'}\;\; \text{RA\scriptsize PP}
\qquad
\frac{e \mapsto e'}{\lambda x \rightarrow e \mapsto \lambda x \rightarrow e'}\;\; \text{RA\scriptsize BS}
$$

$$
\frac{e_1 \mapsto e_1'\quad e_2 \mapsto e_2'}{\mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 \mapsto \mathbf{let}\; x = e_1' \;\mathbf{in}\; e_2'}\;\; \text{RL\scriptsize ET}
$$

$$
\frac{e_i \mapsto e_i'}{\{l_i = e_i\}^i \mapsto \{l_i = e_i'\}^i}\;\; \text{RS\scriptsize TRUCT}
\qquad
\frac{e_i \mapsto e_i'}{\{k_i \rightarrow e_i\}^i \mapsto \{k_i \rightarrow e_i'\}^i}\;\; \text{RA\scriptsize LTS}
$$

Fig. 3. Reduction of terms.

For the purpose of our subject reduction theorem, we define a reduction relation $\mapsto$ for our calculus in Fig. 3. As can be seen, the relation is quite general and encompasses both a call-by-name and a call-by-value semantics. With the exception of the axioms RS\scriptsize EL \normalsize and RC\scriptsize ASE\normalsize, and the rules RS\scriptsize TRUCT \normalsize and RA\scriptsize LTS\normalsize, the same relation also appears in [14].

We also need a notion of *unambiguity*, that allows us to draw certain natural conclusions about the way subtyping judgments are derived.

Definition 6.8 (*Unambiguity*). A constraint-set/subtyping-theory pair (C, P) is *unambiguous* iff the following conditions hold:

(1) If $C \vdash_P t\; \tau_1 \ldots \tau_{n_t} \leqslant t\; \rho_1 \ldots \rho_{n_t}$ then $C \vdash_P \tau_i \leqslant \rho_i$ for all $i \in t^+$ and $C \vdash_P \rho_i \leqslant \tau_i$ for all $i \in t^-$.

(2) For any two distinct record type constructors t and s, if $C \vdash_P t\; \bar{\tau} \leqslant s\; \bar{\rho}$ then there is a subtype axiom $t\; \bar{\alpha} <_P s\; \bar{\rho}'$ such that $C \vdash_P [\bar{\tau}/\bar{\alpha}](s\; \bar{\rho}') \leqslant s\; \bar{\rho}$.

(3) For any two distinct datatype constructors t and s, if $C \vdash_P t\; \bar{\tau} \leqslant s\; \bar{\rho}$ then there is a subtype axiom $t\; \bar{\tau}' <_P s\; \bar{\alpha}$ such that $C \vdash_P t\; \bar{\tau} \leqslant [\bar{\rho}/\bar{\alpha}](t\; \bar{\tau}')$.

The following lemma states a sanity property for our record and datatype extension:

Lemma 6.9. *If $\{d_i\}^i$ is a well-formed set of type declarations introducing t_i, then $(\emptyset, \bigcup_i P_{t_i})$ is unambiguous.*

In other words, an unambiguous pair $(C, \bigcup_i P_{t_i})$ is definitely achievable, and is just a matter of choosing C so that its transitive closure does not interfere with the subtyping rules D\scriptsize EPTH \normalsize and C\scriptsize ONST\normalsize.

Theorem 6.10 (Subject reduction). *Let (C, P) be unambiguous. Then, if $C, \Gamma \vdash_P e : \sigma$ holds and $e \mapsto e'$, $C, \Gamma \vdash_P e' : \sigma$ holds as well.*

We could also go on here and establish a *well-typed-terms-cannot-go-wrong* result about our calculus. Such a result can be obtained by means of the subject reduction theorem, using the standard approach of letting erroneous terms (e.g., terms that attempt to apply a record as a function) reduce to a distinct value *wrong* that has no type. However, since everything not entirely standard about our calculus is captured in Theorem 6.10, these steps are uninteresting an will not be pursued here.

6.5. The inference algorithm

We will now turn to the partial inference algorithm we consider our main result of this chapter. The core of this algorithm is an approximating subtype constraint solver, which has the merit of being simple and efficient—in fact, it is defined as a small extension to the very efficient unification algorithm of Martelli and Montanari [24]. The main characteristic of our solver is that it approximates constraints of the form $\alpha \leqslant \beta$ as equality constraints, and resorts to unification in these cases. When all such constraints are removed from a constraint set, computation of least upper bounds or greatest lower bounds for the remaining variables becomes straightforward, and the algorithm can continue (just like in ordinary unification) by solving any constraints induced by the arguments of the bounding type expressions.

This strategy is not very refined, however, and is bound to fail in certain situations. Consider the constraint set

$$\{\alpha \leqslant CPoint, \alpha \leqslant \beta, Point \leqslant \beta\}$$

With our strategy, α would be unified with β, resulting in the unsolvable constraint set $\{\alpha \leqslant CPoint, Point \leqslant \alpha\}$ (assuming $CPoint <_P Point$). In this case, unifying α with $CPoint$ would have been a better way to proceed.

So, if we want to stick to our simple strategy (which we really do, considering its attractively close relationship to well-known unification techniques), it becomes vital to feed as small constraint sets as possible into the constraint solver, in order to minimize the "damage" that variable/variable constraints can give rise to. For example, if the constraint set above actually was generated as the union of the sets $\{\alpha \leqslant CPoint\}$ and $\{\alpha \leqslant \beta, Point \leqslant \beta\}$, solving the first set separately would lead to success even with our simple strategy.

This requirement actually puts us closer to the standard inference algorithm W [25] than is customary in the literature on subtype inference. As we have indicated, we will have good reasons for solving constraints as soon as they are generated, rather than propagating them upwards in the syntax tree as input to some final simplification/solving pass. The complication we run into is, of course, that constraints involving variables free in the assumption environment cannot be solved immediately if we want the output of the algorithm to be predictable. However, we will postpone the discussion on how we address this problem until we have presented the constraint solver.

6.6. Solving constraints

The definition of the constraint solver is given in Fig. 4. It is presented as an inference system for judgments of the form $\Phi \models_P C$, which should have the operational interpretation "given a constraint set C and subtyping theory P, return substitution Φ". If a constraint set does not match any of the five clauses, the result of the algorithm is considered to be FAILURE. The union symbol $\uplus$ used for pattern-matching against subsets indicates that the subsets must be disjoint.

This algorithm, as well as the subsequent inference algorithm, depends crucially on the ability to generate fresh type variables. Instead of burdening the presentation with unessential details concerning name supplies, we use the following convention: the symbol v always represents a type variable that is distinct from any other variable in the derivation or its context, except for other occurrences of v in the same rule. Likewise, we let $\bar{v}$ represent a vector of zero or more unique variables, equal only to the variables denoted by other occurrences of $\bar{v}$ in the same rule.

Clauses CTRIV to CMERGE in Fig. 4 essentially constitute the unification algorithm of Martelli and Montanari, or more precisely, their nondeterministic *specification* of the algorithm. We prefer to use this abstract formulation here because of its conciseness; the reader is referred to [31] and then further on to [24] for some more concrete information on how to make an efficient implementation, especially on how to avoid the costly occurs-check in clause CMERGE.

The computation of least upper bounds/greatest lower bounds in clause CMERGE is, of course, added by us; the reader should note, though, that our formulation degenerates to the original one in case the trivial subtyping theory $P = \emptyset$ is given. These standard algorithms use a partial order on pure type constants derived from P, which relates t and s if and only if $t = s$ or $t\,\bar{\tau} <_P s\,\bar{\rho}$ for some $\bar{\tau}$ and $\bar{\rho}$. Since neither least upper bounds nor greatest lower bounds are guaranteed to exist (for one thing, $\bar{\tau}$ or $\bar{\rho}$ may be empty), both algorithms might return FAILURE.

$$\frac{}{[\,]\models_P \emptyset}\ \text{CTRIV} \qquad \frac{\Phi \models_P [\beta/\alpha]C}{\Phi \circ [\beta/\alpha] \models_P C \uplus \{\alpha \leqslant \beta\}}\ \text{CVAR}$$

$$\frac{\Phi \models_P C \cup \{\tau_i \leqslant \rho_i\}^{i \in t^+} \cup \{\rho_j \leqslant \tau_j^j\}^{j \in t^-}}{\Phi \models_P C \uplus \{t\,\tau_1 \ldots \tau_{n_t} \leqslant t\,\rho_1 \ldots \rho_{n_t}\}}\ \text{CDEPTH}$$

$$\frac{\alpha \notin fv(C, \bar{\tau}_i, \bar{\rho}_j) \qquad t = (\sqcup_P \overline{t_i})\ [\!]\ (\sqcap_P \overline{s_j})}{\Phi \models_P C \cup \{t_i\,\bar{\tau}_i \leqslant t\,\bar{v}\}^i \cup \{t\,\bar{v} \leqslant s_j\,\bar{\rho}_j\}^j}{\Phi \circ [t\,\bar{v}/\alpha] \models_P C \uplus \{t_i\,\bar{\tau}_i \leqslant \alpha\}^i \cup \{\alpha \leqslant s_j\,\bar{\rho}_j\}^j}\ \text{CMERGE}$$

$$\frac{t\,\bar{\tau}' <_P s\,\bar{\rho}' \qquad \bar{\alpha} = fv(\bar{\tau}', \bar{\rho}')}{\Phi \models_P C \cup \{t\,\bar{\tau} \leqslant [\bar{v}/\bar{\alpha}](t\,\bar{\tau}')\} \cup \{[\bar{v}/\bar{\alpha}](s\,\bar{\rho}') \leqslant s\,\bar{\rho}\}}{\Phi \models_P C \uplus \{t\,\bar{\tau} \leqslant s\,\bar{\rho}\}}\ \text{CSUB}$$

Fig. 4. The constraint solver.

The "fatbar" operator in $(\sqcup_P \overline{t_i}) \, [] \, (\sqcap_P \overline{s_j})$ picks one of its arguments in a failure-avoiding manner. In case both arguments evaluate successfully this choice is assumed to be guided by the inference algorithm, so that the alternative is taken which results in the smallest inferred type. If neither alternative is better than the other we somewhat arbitrarily specify that the left argument is selected. Decorating the constraint solver with an extra parameter representing the type that should be minimized is straightforward, so we leave out the details in the interest of brevity.

Clause CSᴜʙ is our main extension to the unification algorithm; it has no correspondence in the original formulation for the reason that it handles the case where two distinct type constants are compared. Note that this rule is only applicable if there exists an appropriate subtype axiom in P.

The following lemma states a soundness property of the constraint solver.

Lemma 6.11. *If $\Phi \models_P C$ then $\emptyset \vdash_P \Phi C$.*

Since the algorithm deliberately discards certain solutions, completeness cannot hold by definition. However, a distinctive characteristic of our approach is that it is a conservative extension of *unification*. This is made precise below.

Definition 6.12. We say that a substitution Φ *unifies* a constraint set C iff $\Phi\tau = \Phi\rho$ for all $\tau \leqslant \rho \in C$.

Lemma 6.13. *If Φ unifies C, then $\Phi' \models_P C$ and $\Phi = \Phi'' \circ \Phi'$ for some Φ''.*

A vital property of the constraint solver is that it terminates on all input, either with a substitution, or with the implicit result FAILURE if there is no matching clause. Formally,

Proposition 6.14. *The relation $\Phi \models_P C$ is decidable.*

Another important aspect of the solver is that its result is independent of the non-deterministic choices that are allowed by its specification.

Proposition 6.15. *If the constraint solver fails to solve a constraint set C, then it cannot succeed. If it succeeds with a substitution Φ, then Φ is uniquely determined by C up to renaming.*

6.7. Algorithm definition

The actual inference algorithm is shown in Fig. 5. Again we use a formulation in terms of an inference system, whose judgments $C, \Gamma \models_P e : \tau$ should be read "given an assumption environment Γ, a subtyping theory P, and an expression e, return type τ and constraint set C". We will assume here that Γ is of the form $\{x_i : \forall \tilde{\alpha}_i.\tau_i\}$, since handling of constrained type schemes will not be meaningful in our setting until we introduce type annotations (see [30,31] for a formal treatment of this issue).

$$\frac{\overline{\beta_i} = fv(\tau) \quad C = \{v_i \leqslant \beta_i\}^{\beta_i \in (\tau^- \setminus \tilde{x})} \cup \{\beta_i \leqslant v_i\}^{\beta_i \in (\tau^+ \setminus \tilde{x})}}{C, \Gamma \cup \{x : \forall \tilde{\alpha}.\tau\} \models_P x : [\overline{v_i/\beta_i}]\tau} \text{ IV}_{\text{AR}}$$

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leqslant \overline{\tau_i} \to v\}}{\Phi(C \cup \bigcup_i C_i), \Gamma \models_P e \, \overline{e_i} : \Phi v} \text{ IA}_{\text{PP}}$$

$$\frac{C, \Gamma \cup \{x_i : v_i\}^i \models_P e : \tau \quad \Phi \models_P C \setminus C_\Gamma}{\Phi(C_\Gamma), \Gamma \models_P \lambda \overline{x_i} \to e : \Phi(\overline{v_i} \to \tau)} \text{ IA}_{\text{BS}}$$

$$\frac{C, \Gamma \models_P e : \tau \quad C', \Gamma \cup \{x : gen(C, \tau)\} \models_P e' : \tau' \quad \Phi \models_P C' \setminus C'_\Gamma}{\Phi(C \cup C'_\Gamma), \Gamma \models_P \mathbf{let}\ x = e\ \mathbf{in}\ e' : \Phi\tau'} \text{ IL}_{\text{ET}}$$

$$\frac{\forall \tilde{\alpha}.\tau = \sigma_l}{\emptyset, \Gamma \models_P l : [\overline{v}/\tilde{\alpha}]\tau} \text{ IS}_{\text{EL}} \qquad \frac{\forall \tilde{\alpha}.\tau = \sigma_k}{\emptyset, \Gamma \models_P k : [\overline{v}/\tilde{\alpha}]\tau} \text{ IC}_{\text{ON}}$$

$$\frac{\Pi_{\{l_i\}^i} = \{l_i : \forall \tilde{\alpha}.\tau \to \tau'_i\}^i}{\dfrac{C_i, \Gamma \models_P e_i : \rho_i \qquad \Phi \models_P \{\rho_i \leqslant [\overline{v}/\tilde{\alpha}]\tau'_i\}^i}{\Phi\bigcup_i C_i, \Gamma \models_P \{l_i = e_i\}^i : \Phi[\overline{v}/\tilde{\alpha}]\tau}} \text{ IS}_{\text{TRUCT}}$$

$$\frac{\Sigma_{\{k_i\}^i} = \{k_i : \forall \tilde{\alpha}.\tau'_i \to \tau\}^i}{\dfrac{C_i, \Gamma \models_P e_i : \rho_i \qquad \Phi \models_P \{\rho_i \leqslant [\overline{v}/\tilde{\alpha}]\tau'_i \to v\}^i}{\Phi\bigcup_i C_i, \Gamma \models_P \{k_i \to e_i\}^i : \Phi([\overline{v}/\tilde{\alpha}]\tau \to v)}} \text{ IA}_{\text{LTS}}$$

Fig. 5. The inference algorithm.

The existence of a constraint set in the output from our inference algorithm might at first seem contradictory to our whole approach. However, these constraint sets have a very limited form, and are there just for the same purpose as the substitutions returned by Milner's algorithm W: to propagate requirements on the free variables of the assumption environment Γ [25].

But instead of deciding locally on a fixed substitution that makes Γ meet its requirements, our algorithm will work effectively on cloned copies of Γ (rule IV$_{\text{AR}}$), and return a constraint set that relates these clones to the original.[5] It is not until a free variable of Γ exits its scope that its constraints are collected and a satisfying substitution is computed (last premise of rules IA$_{\text{BS}}$ and IL$_{\text{ET}}$). A key element in this step is an operation which takes a constraint set and returns only those constraints which reference variables free in Γ:

$$C_\Gamma = \{\tau \leqslant \rho \mid \tau \leqslant \rho \in C,\ fv(\tau, \rho) \cap fv(\Gamma) \neq \emptyset\}.$$

We formalize the role of these generated constraint sets as follows.

Definition 6.16. We say that a constraint set C is a Γ-*constraint* iff $\tau \leqslant \rho \in C$ implies either $\tau \in fv(\Gamma)$ and $fv(\rho) \cap fv(\Gamma) = \emptyset$ or $\rho \in fv(\Gamma)$ and $fv(\tau) \cap fv(\Gamma) = \emptyset$.

---

[5] Here we let the symbols τ^+ and τ^- stand for the free variables of τ that occur in co- and contravariant positions, respectively.

Lemma 6.17. (1) *If C and C' are Γ-constraints then $C \cup C'$ is a Γ-constraint.*
   (2) *If C is a $(\Gamma \cup \Gamma')$-constraint then C_Γ is a Γ-constraint.*
   (3) *If C is a Γ-constraint, and $vars(\Phi) \cap fv(\Gamma) = \emptyset$, then ΦC is a Γ-constraint.*

Lemma 6.18. *Assume $C, \Gamma \models_P e : \tau$, $\alpha \in fv(\tau)$, and $\beta \in fv(C)$. Then*

(1) *α is a "new" variable, distinct from any variable in the context of the derivation.*
(2) *Either $\beta \in fv(\Gamma)$, or β is a "new" variable, distinct from any variable in the context of the derivation.*
(3) *C is a Γ-constraint.*

Corollary 6.19. *If $C, \Gamma \models_P e : \tau$ and $fv(\Gamma) = \emptyset$ then $C = \emptyset$.*

Thus, for expressions defined on the top-level of a program, the inference algorithm returns just a type if it succeeds.

Generalization plays the same role here as in Milner's W. We define this operation for the full type scheme syntax with constraints, even though we only need this generality in the presence of type annotations (again see [30,31] for details):

$$gen(C, \tau \mid D) = \forall \bar{\alpha}.\tau \mid D \quad \text{where } \bar{\alpha} = fv(\tau, D) \setminus fv(C).$$

Note that *gen* takes the Γ-constraint C as a parameter instead of Γ, since the free variables of $\tau \mid D$ and Γ will always be disjoint.

The vector notations in rule IApp stand for nested application and function type construction, respectively. The possibility of letting more than one argument influence the type of an application expression can have a crucial impact on the result of the algorithm. For example, if $f : \forall \alpha.\alpha \to \alpha \to \alpha$ then $f\ cpt\ pt$ will be assigned the type *Point*, whereas $f\ cpt$ applied to pt will result in a type error (assuming $pt : Point, cpt : CPoint$, and $CPoint <_P Point$). A similar argument, although seemingly less important in practice, applies to the use of nested abstractions in rule IAbs. Thus, it is implicitly understood that rules IApp and IAbs are matched against as large expressions as possible.

It is worth noting that the result of the algorithm is independent of the order in which subexpressions are analyzed (it is only rule IApp that offers any degree of freedom). A related property guarantees that function arguments can be reordered without causing any other effect than a permutation of the corresponding argument types.

Proposition 6.20. *Let $f = \lambda x \to \lambda y \to e$ and $f' = \lambda y \to \lambda x \to e$. Then $C, \Gamma \models_P f : \tau \to \tau' \to \rho$ iff $C, \Gamma \models_P f' : \tau \to \tau' \to \rho$, and $C, \Gamma \models_P f\ e_1\ e_2 : \rho$ iff $C, \Gamma \models_P f'\ e_2\ e_1 : \rho$.*

Detailed examples of how the algorithm works can be found in [31]. We end this section with the main technical results about our inference algorithm.

Theorem 6.21 (Soundness). *If $C, \Gamma \models_P e : \tau$ then $C, \Gamma \vdash_P e : \tau$.*

Definition 6.22. Let $\Gamma \vdash^{HM} e : \forall \bar{\alpha}.\tau$ stand for a typing derivation in which

(1) no use is made of rule Sub,

(2) all subtype constraints are empty,

(3) Γ is of the form $\{x_i : \forall \bar{\alpha}_i.\tau_i\}^i$.

Such a derivation is equivalent to a derivation in the original Hindley–Milner system, or *HM* for short.

Theorem 6.23 (Completeness w.r.t. HM). *Assume $\Phi\Gamma \vdash^{HM} e : \forall\bar{\alpha}.\tau$, where $\bar{\alpha} \notin fv(\Phi\Gamma)$. Then $C, \Gamma \models_P e : \rho$ succeeds, and there is a Φ' such that Φ' unifies C, $\Phi'\Gamma = \Phi\Gamma$, and $\Phi'\rho = \tau$.*

This result gives one lower approximation of the capabilities of our inference algorithm. Another approximative characterization, which is not met by Milner's original inference algorithm, concerns completeness w.r.t. the core type system of Java. For the purpose of this result, we temporarily extend our calculus with type-annotated lambda abstractions, $\lambda x :: \rho \to e$, where $fv(\rho) = \emptyset$. The corresponding typing rule looks as follows:

$$\frac{C, \Gamma \cup \{x : \rho\} \vdash_P e : \tau}{C, \Gamma \vdash_P \lambda x :: \rho \to e : \rho \to \tau} \text{ABS}'$$

The matching clause for the inference algorithm looks almost identical, since no type variables come into play:

$$\frac{C, \Gamma \cup \{x : \rho\} \models_P e : \tau}{C, \Gamma \models_P \lambda x :: \rho \to e : \rho \to \tau} \text{I}_{\text{ABS}}'$$

It should be noted here that annotated lambda bindings are required only because we want to prove completeness, since it is trivial to show that even the identity function lacks a principal type if we abandon polymorphism as in Java. In practical O'Haskell programming, polymorphism and subtyping can be exploited simultaneously, and, as our program examples in the previous sections have hopefully been able to show, explicit type annotations are hardly ever necessary.

Definition 6.24. Let $\Gamma \vdash_P^J e : \tau$ stand for a typing derivation in which

(1) there is no occurrence of any type variable,

(2) all subtype constraints are empty,

(3) all lambda bindings are annotated with their type,

(4) P is a partial order on *nullary* type constants where lubs/glbs are unique, if they exist.

We argue that derivations of this form capture the *essence* of the basic type system of Java. The correspondence is not perfect; for one thing our system lacks Java's static overloading facility. In another aspect the system $\vdash_P^J$ is stronger than Java, since it supports higher-order functions, and does not require the different branches of a datatype selection expression ($\approx$ Java's conditional operator "?") to be trivially subtype-related.

However, what we really have aimed at here is a simple formulation of a language with monomorphic subtyping and records, and Java matches this characterization closely enough to lend it its name.

Now let $C \vdash_P \Gamma \leqslant \Gamma'$ iff $dom(\Gamma) = dom(\Gamma')$ and $C \vdash_P \Gamma(x) \leqslant \Gamma'(x)$ for all $x \in dom(\Gamma)$. Our second partial completeness result is stated as follows:

Theorem 6.25 (Completeness w.r.t. Java). *If* $\Gamma \vdash_P^J e : \tau$ *and* $\emptyset \vdash_P \Gamma' \leqslant \Gamma$, *then* $\emptyset$, $\Gamma' \models_P e : \rho$ *succeeds,* $fv(\rho) = \emptyset$, *and* $\emptyset \vdash_P \rho \leqslant \tau$.

Detailed proofs of this and other results stated in this section can be found in the author's dissertation [31].

7. Related work

Algorithms for polymorphic subtype inference are described in [2,9,10,12–14,20, 23,27,28,38–41,44–47]. Strategies for simplification of the inferred constraint sets are discussed in [2,9,12,23,39,40,41] in particular. The choice to settle for a deliberately incomplete inference algorithm sets our work apart from most of the cited systems, though.

Smith and Trifonov [47], and Pottier [38,39] approximate entailment checking with decidable variants that result in loss of information, but this move is not motivated by any desire to actually eliminate constrained types; the algorithms are rather designed to have a negligible effect in this respect. Aiken and Wimmers' constraint solver also makes some non-conservative, but hardly radical, simplifications in the interest of efficiency [2]. Reppy and Riecke, though, have implemented type inference for their object-oriented ML variant *OML*, in such a way that only constraint-free type schemes are inferred [43]. However, they do not describe their algorithm, nor do they provide any technical results.

Most related to our approach is the Pierce and Turner work on *local type inference* for Pict [37]. They start with a variant of (the explicitly typed) *System* $F_{\leqslant}$, and develop an inference technique that is able to fill in missing type arguments, as well as missing annotations, in many cases. The Pierce and Turner algorithm is not complete w.r.t. the Hindley/Milner system, but they have the advantage of a declarative specification of all programs that the inference algorithm accepts. Their method for inferring types for anonymous abstractions is moreover similar to our implementation of integrated type-checking (as reported in [30]), although their algorithm switches between strict phases of inference and checking, which ours does not. A recent proposal for *colored* local type inference, that allows propagation of partial type information in a top-down manner, is closer to our work in this respect [34].

Cardelli's implementation of $F_{\leqslant}$ [5] contains a partial type inference algorithm that, like ours, uses unification to solve constraints involving unbound type variables. However, this "greedy" algorithm solves *all* constraints, including those on variables in the assumption environment, at the earliest stage possible, thus its behavior can sometimes appear quite unpredictable.

In the specific setting of *usage analysis*, Wansbrough and Peyton Jones describe a constraint approximating algorithm that has similarities to our approach [48]. They share our objective to infer constraint-free types only, they resort to unification for variable-to-variable constraints, and they also notice the problems inherent in finding a precise specification for such an algorithm. Nevertheless, many difficulties are avoided in their system due to their restriction to a trivial two-point subtyping lattice, where a constant bound on a variable uniquely determines a solution, if it is not redundant.

7.1. Name inequivalence

Most of the work on subtype inference cited above take a *structural* view of the subtype relation, that is, types are subtype-related if and only if they have the same structure (a function, a pair, a list, etc.) and their respective constituent parts are subtype-related. This leads to a subtyping theory where the only primitive subtype relationships are between nullary base types. The systems described in [9,44,45] follow [6] and allow two records to be subtype-related if their set of selectors are in a *superset* relationship. Still, it is the structure of a record that defines its position in the subtype hierarchy, not any intentions explicitly declared by the programmer. Henglein's type system [14] is unique in that it is *generic* in the choice of subtyping theory, something which we have exploited in our system based on *name* inequivalence.

Despite the fact that type systems with name-based (in)equivalence dominate among both functional and object-oriented languages currently in widespread use, polymorphic subtype inference for such systems have not gained much attention. The Reppy and Riecke OML employs a scheme where object types are related by declaration [43], while Jategonkar and Mitchell outline a similar (future) system for abstract datatypes in [16]. Both these systems consider only parameterless type constants. Sequeira allows user-defined type constructors with declared nonatomic upper bounds, but the type constructors themselves must always have zero arity [44].

The language closest to O'Haskell in this respect is the Java extension *Pizza* [32], which allows classes to be parameterized on types, and which supports extension of parameterized classes in a way that resembles our polymorphic subtype axioms. There is also a notion of algebraic datatypes in Pizza, but here type extension is not supported. In contrast to O'Haskell, Pizza only supports *invariant* subtyping (i.e. equality) inside a parameterized type constructor, and the *subsumption* principle is not upheld; that is, if an expression of type A is expected and B is a subtype of A, then an expression of type B *cannot* be used in Pizza, except in the special case of method lookup. Pizza is moreover an explicitly typed language on the whole, although polymorphic identifiers are instantiated correctly without manual intervention. The completeness result for our inference algorithm w.r.t. the Java type system could probably be generalized to hold relative to a suitably adapted version of Pizza's type system as well, but proving this remains as future work.

Recent work by Odersky and Zenger introduces the term *nominal* to denote type systems where equality and subtyping is based on declaration rather than on structural properties [33]. The authors argue that such systems are more suitable than the usual theoretical approaches for modeling common object-oriented languages, a position we

have taken in our work as well. The Odersky and Zenger type system has more far-reaching goals than ours, though, in that it seeks to model nested module-like structures of the kind expressible in SML in addition to ordinary object types and classes. In particular, parameterized types are absent in their system, in favor of records that may carry types as well as terms as components.

Subtyping based on variance annotations can be found in the various object calculi by Abadi and Cardelli [1], although here the annotations are attached to method names instead of type constructor arguments. Freeman's work on refinement types involves inferred variances for datatype constructor arguments [11], but his setting is rather different from ours, in the sense that his subtyping system serves to provide additional information on terms that already have an ML type. The system $ML_{\leqslant}$ allows both variances and subtype orderings to be declared, but imposes the restriction that the variance as well as the arity must be uniform for all related type constructors [4]. We are not aware of any system besides ours that provides a combination of subtype declarations as general as our polymorphic subtype axioms, and variance-based depth subtyping.

Objective Caml (Objective ML) conservatively extends ML with subtype inference and a powerful type abbreviation mechanism that achieves much of the succinctness we obtain by using name inequivalence [42]. However, the type system of Objective Caml is based on extensible records (in contrast to extensible record *types*), and does not support subsumption.

Many of the systems mentioned allow subtype constraints to be recursive [2,9,20, 23,32,42], thus facilitating certain object-oriented programming idioms not otherwise expressible [3]. Our system does not currently support recursively constrained types, although a generalization that would allow such constraints to be *checked* (just as ordinary constraints can be checked but not automatically inferred by our system) would be an interesting future extension.

8. 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 a working understanding of a type inference algorithm. Arguably, most people 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.

A comparison with the Pierce and Turner work on local type inference is instructive here as well [37]. Their algorithm 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 behavior, 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 are discussed, it is noticeable that the provided examples are exclusively chosen among the category of programs that would be typeable in plain ML [15]. 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; that is, the purely polymorphic and the purely subtyping systems that programmers 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.

Acknowledgements

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, New York, 1996.

[2] A. Aiken, E. Wimmers, Type inclusion constraints and type inference, ACM Functional Programming and Computer Architecture, Copenhagen, Denmark, June 1993, ACM Press, New York.

[3] K.B. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group (J. Eifrig, S. Smith, V. Trifonov), G.T. Leavens, B. Pierce. On binary methods, Theory Practice Object Systems 1 (3) (1996) 221–242.

[4] F. Bourdoncle, S. Merz, Type-checking higher-order polymorphic multi-methods, Conf. Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Paris, France, 15–17 January 1997, pp. 302–315.

[5] L. Cardelli, An implementation of $F_{\leqslant}$, Technical Report Research Report 97, DEC Systems Research Center, February 1993.

[6] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, Comput. Surveys 17 (4) (1985) 471–522.

[7] M. Carlsson, T. Hallgren. Fudgets—a graphical user interface in a functional language, ACM Functional Programming and Computer Architecture, Copenhagen, Denmark, June 1993, ACM Press, New York.

[8] K. Claessen, E. Meijer, T. Vullinghs, Implementing graphical paradigms in TkGofer, Proc. 1997 Internat. Conf. on Functional Programming, ACM, Amsterdam, 1997.

[9] J. Eifrig, S. Smith, V. Trifonov, Sound polymorphic type inference for objects, OOPSLA '95, ACM, 1995.

[10] M. Fahndrich, A. Aiken, Making set-constraint program analyses scale, Technical Report CSD-96-917, University of California, Berkeley, 1996.

[11] T. Freeman, Refinement types for ML, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, March 1994.

[12] Y. Fuh, P. Mishra, Polymorphic subtype inference: closing the theory-practice gap, Theory and Practice of Software Development, Springer, Barcelona, Spain, 1989.

[13] Y. Fuh, P. Mishra, Type inference with subtypes, Theoret. Comput. Sci. 73 (2) (1990) 155–175.

[14] F. Henglein, Syntactic properties of polymorphic subtyping, TOPPS Technical Report (D-Report Series) D-293, DIKU, University of Copenhagen, May 1996.

[15] H. Hosoya, B.C. Pierce, How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, 1999.

[16] L. Jategaonkar, J.C. Mitchell, Type inference with extended pattern matching and subtypes, Fund. Inform. 19 (1993) 127–166.

[17] M.P. Jones, Using parameterized signatures to express modular structure, ACM Principles of Programming Languages, ACM Press, St. Petersburg, FL, 1996.

[18] M.P. Jones, First-class polymorphism with type inference, in: ACM Principles of Programming Languages, ACM Press, Paris, France, January 1997.

[19] M.P. Jones, S.P. Jones, Extensible records for Haskell, Proc. 1999, Haskell Workshop, Paris, France, October 1999. Available as Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[20] S. Kaes, Type inference in the presence of overloading, subtyping and recursive types, ACM Lisp and Functional Programming, San Francisco, CA, 1992, ACM Press, New York, pp. 193–204.

[21] K. Läufer, Polymorphic type inference and abstract data types, Ph.D. Thesis, New York University, 1992.

[22] D. Leviant, Polymorphic type inference, Conf. Record of POPL'83: The 10th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, 1983, pp. 88–98.

[23] S. Marlow, P. Wadler, A practical subtyping system for Erlang, ICFP, Amsterdam, Holland, June 1997, pp. 136–149.

[24] A. Martelli, U. Montanari, An efficient unification algorithm, ACM Trans. Programming Languages Systems 4 (2) (1982) 258–282.

[25] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (3) (1978) 348–375.

[26] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, Cambridge, MA, 1990.

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

[28] J. Mitchell, Type inference with simple subtypes, J. Funct. Programming 1 (3) (1991) 245–285.

[29] J. Nordlander, M. Carlsson, Reactive objects in a functional language—an escape from the evil "I", Proc. of the Haskell Workshop, Amsterdam, Holland, 1997.

[30] J. Nordlander, Pragmatic subtyping in polymorphic languages, Proc. Third ACM SIGPLAN Internat. Conf. on Functional Programming (ICFP '98), Baltimore, MD, September 1998.

[31] J. Nordlander, Reactive objects and functional programming, Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, May 1999.

[32] M. Odersky, P. Wadler, Pizza into Java: translating theory into practice, ACM Principles of Programming Languages, ACM Press, Paris, France, January 1997.

[33] M. Odersky, C. Zenger, Nested types, 8th Internat. Workshop on Foundations of Object-Oriented Languages (FOOL8), January 2001.

[34] M. Odersky, C. Zenger, M. Zenger, Colored local type inference, ACM Symp. on Principles of Programming Languages, January 2001.

[35] J.K. Ousterhout, Tcl and Tk Toolkit, Addison-Wesley, Reading, MA, 1994.

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

[37] B.C. Pierce, D.N. Turner, local type inference, Conf. Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, 1998.

[38] F. Pottier, Simplifying subtyping constraints, Internat. Conf. on Functional Programming, May 1996, pp. 122–133.

[39] F. Pottier, A framework for type inference with subtyping, Proc. 3rd ACM SIGPLAN Internat. Conf. on Functional Programming (ICFP '98) September 1998, pp. 228–238.

[40] F. Pottier, Simplifying subtyping constraints, Inform. and Comput. 170 (2) (2001) 153–183.

[41] J. Rehof, Minimal typings in atomic subtyping, ACM Principles of Programming Languages, ACM Press, Paris, France, 1997.

[42] D. Rémy, J. Vouillon, Objective ML: a simple object-oriented extension of ML, ACM Principles of Programming Languages, ACM Press, Paris, France, January 1997.

[43] J. Reppy, J. Riecke, Simple objects for standard ML, SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), May 1996.

[44] D. Sequeira, Type inference with bounded quantification, Ph.D. Thesis, University of Edinburgh, 1998.

[45] G. Smith, Principal type schemes for functional programs with overloading and subtyping, Sci. Comput. Programming 23 (1994) 197–226.

[46] M. Sulzmann, M. Odersky, M. Wehr, Type inference with constrained types, TAPOS 5 (1) (1999) 35–55.

[47] V. Trifonov, S. Smith, Subtyping constrained types, Proc. 3rd Interna. Static Analysis Symposium, Lecture Notes in Computer Science, vol. 1145, Springer, Berlin, 1996, pp. 349–365.

[48] K. Wansbrough, S.P. Jones, Simple usage polymorphism, 3rd ACM SIGPLAN Workshop on Types in Compilation, September 2000.