Lexically-scoped type variables

April 1, 2004

Simon Peyton Jones and Mark Shields
Microsoft Research, Cambridge

Abstract

As type inference systems become more sophisticated, it becomes increasingly important to allow the programmer to give type annotations that both document the program and guide type inference. In Haskell 98, it is not possible to write certain type annotations, because they must mention a type that is "in scope" and the language provides no way to name such types.

The obvious solution is to provide language support for *lexically-scoped type variables*, an area whose design space has not been systematically explored. Our contribution is to bring together the relevant folk lore, in coherent form, and make it accessible to a much larger community than hitherto. In particular, we describe and contrast two main alternative designs — the "type-lambda" approach of SML 97, and an alternative "type-sharing" approach which is used by GHC and OCaml — and survey some alternative design choices.

Scoped type variables will play a key role in the type systems of the future; they can no longer be added as an afterthought to language implementations.

1 Motivation

Consider the following Haskell function:

```
prefix :: a -> [[a]] -> [[a]]
prefix x yss = let
                 -- xcons :: [a] -> [a]
                 xcons ys = x : ys
               in
               map xcons yss
```

Haskell allows the programmer to give a type signature for `prefix`, as shown. The type variable "`a`" is implicitly quantified, so the type signature for `prefix` really means:

```
prefix :: forall a. a -> [[a]] -> [[a]]
```

Submitted to ICFP'04

(In fact, Haskell 98 does not permit an explicit `forall` in a type signature, but for the purposes of this paper we will assume that it does – see Section 4.1.5.)

Unfortunately, there is no way to give a type signature for the local function `xcons`, which is why we have put it in comments. If the type signature was un-commented, it would mean

```
xcons :: forall a. [a] -> [a]
```

because of the implicit-quantification rule, and `xcons` simply does not have that type because `x` is free in `xcons`'s right side. Even if Haskell did not have an implicit-quantification rule, and all `forall`'s were explicit, and we wrote

```
xcons :: [a] -> [a]
```

there would remain the question of what the free "`a`" refers to.

The inability to supply a type signature for `xcons` might seem merely inconvenient, but we believe that it is just the tip of an iceberg. Haskell uses type inference to infer types, and that is a wonderful thing. However, insisting on *complete* type inference — that is, the ability to infer types for any typeable program with no help from the programmer — places serious limits on the expressiveness of the type system. Like Pierce and Turner [Pierce and Turner, 1998], we believe that a better design choice is to allow the programmer to guide the type inference process by supplying occasional explicit type annotations, thereby leaving the door open to more sophisticated type systems, as we discuss in Section 2.

So the challenge we address is this: *it should be possible for the programmer to write an explicit type signature for any sub-term of the program*. To do so, some type signatures must refer to a type that is already in the static environment, so we need a way to *name* such types.

The obvious way to address this challenge is by providing language support for *lexically scoped type variables*. Scoped type variables are hardly a new idea but their design space has not been well documented or explored. We offer the following contributions:

- We describe and contrast two rather different approaches to the challenge, the "type-lambda approach" exemplified by SML (Section 3), and the "type-sharing approach" (Section 4) exemplified by the Glasgow Haskell Compiler. The former is specified by the SML Definition, but the presentation here, focused on scoped type variables, is more accessible. The latter has not been formally described before at all.

- Abstracting from these designs, we sketch the salient aspects of the design space, discuss implications and variants, and mention other implementations known to us (Section 5).

Overall, the contribution of the paper is to bring together some folk lore that is well known to a tiny group, to make it accessible to a much larger community. Scoped type variables will play an increasingly important role in the type systems of the future; they can no longer be added as an afterthought to language implementations.

2 Why type annotations are important

Scoped type variables occur in programmer-written type annotations. But are such annotations necessary or desirable in the first place? When SML first appeared on the scene, its type system seemed magical [Milner, 1978]. The compiler can infer polymorphic types for each function, including recursive functions, without any help from the programmer. Indeed, type inference can find the *most general* type for each function: if the program has any valid typing, the inference engine can find it (or a more general one). This approach has proved tremendously attractive and influential, and is adopted by many languages, Haskell among them.

All mainstream languages that take the type-inference approach also allow the programmer to supply *type annotations*. Haskell, for example, permits two forms of type annotation: *declaration type signatures*; and *expression type signatures*. This example shows both:

```
fst :: (Int,Int) -> Int
fst (x,y) = x :: Int
```

In the type inference framework, type annotations written by the programmer are seen as type *restrictions*: if the program type-checks with the type annotations in place, it will also type-check if they are all deleted. The programmer's type annotations are seen merely as machine-checked documentation, which sometimes have the additional effect of restricting the type of the program fragments.

However, in the last ten years or so, there has been an increasing realisation that an insistence on complete type inference — insisting that the type inference engine can type the program when all type annotations are deleted — places inconvenient limitations on the sophistication of the type system.

2.1 Type annotations guide type inference

The most obvious example of this limitation is *polymorphic recursion*. The elegant compromise that makes Hindley-Milner type system work is that a recursive function can only call itself at the same type instance as its current instantiation. Here is an example, taken from [Okasaki, 1998], concerning a data type of sequences:

```
data Seq a = Nil
           | Zero  (Seq (a,a))
           | One a (Seq (a,a))

cons :: a -> Seq a -> Seq a
cons x Nil        = One x Nil
cons x (Zero ps)  = One x ps
cons x (One y ps) = Zero (cons (x,y) ps)
```

The `cons` function is recursive, but its recursive call is at a different type than the current instantiation. It is not typeable using standard Hindley-Milner type inference. Examples like this crop up occasionally in SML, but they are unusual. Haskell made the problem much more pressing because Haskell permits higher-kinded type variables; for example see [Okasaki, 1999].

One possible response to the problem of polymorphic recursion is to make the type inference engine more powerful, so that it can correctly infer types for programs that use polymorphic recursion. Though the problem is in general undecidable, it turns out to be tractable in practice, but only at the cost of fairly heroic changes to the inference algorithm [Mycroft, 1984, Kfoury etal., 1993, Henglein, 1993].

Haskell 98, however, adopts a much simpler approach: polymorphic recursion is permitted if (and only if) the programmer supplies a type signature for the function, as we did for `cons` above. This approach was first used in the language Hope [Burstall etal., 1980], and subsequently in Miranda [Turner, 1985]. It requires virtually no change to the type inference algorithm: when analysing the right-hand side of `cons`, type inference now has available a polymorphic type for `cons`, supplied by the programmer, and it uses that at `cons`'s occurrences. Type inference of the definition of `cons` concludes by checking that the type of the right-hand side matches the supplied signature. It is rather like supplying an induction hypothesis to a theorem prover; doing so reduces the problem from a search of a huge space to a simple matter of checking deductions.

Another carefully-crafted compromise of the Hindley-Milner type system is that the type of a lambda-bound variable must not be a type scheme. For example, this definition of `f` is rejected:

```
f g = (g True, g 'a')
```

Why? Because the lambda-bound variable `g` is applied to two different types, a `Bool` and a `Char`. This is irritating, because if we pass a *polymorphic* function to `f` all would, in fact, be well. For example, no runtime error would result from evaluating `(f (\ x -> x))`.

Again, one could imagine a more cunning type inference algorithm to attack this problem [Kfoury and Tiuryn, 1992], but again there is a much simpler solution: let the programmer supply the hard-to-infer information using a type signature:

```
f :: (forall a. a -> a) -> (Bool, Char)
f g = (g True, g 'a')
```

Now there is no difficulty. When performing type inference on `f`, it is clear that `g` should be given a polymorphic type; and one can also check that applications of `f` apply the function to a suitably polymorphic argument [Odersky and Läufer, 1996, Peyton Jones and Shields, 2004].

2.2 Scoped type variables

The general lesson is this: a little help from the programmer can turn a seriously hard problem into an easy one, thereby allowing the language to support a richer and more expressive type system. If this is the way of the future, and we believe it is, then some type annotations become an essential part of the program text, rather than constituting optional, machine-checked documentation. That in turn makes lexically-scoped type variables an essential component of the language, so that it is possible to write the annotations in the first place.

We begin with Standard ML, which has provided scoped type variables for many years.

3 The type-lambda approach: SML

One plausible approach to adding scoped type variables is to take a hint from System F, the explicitly-typed polymorphic lambda calculus [Girard, 1990]. System F is a well-studied formal calculus that includes scoped type variables. Indeed, GHC, in common with several other typed compilers, uses a variant of System F as its intermediate language, translating Haskell programs into System F after type checking.

In System F, a *type lambda*, written "Λ", binds a type variable, just as a term lambda, written "λ", binds a term variable. For example:

$$\begin{array}{l} id : \forall\alpha.\ \alpha \to \alpha \\ id\ =\ \Lambda\alpha.\ \lambda x : \alpha.\ x \end{array}$$

A term $\Lambda\alpha\,.\,e$ has type $\forall\alpha\,.\,\tau$, for some type τ, just as a term $\lambda x\,.\,e$ has type $\tau_1 \to \tau_2$.

Hence, a very natural idea is to bind a source-language type variable with a source-language type lambda. We call this "the type-lambda approach", and it is the one adopted by SML 97 [Milner etal., 1997]. In SML one can write:

```
fun 'a prefix (x : 'a) yss
  = let
      fun xcons (ys : 'a list) = x :: ys
    in
    map xcons yss
```

Here, "`'a`" following the keyword `fun` is the binding site of an (optional) type parameter of `prefix`; it scopes over the patterns of the definition and its right hand side. We call the type annotations on `x` and `ys` *pattern type signatures*, since they are attached to patterns.

3.1 Implicit scoping

Just as Haskell has implicit universal quantification in type signatures, SML allows the programmer to introduce *implicit* type lambdas. This definition, for example, is elaborated into the previous one:

```
fun prefix (x : 'a) yss
  = let
      fun xcons (ys : 'a list) = x :: ys
    in
    map xcons yss
```

The language definition gives somewhat intricate rules that explain how the implicit type lambdas are placed. For example, consider:

```
fun f x = ....(fun (y:'a) => y)....
```

Where is the type lambda that binds the type variable `'a`? In SML one cannot answer that question without knowing both what the "...." is, and the context for the definition `fun f`. Roughly speaking, the type lambda for an implicitly-scoped type variable `'a` is placed on the innermost function definition that encloses all the free occurrences of `'a`. Here is the exact rule, quoted from [Milner etal., 1997]:

> Every occurrence of a value declaration is said to *scope* a set of explicit type variables, as follows. First, a free occurrence of α in a value declaration `val` *tyvarseq valbind* is said to be *unguarded* if the occurrence is not part of a smaller value declaration within *valbind*. In this case we say that α *occurs unguarded* in the value declaration. Then we say that α is *implicitly scoped* at a particular value declaration `val` *tyvarseq valbind* if (1) α occurs unguarded in *valbind*, and (2) α does not occur unguarded in any larger value declaration containing *valbind*.

This informal, albeit carefully worded, paragraph is the entire specification of the binding of implicitly-scoped type variables; the formal typing rules assume that a pre-processing pass has inserted an explicit binding for every type variable that is implicitly bound by the above rule.

The scoping rules sometimes give slightly odd results. For example:

```
let val (x:'a, y:'b) = <rhs> in <body>
```

Here, `x` and `y` are in scope in `<body>` but not in `<rhs>`, while `'a` and `'b` are in scope in `<rhs>` but not in `<body>` (because they are thought of as type-lambda bindings). In fact, since the definitions would need to be polymorphic in `'a`, such a program would often be rejected anyway. For example:

```
fun swap p = let
               val (x:'a, y:'b) = p
             in (y,x)
```

Here, `p` is free in the environment, so `x` and `y` cannot be generalised and the program would be rejected — and this question of generalisation is what we turn to next.

3.2 Generalisation

Should this function be well typed?

```
fun 'a implies (x:'a) (y:'a) = not x || y
```

Statically, since `(||) :: Bool->Bool->Bool`, it follows that `x` must have type `Bool`. So is it valid to claim it has type "`'a`"? The type-lambda approach would clearly reject this definition: the function `implies` does not have a polymorphic type, and so has no type lambda. Or, to put it another way, the programmer is claiming that `x`'s type is arbitrary, so it should jolly well *be* arbitrary, and not `Bool`.

In this case the type-lambda approach gave a clear answer, but that is not always the case. Consider `prefix` again. Is this version OK?

```
fun prefix x yss
    = let
        fun 'a xcons (ys:'a list) = x :: ys
      in
      map xcons yss
```

In the absence of type annotations, type inference would attribute `ys` with the type `'a list` where `'a` is a type variable, *but* `xcons` *would not be polymorphic in this type*, because of the free occurrence of `x` in its right hand side. So `xcons` does not have a polymorphic type, and has no type lambda in its System F translation. Hence, the type-lambda approach should reject the definition, and indeed SML does so.

A programmer might find that hard to understand. After all, `ys` really does have the type `'a list`; it just happens that `xcons` is not the function that is parametric in `'a`; rather, it is `prefix`. It might also be inconvenient, because it may force the programmer to bind the type variable far away from where it is used. In our example, the definition becomes valid only if we bind `'a` at the definition of `prefix`, as we did in the definitions of Section 3.

Similar subtleties arise when more than one type variable is mentioned:

```
fun f (x:'a) (y:'b) = [x,y]
```

The type-lambda approach rules this out too, because the two supposedly-polymorphic type variables `'a` and `'b` turn out to be the same — `f` has only one type lambda. Indeed, the function is rejected even if the different type variables scope over different equations:

```
fun choose True  (x:'a) (y:'a) = x
  | choose False (x:'b) (y:'b) = y
```

Each equation by itself is OK but, when they are put together, the type variables `'a` and `'b` are unified, and that is rejected (by SML) as being insufficiently polymorphic. This happens even if the two type variables occur in separate, but mutually recursive, functions:

```
rec fun
  ch1 True (x:'a) (y:'a) = x
  ch1 b    x      y      = ch2 b x y
and
  ch2 False (x:'b) (y:'b) = y
  ch2 b     x      y      = ch1 b x y
```

Again, this is rejected under the type-lambda approach, because `'a` and `'b` are unified; there is only one type lambda, not two.

3.3 Type checking

We summarise the effect of scoped type variables on SML's static semantics in Figure 1.

A scoped type variable can only be *bound* at a value binding. To keep things simple, we provide syntax for a simple, non-recursive `let` expression; the $\overline{\alpha}$ are the explicitly-bound type variables. A scoped type variable can *occur* in a type written by the programmer; in our tiny syntax, such a type can annotate the bound variable of a `fun` (lambda), or serve as the type signature of arbitrary term. We let t range over terms, τ and υ over types, and σ over type schemes. We write $\overline{\alpha}$ and $\overline{\tau}$ to denote sequences of type variables or types.

The first two judgements formalise the pre-processor that adds explicit bindings for implicitly-bound scoped type variables. The judgement $\overline{\alpha} \vdash t \hookrightarrow t'$ means that term t is transformed to the explicitly-annotated term t', in a context where the type variables $\overline{\alpha}$ are bound in some outer scope. The auxiliary judgement $\vdash t \nabla \overline{\delta}$ means that the type variables $\overline{\delta}$ occur *unguarded* in t (Section 3.1 gave the SML defintion of unguarded). Notice the way that this judgement ignores the right-hand side of a `let` (rule ULET) because occurrences there are guarded.

Once the term is fully annotated we can describe type checking, which is the third judgement in Figure 1. The environment Γ contains typings for identifiers, $(x:\sigma)$, and type variables, α. The latter are added to Γ when a binding for a lexically-scoped type variable is encountered. The judgement $\Gamma \vdash \tau$ means that τ is well-formed in environment Γ, and is used to check the validity of a type written by the programmer. A type τ is well-formed in Γ if (a) the free type variables of τ are bound in Γ, and (b) the type is well-kinded. So far as well-kinding is concerned, we assume that each type constructor (`int`, `list`, etc) carries its kind with it.

As we have discussed, lexically-scoped type variables are intimately coupled with generalisation, as rule LET shows. The explicitly-scoped type variables are added to the environment before type-checking the right-hand side u, for two reasons: (a) it supports the well-formed-ness check for type annotations in u, and (b) when generalising inner bindings in u we must not generalise over any type variables in $\overline{\alpha}$. That is the reason that $fv(\Gamma)$, defined in Figure 1, includes the type variables *bound* in Γ as well as those *occurring* in types in Γ. Finally, the condition $\overline{\alpha} \notin fv(\sigma)$ checks that we have generalised over all the explicitly-bound type variables.

The rules for type annotations are straightforward (AABS and ANNOT) but notice that SML only permits a term to be annotated with a monotype τ. (In constrast, Haskell allows a quantified type, σ – see Section 4.2.)

Syntax

$$\begin{array}{rcl} t & ::= & \ldots \mid \texttt{let val}\,\overline{\alpha}\,x = u\;\texttt{in}\;t \\ & \mid & \texttt{fun}\,(x{:}\tau)\,.\,t \mid t : \tau \\[1ex] \Gamma & ::= & \varepsilon \mid \Gamma,(x:\sigma) \mid \Gamma,\alpha \\[1ex] fv(\Gamma) & = & \{fv(\sigma) \mid (x:\sigma)\in\Gamma\} \cup \{\alpha \mid \alpha\in\Gamma\} \end{array}$$

Pre-processing $\boxed{\overline{\alpha} \vdash t \hookrightarrow t'}$

$$\frac{\begin{array}{c}\overline{\alpha} \vdash t \hookrightarrow t' \\ \vdash u \nabla \overline{\delta} \\ \overline{\beta'} = \overline{\beta} \cup (\overline{\delta} \setminus \overline{\alpha}) \\ \overline{\alpha} \cup \overline{\beta'} \vdash u \hookrightarrow u'\end{array}}{\overline{\alpha} \vdash \texttt{let val}\,\overline{\beta}\,x = u\;\texttt{in}\;t \hookrightarrow \texttt{let val}\,\overline{\beta'}\,x = u'\;\texttt{in}\;t'}\;\textsc{PLet}$$

$$\frac{\overline{\alpha} \vdash t \hookrightarrow t'}{\overline{\alpha} \vdash \texttt{fun}\,(x{:}\upsilon)\,.\,t \hookrightarrow \texttt{fun}\,(x{:}\upsilon)\,.\,t'}\;\textsc{PAbs}$$

$$\frac{\overline{\alpha} \vdash t \hookrightarrow t'}{\overline{\alpha} \vdash (t : \tau) \hookrightarrow (t' : \tau)}\;\textsc{PAnnot}$$

Unguarded occurrence $\boxed{\vdash t \nabla \overline{\delta}}$

$$\frac{\vdash t \nabla \overline{\delta}}{\vdash \texttt{let val}\,\overline{\alpha}\,x = u\;\texttt{in}\;t \nabla \overline{\delta}}\;\textsc{ULet}$$

$$\frac{\vdash t \nabla \overline{\delta}}{\vdash \texttt{fun}\,(x{:}\upsilon)\,.\,t \nabla \overline{\delta} \cup fv(\upsilon)}\;\textsc{UAbs}$$

$$\frac{\vdash t \nabla \overline{\delta}}{\vdash (t : \tau) \nabla \overline{\delta} \cup fv(\tau)}\;\textsc{UAnnot}$$

Type checking $\boxed{\Gamma \vdash t : \tau}$

$$\frac{\begin{array}{c}\Gamma,\overline{\alpha} \vdash u : \tau_x \\ \overline{\beta} \notin fv(\Gamma) \\ \sigma = \forall\overline{\beta}\,.\,\tau_x \\ \overline{\alpha} \notin fv(\sigma) \\ \Gamma, x:\sigma \vdash t : \tau\end{array}}{\Gamma \vdash \texttt{let val}\,\overline{\alpha}\,x = u\;\texttt{in}\;t : \tau}\;\textsc{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash \upsilon \\ \Gamma,(x:\upsilon) \vdash t : \tau\end{array}}{\Gamma \vdash \texttt{fun}\,(x{:}\upsilon)\,.\,t : \upsilon \to \tau}\;\textsc{AAbs}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \\ \Gamma \vdash t : \tau\end{array}}{\Gamma \vdash (t : \tau) : \tau}\;\textsc{Annot}$$

Figure 1: Additional type-checking rules for ML-style explicitly-scoped type variables

4 The type-sharing approach: GHC

The type-lambda approach is perfectly viable, as SML shows, but as we have seen, its details are somewhat subtle. In this section we describe an alternative approach, which has been implemented in the Glasgow Haskell compiler for several years. (In the rest of the paper we revert to Haskell syntax.)

The central idea is this:

> *A scoped type variable is simply a name for a type.*

So, if I write `(\x::a -> (x,True) :: (a,Bool))`, I mean only that `a` is a name for the type of `x`, no more and no less. This name is mentioned in the expression type signature `(a,Bool)`, which claims that `(x,True)` has a pair type whose first component is `a`, the type of `x`, and whose second component is `Bool`.

In contrast to the type-lambda approach, there is no connection between scoped type variables and generalisation. For example, consider again our `prefix` example:

```
prefix x yss = let
                 xcons (ys::[a]) = x : ys
               in
               map xcons yss
```

This definition is valid in our system; the scoped type variable `a` simply names the type of the elements of `ys`, without any requirement that `xcons` be parametric in that type. Scoped type variables simply identify a sharing point in the type graph, rather than identifying polymorphism. For this reason, we call it the *type-sharing approach*[1].

The type named by a scoped type variable does not even need to be a type variable. For example, our `implies` example is valid in our system:

```
implies (x::a) (y::a) = not x || y
```

Here `a` is simply a name for the type of `x` and `y` (which must therefore be the same). The two pattern type signatures require that `x` and `y` have the same type, but say nothing about what that type is. The fact that `x` is an argument to `not` forces that type to be `Bool`, so in the end `a` names the type `Bool`.

We have found that this view is much easier to explain than the type-lambda view. Having implemented both, we know that it is also easier to implement.

4.1 Scoping in GHC

Syntactically, GHC's scoped type variables look very similar to the implicit-type-lambda approach: a scoped type variable can be introduced by binding it in a *pattern type signature*. For example:

```
prefix :: b -> [[b]] -> [[b]]
prefix (x::a) yss
  = let
      xcons (ys::[a]) = x : ys
    in
    map xcons yss
```

The "`::a`" in the first argument of `prefix` is called a "pattern type signature", because it gives a type annotation for a pattern. It brings into scope the type variable `a`.

[1]Not to be confused with the type sharing constraints of the SML and OCaml module systems.

4.1.1 Pattern type signatures

Pattern type signatures obey the following rules:

- A pattern type signature brings into scope any type variables free in the signature that are not already in scope. For example, the pattern `x::a` brings the type variable `a` into scope (as well as the term variable `x`). On the other hand, the pattern `ys::[a]` *mentions* `a`, rather than shadowing it as a conventional lambda binding would, because `a` is already in scope.

- The scope of a freshly-introduced type variable is lexical, *and is the same as the scope of any term variable bound by the same pattern*. For example, the scope of `a` is the same as the scope of `x`. This simple scoping rule contrasts with the subtleties involved in finding a suitable type-lambda binding site (Section 3.1).

- The scope of a freshly-introduced type variable includes subsequent patterns in the same binding. For example, we could write `prefix` like this:

  ```
  prefix (x::a) (yss::[[a]]) = ...
  ```

  The pattern `x::a` brings into scope the type variable `a`, and *it scopes over subsequent patterns*[2], including `ys::[[a]]`.

- A pattern type signature may be attached to any pattern whatsoever: in function arguments, lambda abstractions, case expressions, and pattern bindings in `let` and `where` blocks. The pattern need not be a simple variable, nor does the type in a pattern type signature need to be a simple type variable. For example, the following are all legal

  ```
  -- Function argument
  f ((x,y)::a) = x && y
        -- Here a = (Bool,Bool)

  -- Lambda abstraction
  f = \ ((x,y)::(a,b)) -> x + y

  -- Case expression
  f p = case p of
          (x::a, y::b) -> x + y

  -- Pattern binding
  f p = let (x,y)::(a,b) = p in x+y
  ```

  In all these cases, the scope of `a` and `b` is the same as the scope of `x` and `y`. For example, in the last example, `a` and `b` scope over the right-hand side of the `let` as well as the `in` part. (In Haskell, `let` is always recursive.)

A scoped type variable is a name for a *type*, and not a *type scheme*. (A type has no embedded `forall`'s inside it.) This restriction is necessary to preserve type inference, and it is not onerous in practice; when writing this paper we were unable to produce any plausible example programs that violate it.

4.1.2 Result signatures

A pattern type signature allows one to bind a scoped type variable to (part of) the type of an *argument*. Sometimes, though, it is helpful to name components of the *result* of a function.

[2]Since patterns never contain *occurrences* of term variables, it does no harm for term-variable bindings to scope over subsequent patterns as well.

```
f n :: ([a] -> [a])
  = let
       g (x::a, y::a) = (y,x)
    in
    \xs -> map g (reverse xs ‘zip‘ xs)
```

The type annotation “`:: ([a] -> [a])`” is a *result signature*. It claims that the right hand side of `f` has type `[t] -> [t]` for some type `t`, and binds the scoped type variable `a` to `t`. The scope of the bound type variable(s) is the right-hand side of the definition.

Result signatures can be used on lambda abstractions and `case` expressions, as well as function definitions.

4.1.3 *Class and instance declarations*

In GHC, the type variables introduced by Haskell’s `class` or `instance` declarations also scope over the body of the declaration. For example:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  (/=) x y = not ( (x::a) == (y::a) )

instance (Eq a, Eq b) => Eq (a,b) where
  (==) (x1,y1) (x2,y2) =  ((x1::a) == x2)
                       && ((y1::b) == y2)
```

In the `class` declaration, the type variable `a` scopes over the entire `where` clause, including the type signatures `(x::a)` and `(y::a)`; and similarly for the type variables `a` and `b` introduced in the instance declaration.

4.1.4 *Existentials*

Several Haskell compilers support *existential data types*. For example:

```
data Ap = forall a. Ap [a] ([a] -> Int)
```

This declaration defines `Ap` as an existential pair: the constructor `Ap` has type

```
Ap :: forall a. [a] -> ([a] -> Int) -> Ap
```

(Hence the “`forall`”.) Notice that the type `Ap` is not parameterised over the type `a`.

Values of type `Ap` can be taken apart using pattern-matching as usual, and pattern type signatures can be particularly useful to name the existential type:

```
revap :: Ap -> Int
revap (Ap (xs::[a]) f) = f ys
                       where
                         ys :: [a]
                         ys = reverse xs
```

Here the pattern type signature for `xs` brings `a` into scope, while the declaration type signature for `ys` makes use of that binding.

4.1.5 *Implicit quantification of type signatures*

Pattern type signatures bring type variables into scope, and these lexically-scoped type variables modify the implicit-quantification rule for Haskell’s existing type signatures, in the following way. In an ordinary Haskell type signature (on declarations or expressions), *the implicit-quantification rule applies only to type variables that are not in scope*. So we could write `prefix` like this:

Syntax

External monotypes	υ	$::=$	$a \mid \upsilon_1 \to \upsilon_2$
External polytypes	ϕ	$::=$	$\texttt{forall}\,\overline{a}\,.\,\upsilon$
Internal monotypes	τ	$::=$	$K \mid \alpha \mid \tau_1 \to \tau_2$
Internal polytypes	σ	$::=$	$\forall\overline{\alpha}\,.\,\tau$
Terms	t	$::=$	$\ldots \mid \backslash(x :: \upsilon)\,.\,t \mid t :: \phi$
Identifier envt	Γ	$::=$	$\varepsilon \mid \Gamma[x : \sigma]$
Type var envt	Φ	$::=$	$\varepsilon \mid \Phi[a \mapsto \tau]$

Type checking $\boxed{\Gamma ; \Phi \vdash t : \tau}$

$$\frac{\overline{a} = fv(\upsilon) \setminus dom(\Phi) \quad \Phi' = \Phi[\overline{a \mapsto \tau}] \qquad \tau_x = \Phi'(\upsilon) \quad \vdash \tau_x \quad \Gamma[x : \tau_x] ; \Phi' \vdash t : \tau_r}{\Gamma ; \Phi \vdash \backslash(x :: \upsilon)\,.\,t : \tau_x \to \tau_r}\ \text{AABS}$$

$$\frac{\overline{K}\ \text{fresh} \quad \tau_b = \Phi[\overline{a \mapsto K}](\upsilon) \qquad \vdash \tau_b \quad \Gamma ; \Phi \vdash t : \tau_b}{\Gamma ; \Phi \vdash (t :: \texttt{forall}\,\overline{a}\,.\,\upsilon) : \Phi[\overline{a \mapsto \tau}](\upsilon)}\ \text{ANNOT}$$

Figure 2: Additional syntax and type-checking rules for sharing-style scoped type variables

```
prefix :: b -> [[b]] -> [[b]]
prefix (x::a) yss
  = let
       xcons :: [a] -> [a]
       xcons ys = x : ys
     in
     map xcons yss
```

In the declaration type signature for `xcons` there is no implicit universal quantification at all, because `a` is in scope. One can have mixtures, of course. We could define `xcons'` like this:

```
xcons' :: b -> [a] -> [a]
xcons' v ys = x : ys
```

Since `b` is not in scope, but `a` is, the implicit-quantification rule means that `b` is quantified, but `a` is not:

```
xcons' :: forall b. b -> [a] -> [a]
```

Whenever a language does something implicitly, it is good practice to provide a way to do it explicitly too. For example, parentheses allow the programmer to make operator precedence explicit, rather than relying on implicit grouping based on operator priorities. So far as quantification is concerned, it seems very desirable to allow the programmer to quantify type signatures explicitly, with `forall`, and GHC supports this. A use of `forall` in a type completely switches off implicit quantification for that type.

4.2 Type checking

Figure 2 shows the additional syntax and type-checking rules required to add scoped type variables to Haskell. All existing syntax and rules remain unchanged.

Our presentation is made simpler if we distinguish between *external types*, (υ, σ, with type variables a), and *internal types* (τ, ρ, with type variables α). Type annotations in the language syntax are written using external types, while the typing judgements use internal types. We will continue to use the term "scoped type variable" to mean the same as "external" type variable. We write $fv(\upsilon)$ to denote the set of all free (external, scoped) type variables in υ.

The basic type-checking judgement form is quite conventional:

$$\Gamma;\Phi \vdash t : \tau$$

which is read "in environments Γ and Φ, term t has type τ". Here, Γ is conventional: it ranges over type contexts, bind each in-scope term variable x to its (internal) type σ.

The context Φ maps each scoped type variable, a, to the monotype, τ, that it represents. It follows that Φ is idempotent, because its range and domain are disjoint. The idea is that when the binding site of a scoped type variable is encountered, Φ is extended to bind that variable to a monotype. Then Φ is applied to all programmer-written types, to replace their scoped type variables by the corresponding monotypes.

Rule AABS type checks a type-annotated λ-abstraction. First, we find $\overline{a}$, the free exernal type variables of υ that are not already bound by Φ; these are the ones that are newly brought into scope. Then we invent arbitrary monotypes $\overline{\tau}$ for each of them, and extend Φ to reflect this choice. Finally, we apply that substitution to υ, giving x's type τ_x, then kind-check τ_x, and type-check the body of the lambda. The judgement $\vdash \tau_x$ checks that the chosen substitution makes τ_x is well-kinded; there is no need for an environment for this judgement, because all the lexically-scoped type variables in υ have by now been substituted away – or, if not, the $\vdash \upsilon$ judgement should fail. As in the SML case, the kind check requires that any type constructors carry their kinds with them.

Rule ANNOT checks that t has the specified external type ϕ. We invent fresh skolem type constants $\overline{K}$, and use them to instantiate the supplied type by applying the extended substitution $\Phi[\overline{a \mapsto K}]$, to get the expected type of the body τ_b. Then we check that τ_b is well-kinded, and that the body t does indeed have that type. Notice that the type variable environment Φ is not extended here. Finally, the supplied type signature is specialised by $\overline{\tau}$ in the result. Haskell allows a polytype, σ, to annotate a term, while SML only permits a monotype. That is why the rule is a bit more elaborate than the corresponding one in Figure 1.

4.3 Extending to Haskell

We have implemented scoped type variables for the full Haskell language. It turns out that there are really no significant complications in scaling up the rules we give to the full Haskell language, including its many extensions.

The only exception is the question of *kind inference* for scoped type variables. In the type checking rule ANNOT, and its corresponding inference rule IANNOT, we took care to check that the user-written type was well-formed, which includes a well-kindedness check. Haskell has higher-kinded type variables, so we must infer the kind of any newly-bound type variables. Our implementation infers these kinds by a separate traversal of the "nearby" type(s) – that is, other type signatures in the same pattern-matching construct.

5 Discussion

We have now seen two designs for lexically-scoped type variables in some detail, so we have enough context to understand something of the design space. The main design choices seem to as follows.

Lexical structure (Section 5.1). Given an occurrence of a scoped type variable in the program text, can one point to the binding occurrence in the program text? If so, we say that the type variable is explicitly bound; otherwise it is bound implicitly.

In the case of explicit binding we take it for granted that alpha-renaming does not change the meaning of the program – that is the very essence of lexical scoping! In the implicit-binding case, though, what is the rule that specifies which set of occurrences of some scoped type variable a may be simultaneously renamed to (say) b?

Type or type variable (Section 5.2)? Does a type variable in the program text stand for a type variable, or for an arbitrary type? In the latter case, can the type be polymorphic?

Quantification (Section 5.3). Does the use of a scoped type variable affect where quantification takes place, or the nature of that quantification (e.g. universal vs existential)? Can two distinct scoped type variable denote the same internal type?

5.1 Lexical structure

The simplest binding choice is to require every lexically-scoped type variable to be bound explicitly by the programmer. That is certainly what happens in explicitly-typed languages that embrace polymorphic types, such as Generic Java and C#. However, both Haskell and all variants of ML have always supported implicit binding of universally-quantified type variables. For example, in Haskell, one writes

```
reverse :: [a] -> [a]
```

rather than the more-explicit

```
reverse :: forall a. [a] -> [a]
```

The scope of these implicit `forall` bindings is very local, however: it is just a single type annotation.

GHC goes no further: all other binding of scoped type variables is explicit, in the sense that one can point to a textual binding site for every other type variable. Furthermore, in the most common case where the type variable is bound as part of a pattern, the scope of such bindings is identical to the scope of term variables bound in the same pattern. Having said that, GHC is a little coy about exactly *which* type variables occurring in patterns are binding occurrences. For example, consider

```
f (x::a) = let g (y::a) = y in g x
```

The first occurrence of `a` in "`(x::a)`" (assuming that `a` is not already in scope), while the second is certainly a bound occurrence. This coyness is a design choice, of course. One could use some lexical notation (a tick, perhaps, thus `'a`) to distinguish a binding site from an occurrence:

```
f (x::a) = let g (y::'a) = y in g x
```

In contrast, SML makes a syntactic distinction between binding sites and mere occurrences. On the other hand, it goes much further in the implicit-binding direction, by giving a rather tricky rule that defines the implicit binding site that encloses two occurrences, as we saw in Section 3.1.

Two other experimental Haskell-like languages, Mondrian [Meijer and Claessen, 1997] and Chameleon [Sulzmann, 2003], use *declaration* type signatures to bring type variables into scope. For example, the type signature

```
prefix :: b -> [[b]] -> [[b]]
```

would bring b into scope in the definition of prefix. One advantage of this is that it can also bring into scope type variables that do not appear in the type of any value, but instead appear only in the constraints:

```
eval :: Eval (a->(b,c)) => a->b
```

Here, the type-class constraint Eval (a->(b,c)) mentions the type variable c that does not appear in the rest of the type at all. These types simply cannot be named using pattern type signatures alone. There are several disadvantages too. First, it couples scoped type variables with polymorphism and type lambdas (see Section 5.3). Second, the type signature in a Haskell program can be arbitrarily far away from the function definition itself, so it may not be easy to find the binding site for a scoped type variable. Lastly, it seems messy to say that a *closed* type, such as

```
forall b. b -> [[b]] -> [[b]]
```

should bring anything into scope elsewhere.

The Caml language displays another variant. In Caml, all implicitly-scoped type variables are brought into scope at the level of the *top-level* (or module-level) definition that mentions it. For example, consider:

```
let pair = (fun (x : 'a) -> x),
           (fun (x : 'a) -> x)
```

Here, 'a is considered to have been brought in scope by the top-level let. Thus, the two 'a are treated as the same type variable, giving pair the type

```
forall 'a. ('a -> 'a) * ('a -> 'a)
```

rather than

```
forall 'a, 'b. ('a -> 'a) * ('b -> 'b)
```

as it would have in GHC.

5.2 Type variable or type?

In the type-lambda approach a scoped type variable stands for a type variable in the program's typing derivation. In the type-sharing approach, as we described it, a scoped type variable stands for an arbitrary monotype.

This difference is more apparent than real, however. A minor variant of the type-sharing approach would insist that a scoped type variable names a *type variable* rather than a *type*. The technical aspects are essentially unchanged, so this choice is primarily a matter of taste. The approach as described allows the programmer to name, say, x's type without having to write down its structure. Arguably, though, if the programmer *writes* something that looks like a type variable she should *get* a type variable.

In any case, the type-sharing approach accommodates both choices, while the type-lambda approach absolutely requires the stands-for-variable choice. Both GHC and Caml, which take the type-sharing approach, currently bind a scoped type variable to a type, rather than a type variable. Chameleon uses different syntax to support both: the signature f :: a -> a means that f has the polymorphic type $\forall a \,.\, a \to a$ and binds a to the argument/result type a in f's body. The signature f ::: a -> a, with three colons, means that f has the monomorphic type $\tau \to \tau$, for some type τ, and binds a to τ in f's body.

5.3 Quantification

Scoped type variables inevitably interact in some way with quantification (notably generalisation), and here there is more variation, and it is harder to classify:

In the type-lambda approach, scoped type variables are inextricably bound up with generalisation. Every scoped type variable is bound at the point at which it is generalised. Several somewhat-unexpected consequences of this choice were explored in Section 3.2.

In the type-sharing approach, in contrast, one can introduce a scoped type variable with no connection whatsoever to generalisation. For example, this expression is perfectly legal:

```
map (\(xs::[a]) -> ys ++ reverse xs)
```

The lambda abstraction passed to map is not generalised (because it is not let-bound), and indeed cannot be (since ys is free), but it is nevertheless perfectly OK to bind a to the type of the elements of xs. Similarly, we may bind type variables in the patterns of a case expression, thus:

```
case v of { (x::a, y) -> (x, x) :: (a,a) }
```

Again, no generalisation is involved. This separation of scoping and quantification is particularly welcome, because it permits *existential*, as well as universal, quantification, (Section 4.1.4).

Nevertheless, there is one subtle interaction between scoped type variables and generalisation. One cannot, of course, generalise over a type variable that is free in the environment. But consider the Haskell binding

```
let { (x, y) = (\v->v, True) } in <body>
```

The left hand side of the binding is a pattern, rather than a function applied to argument patterns, so it is called a *pattern* binding. Pattern bindings are generalised as usual, so GHC will infer the type $\forall a \,.\, a \to a$ for x and Bool for y. Now suppose that we add a type annotation to x:

```
let { (x::(a->a), y) = (\v->v, True) } in <body>
```

What is the scope of a? The same as the scope of x, of course! Hence <body> may mention a, and x cannot be generalised because a is free in the environment. This applies even for degenerate patterns consisting of a single variable. For example, the following definition binds x to a *monomorphic* identity function, whether or not a is actually mentioned in <body>:

```
let { x::(a->a) = \v->v } in <body>
```

Caml has the same rule, that one cannot generalise over a type variable free in the environment, but since in Caml all user-provided type variables are always in scope, this means that they are never generalized in an inner let, but only at the top-level (or module-level) let. For example, this definition

```
let y = let id (x : 'a) = x
        in (id 1, id true)
```

is rejected because the 'a in the type 'a -> 'a for id cannot be generalised in the local let. Leroy writes[3]: "this is the really questionable aspect of the Caml approach, which we'll have to address in the future".

[3]Personal communication

6 Conclusion

As type systems become more sophisticated, the complexity of complete type inference rises very sharply indeed, whether measured in terms of algorithmic complexity, in terms of the subtlety of the code, or in terms of the predictability of the algorithm as seen by the programmer. Rather than design increasingly heroic type-inference algorithms, we believe that it is more productive to use type inference to fill in the gaps between programmer-supplied type annotations. Scoped type variables will become essential to allow such type annotations to be expressed at all.

The main contribution of this paper is to describe and contrast the two main currently-proposed approaches to adding lexically-scoped type variables to Hindley-Milner-style type systems. The type-lambda approach of SML is well specified by the SML Definition but not well known; the type-sharing approach of GHC and Caml is, so far as we know, described only in the respective user manuals of those compilers. Even the central idea, that of regarding a scoped type variable as simply a name for a type, has received little attention.

So which is better? Or would some other variant (Section 5) be superior? We hope that the paper has equipped the reader to better judge for herself, but the authors lean to the type-sharing approach. Its formal expression does seem rather simpler (compare Figures 1 and 2). Furthermore, the decoupling of generalisation and scoped type variables is a big advantage: generalisation is the trickiest area in polymorphic type systems, especially when mutual recursion, the value restriction, value recursion, not to mention type classes in the case of Haskell, and so on, are involved. In fact, our first implementation of scoped type variables in GHC did use the type-lambda approach, but it became horribly complicated to implement and very tricky to describe to the programmer. Switching to the type-sharing approach made matters much simpler; it simply seems to scale better with language complexity.

Be that as it may, we hope that this paper may set a context for futher exploration of an increasingly-important design space.

Acknowledgements

Thanks to Mark Jones for earlier discussions that led to GHC's initial implementation of scoped type variables, and to Xavier Leroy who explained to us how Caml's scoped type variables work. We also thank Nick Benton, Benjamin Pierce, Martin Sulzmann, Claudio Russo, Keith Wansbrough, and Stephanie Weirich for many suggestions that helped improve the paper.

7 References

[Burstall et al., 1980] Burstall, R. M., MacQueen, D. B., and Sannella, D. T. (1980). HOPE: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143.

[Girard, 1990] Girard, J.-Y. (1990). The system F of variable types: fifteen years later. In Huet, G., editor, *Logical Foundations of Functional Programming*. Addison-Wesley.

[Henglein, 1993] Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289.

[Kfoury and Tiuryn, 1992] Kfoury, A. and Tiuryn, J. (1992). Type reconstruction in finite rank fragments of second-order lambda calculus. *Information and Computation*, 98(2):228–257.

[Kfoury et al., 1993] Kfoury, A., Tiuryn, J., and Urzyczyn, P. (1993). Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311.

[Meijer and Claessen, 1997] Meijer, E. and Claessen, K. (1997). The design and implementation of mondrian. In Launchbury, J., editor, *Haskell workshop*, Amsterdam.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *JCSS*, 13(3).

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts.

[Mycroft, 1984] Mycroft, A. (1984). Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer-Verlag.

[Odersky and Läufer, 1996] Odersky, M. and Läufer, K. (1996). Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 54–67. ACM, St Petersburg Beach, Florida.

[Okasaki, 1998] Okasaki, C. (1998). *Purely functional data structures*. Cambridge University Press.

[Okasaki, 1999] Okasaki, C. (1999). From fast exponentiation to square matrices: an adventure in types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 28–35, Paris. ACM.

[Peyton Jones and Shields, 2004] Peyton Jones, S. and Shields, M. (2004). Practical type inference for higher-rank types. Unpublished manuscript.

[Pierce and Turner, 1998] Pierce, B. C. and Turner, D. N. (1998). Local type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego. ACM.

[Sulzmann, 2003] Sulzmann, M. (2003). A Haskell programmer's guide to Chameleon. Available at `http://www.comp.nus.edu.sg/~sulzmann/chameleon/-download/haskell.html`.

[Turner, 1985] Turner, D. (1985). Miranda: A non-strict functional language with polymorphic types. In Jouannaud, J.-P., editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France. Springer-Verlag.