Type inference for arbitrary-rank types is easy

Simon Peyton Jones

Mark Shields

Abstract

Haskell's popularity as a modelling language has driven the need for ever more expressive type system features, most of which push against the decidability and practicality of Damas-Milner style type inference. In this paper we reconsider one of the earliest extensions to Haskell's type system: predicative System-F_ω style polymorphism. Inference for System-F is undecidable at higher-ranks, so we are forced to require some form of programmer-supplied annotations. Reconciling this need for annotations against type inference has been an ongoing topic of research.

We take as our starting point a λ-calculus proposed by Odersky and Läufer. Their system supports arbitrary-ranked polymorphism through the exploitation of type annotations on λ-bound arguments and arbitrary sub-terms. Though elegant, and more convenient than other proposals, Odersky and Läufer's system is potentially very expensive at compile-time, and requires many annotations. We systematically address these two difficulties to yield a type inference system only mildly more complicated than existing implementations of Hindley-Damas-Milner, and of comparable compile-time cost.

[SPJ: Not a good abstract: improve.]

Note: this is a draft of a paper that we plan to submit to the Journal of Functional Programming. It's not quite complete, especially in the final sections, largely because I'm not sure how much more to say. The introduction is also not well written.

I would love your feedback about the paper. Do any sections move too fast? Too slowly? Where do you get lost? Does the main point (that arbitrary-rank type inference is easy) come across convincingly?

The intended audience is quite wide... it's ended up as a tutorial-style paper rather than one intended for type-system insiders. I hope that a graduate student with moderate familiarity with functional programming and compilers would be able to make sense of it. If there are things we can do that would avoid stumbling blocks for people not already familiar with type inference, I'd like to do them. Suggestions?

The code in the Appendix has been through the compiler, but the code in the rest of the paper has not. So look for bugs!

Email: simonpj@microsoft.com

1 Motivation

It is rare to get something valuable for almost nothing, but that is what we offer in this paper. Odersky and Laufer's beautiful paper "Putting Type Annotations to Work" shows how to extend the well-established Damas-Milner system to support so-called *arbitrary-rank types* (Odersky & Läufer, 1996). By requiring the programmer to add certian type annotations, type inference remains possible, but only in a very inefficient way. Building on their insights, we show how an ordinary Damas-Milner-style type inference engine can be extended with very little work to support arbitrary-rank types.

Our contributions are of a practical nature. We have built what we believe is the first full-scale implementation of the Odersky/Laufer idea, in a compiler for Haskell, the Glasgow Haskell Compiler (GHC). Doing so forced us to confront many practical issues; this paper presents the fruits of that experience. More precisely, our contributions are these:

- Higher-rank polymorphism has proved to be useful in Haskell programs, and both the GHC and Hugs implementations of Haskell have supported it for some time, but only in very restricted ways (Section 2.1). Using Odersky and Laufer's ideas has enabled us to lift all these *ad hoc* restrictions without making the type inference engine much more complicated. We had to add around 300 lines of code (some of which replaced code we deleted), and make sundry small, albeit careful, modifications. In the context of a typechecker of some 17,000 lines that is a good result. (All these figures include comments and are necessarily rather approximate.)
- Beautiful though Odersky and Laufer's paper is, it is not for the faint-hearted, nor is it immediately obvious how to implement it. Indeed, to the best of our knowledge it has never been implemented in a full-scale language. We explain the significance of the paper for language designers and implementors.
- The inference algorithm presented by Odersky and Laufer performs generalisation *at every sub-term* of an expression. In contrast, the Damas-Milner algorithm performs generalisation only at `let`/`letrec` bindings, which is a good deal more efficient. We show how to make the Odersky/Laufer approach efficient in the same way.
- Simple type inference engines infer a type for an expression, and subsequently compare that inferred type with the type expected by the expressions's context. Most formal presentations of type systems use this style, including that of Odersky and Laufer. However, dramatically better error messages can be produced if one instead propagates the expected type inward. We show how to adapt the Odersky/Lafer approach to do this.

  Odersky and Laufer require that if a lambda or `case`-bound variable is to get a polymorphic type, it must have an explicit type annotation. We show that the same technique, that of propagating the expected type inward, can be used to relax this restriction in practice, making the system much more convenient to use.
- GHC uses a typed intermediate language, roughly System $F\omega$, and GHC's type inference process adds the extra type abstractions and applications that are missing from the source program, along with the dictionary abstractions and applications necessary to implement type classes. We extend the Odersky/Laufer approach to perform this translation as well. *[SPJ: Not true yet. Should we add stuff for this?]*

- Haskell's type system involves *type classes*, which interact in a significant way with generalisation. We show that the Odersky/Laufer approach generalises smoothly to accommodate type classes. *[SPJ: Not true yet. Should we add stuff for this?]*

All of these contributions relate to the application and implementation of Odersky and Laufer's original idea, but we believe that together they bridge a siginficant gap between concept and implementation. Little that we say is specific to Haskell (with the exception of type classes). Indeed, our conclusion is that virtually any language with a Damas-Milner-style type inference system could (and should) readily be extended in the same way.

2 Motivation and background

We begin by explaining what higher-ranked types are, why one might want them, and why they are not routinely provided in widely-used langauges.

2.1 Higher ranked types

Here is the syntax of types, stratified by rank (Kfoury & Tiuryn, 1992):

$$\begin{array}{llcll}
\text{Monotypes} & \tau, \sigma^0 & ::= & a & \text{Type variable} \\
 & & | & T & \text{Type constant} \\
 & & | & \tau_1 \rightarrow \tau_2 & \text{Function type} \\
 & & & & \\
\text{Polytypes} & \sigma^{n+1} & ::= & \sigma^n & \\
 & & | & \sigma^n \rightarrow \sigma^{n+1} & \\
 & & | & \forall a \,.\, \sigma^{n+1} &
\end{array}$$

A rank-0 type, denoted τ or σ^0, has no $\forall$'s anywhere in it. A *rank-1 type* can have $\forall$'s but only has a rank-0 type to the left of a function arrow. Here are some examples:

$$\begin{array}{rl}
\texttt{Int} \rightarrow \texttt{Int} & \text{Rank 0} \\
\forall a \,.\, a \rightarrow a & \text{Rank 1} \\
\texttt{Int} \rightarrow (\forall a \,.\, a \rightarrow a) & \text{Rank 1} \\
(\forall a \,.\, a \rightarrow a) \rightarrow \texttt{Int} & \text{Rank 2}
\end{array}$$

Throughout this paper we will use the term "*monotype*" for a rank-0 type, and "*polytype*" for a type of rank 1 or greater. (In the literature, the term "type" is often used to mean monotype, but we prefer to be more explicit here.)

The Damas-Milner type system uses rank-1 types only, and then only for `let`-bound variables; lambda-bound variables get only monotypes. Higher-ranked types are turning up increasingly in papers about programming techniques for higher-order typed languages. For example, it is often useful to make a record of polymorphic functions, thus[1]:

[1] We write the quantification explicitly, an extension that is not in Haskell 98 but is supported by several Haskell implementations, including GHC. When writing Haskell program text we will use typewriter-font, "`forall a`"; everywhere else we will use the conventional Greek font "$\forall a$".

```
data Mon m
  = Mon { unit :: forall a. a -> m a,
          bind :: forall a b. m a
                        -> (a -> m b) -> m b
    }
```

Here, a value of type `Mon` τ is a record with two components, each of which is a polymorphic function. The trouble is that the constructor `Mon` has a rank-2 type, and hence is not legal in Haskell 98:

```
Mon :: forall m .
       (forall a. a -> m a)
       -> (forall a b. m a -> (a -> m b) -> m b)
       -> Mon m
```

Pattern-matching on `Mon` values binds polymorphic variables:

```
mIntBool :: Mon m -> m Int -> m Bool -> m (Int,Bool)
mIntBool (Mon { unit = u, bind = b }) mi mb
  = mi  `b`  \x ->
    mb  `b`  \y ->
    u (x,y)
```

The details of the code are unimportant[2]. What matters is that `b`, the `bind` extracted by pattern-matching from `f`'s first argument, is used at two different types, `Int` and `Bool`. So `b` really must have a polymorphic type.

It is tiresome to restrict rank-2 types to constructors only. For example, one might want to write a wrapper function for the `Mon` data constructor that checked some invariant. Furthermore, there are other useful functions that have rank-2 types. For example, the encapsulated state monad `ST` (Launchbury & PeytonJones, 1995) requires a function `runST` with type:

```
runST :: forall a. (forall s. ST s a) -> a
```

`runST` takes a polymorphic value as its argument, and hence has a rank-2 type. Again, this type is not legal Haskell 98.

Another example of a programming technique that makes essential use of rank-2 types is the generic programming of (Laemmel & Peyton Jones, 2003). *[SPJ: Cite other papers here; does anyone have suggestions?]*

When a program contains rank-2 types it is not too long before one wants to have a record with a field that has a rank-2 type, so the record constructor has a rank-3 type. Similarly, one may want to have a function parameter with rank-2 type so the function itself has a rank-3 type. And so on. That said, we have only encountered one genuine case of a rank-3 type, due to Ralf Hinze:

[2] In Haskell, backquotes turn a function into an infix operator; and lambda abstractions exetend as far to the right as possible.

```
type Map f g = forall a b. (a->b) -> f a -> g b

newtype Hfix h a = H (h (Hfix h) a)

hmap :: forall h i. (forall f g. Map f g -> Map (h f) (i g))
                    -> Map (Hfix h) (Hfix i)
```

Because they are so useful, GHC has for several years supported a limited form of rank-2 types, but only with rather *ad hoc* restrictions. For example, partial applications of rank-2 functions were not permitted, because then the unifier might encounter a $\forall$ type, which it cannot handle. These restrictions are hard to document and justify; indeed the whole implementation of rank-2 types in GHC was a bit of a hack. The purpose of this paper is to propose a practical, efficent, and easy-to-implement generalisation of Damas-Milner type inference that solves this problem.

2.2 Type inference for higher-ranked types

Damas-Milner type inference has the remarkable property that a compiler can infer principal types for polymorphic functions, without any help from the programmer. But Damas-Milner stands on a delicate cusp: almost any extension of the type system destroys this unaided-type-inference property.

The big difficulty with higher-ranked types is precisely that they threaten type inference. Indeed, type inference for a program that has no type annotations is known to be undecidable even for rank 2 types (**?**). There are two possible responses:

- Design a cunning algorithm which finds types for many, but not all, typable programs. Kfoury shows how to do this for rank 2 (**?**), but the details are daunting, and the approach becomes unmanageable at higher ranks.
- Make use of type annotations supplied by the programmer to guide the type inference process. This approach has become increasingly popular; for example, it is used to support polymorphic recursion in Haskell (and earlier in Hope and Miranda); to support polymorphism-plus-subtyping using local type inference (Pierce & Turner, 1998); and by Odersky and Laufer to support higher ranked types (Odersky & Läufer, 1996).

We adopt the second approach, which has the major benefit that it can be implemented without major changes to an ordinary Damas-Milner type inference engine, as we shall see. Furthermore, the annotation burden turns out to be modest and, especially where the types become complex, programmers positively *want* to write types as machine-checked documentation.

It is important that the inference algorithm be *preditable*; that is, the programmer should be able to reason about just where to place type annotations, rather than being forced into a trial-and-error dialogue with the type inference engine. Our system has this property of predictability.

2.3 Summary

In summary, rank-2 functions are definitely useful, and we advocate the approach described in this paper even if only as a clean way to offer rank-2 functions. The fact that the approach actually supports functions of arbitrary rank is a welcome bonus.

3 Our design: programmers-eye view

We begin the main thread of the paper by giving an informal, programmer's eye view of our extension to Haskell. The goals of our extension are these:

- Support arbitrary-rank polymorphism.
- Accept that some programmer type annotations are required to guide type inference, but keep them to a minimum.
- Produce decent type error messages.
- Be scalable to a fully-fledged language. In particular, Haskell has (a) type classes, (b) higher-kinded types, (c) an explicitly-typed intermediate language.

To achieve these goals, we make three main extensions to the source language:

(i) We allow types in type signatures to be *explicitly* quantified, rather than permitting only *implicit* quantification as Haskell 98 does. For example, in Haskell 98 one writes:

```
fst :: (a,b) -> a
fst (x,y) = x
```

The type signature for `fst` is implicitly universally quantified over `a` and `b`. We allow the programmer to make that quantification explicit:

```
fst :: forall a b. (a,b) -> a
fst (x,y) = x
```

When we want to write type signatures for rank-2 (or higher) types, explicit quantification is essential, of course.

(ii) We allow the programmer to write types of *arbitrary rank* in type signatures, as shown in the examples in the previous section. The same applies to the types of the fields of constructors, declared in `data` or `newtype` declarations.

(iii) We allow the programmer to supply type signatures on patterns. For example:

```
f = \ (id :: forall a. a->a) -> (id 3, id True)
```

Here the lambda-bound variable `id` is given the type `forall a. a->a`.
A pattern type signature can also bind *lexically-scoped type variables*, although we will not make use of this ability in this paper. The full details are in another paper (Shields & Peyton Jones, 2002).

3.1 Type inference

We remarked earlier on the importance of predictability in type inference. Our system has the following properties:

- For variables bound by `let`, `where`, or top-level bindings, our system will infer a rank-1 type just like the Damas-Milner algorithm, *unless the programmer supplies an explicit type signature for that variable.*
- For all other variables our system will infer only a monotype, unless the variable has a type scheme that can be "easily inferred" from the context of the variable's binding.

By "all other variables" we mean variables bound by lambda, `case`, `do`-notation, list comprehensions, or by the patterns on the left hand side of a function definition. In general, we call these *abstracted variables*, because (unlike, say, `let`-bound variables) the expression to which the variable will be bound is not immediately apparent. Similarly, the patterns in lambda, `case`, and so on, are called *abstracted patterns.*

A variable has an "easily inferred" type scheme if its type can be deduced by propagating type information inwards from the context of variable's binding site. For example:

- If a function definition has an explicit type signature, then the types of its left-hand-side arguments is easily inferred. For example:

  ```
  f :: (forall a. a->a) -> (Int,Bool)
  f id = (id 3, id True)
  ```

- The same holds if the argument is lambda-bound:

  ```
  f :: (forall a. a->a) -> (Int,Bool)
  f = \id -> (id 3, id True)
  ```

  even if it is "hidden" under a `let` or `case`:

  ```
  f :: (forall a. a->a) -> Int
  f = let x = 3 in \id -> (id x, id True)
  ```

- The variables bound by a pattern match have types that is easily inferred from the constructor's definition. For example, in the definition of `mIntBool` of Section 2.1, it is immediately apparent from the definition of contructor `Mon` that `u` has type `forall a. a -> m a`.

In Section 6.5 we give a precise specification of what it means to be "easily inferred", but in practice we expect that the programmer will use simple rules, such as those above, rather than trying to exploit every ounce of inference.

3.2 Subsumption

Suppose that we have variables bound with the following types:

```
k  :: forall a b. a -> b -> b
f1 :: (Int -> Int -> Int) -> Int
f2 :: (forall x. x -> x -> x) -> Int
```

Is the application `(f1 k)` well typed? Yes, it is well-typed in Haskell or ML as they stand; one just instantiates `a` and `b` to `Int`. If we write the type applications explicitly (and GHC's type checker performs precisely this translation) we would get `(f1 (k Int Int))`.

Now, what about the application `(f2 k)`? Even though `k`'s type is not identical to that of `f2`'s argument, this application too should be accpeted. Why? Because `k` is more polymorphic than the function `f2` requires. We can make this argument precise by giving a *witness*, a term that exactly matches `k` to its context, thus:

```
(f2 (/\ x. k x x))
```

The "`/\`" is a type abstraction; we abstract over the type `x` and then apply `k` to `x`, twice, just as we did for `f1`.

So there is a kind of sub-typing going on: an argument is acceptable to a function if its type is more polymorphic than the function's argument type. Odersky and Laufer call the *subsumption* relation. When extended to arbitrary rank, the usual co/contra-variance phenomenon occurs. For example, consider

```
g  :: ((forall x. [x] -> [x]) -> Int) -> Int
k1 :: (forall a. a -> a) -> Int
k2 :: ([Int] -> [Int]) -> Int
```

Precisely because `k1` is more polymorphic than `(forall x. [x]->[x])`, the application `(g k1)` is ill-typed: `k1` requires to be given an argument of type `(forall a. a->a)`, whereas `g` only promises to pass it a (less polymorphic) argument of type `(forall x. [x] -> [x])`. On the other hand, the application `g k2` is well typed; here is the witness:

```
g (\(v::forall x.[x]->[x]). k2 (v Int))
```

The points to remember are these:

- There is a form of subsumption at function application, that has the same kind of co/contra-variance properties as subtyping.
- The witness terms become more complicated as we move to higher-rank types.

3.3 Predicativity

The type system remains *predicative*, just like Haskell. That is, *a type variable ranges over monotypes, not over type schemes*. This allows our system to enjoy the property that, while polymorphic variables may need a type annotation at their *binding site*, they need no annotation at their *call sites*. This property is shared with similar systems, such as local type inference (Pierce & Turner, 1998).

Type variables	a, b			
Term variables	x, y, z			
Integers	i			
Terms	t, u	$::=$	i	Literal
		$\mid$	x	Variable
		$\mid$	`\`p`.` t	Abstraction
		$\mid$	$t\ u$	Application
		$\mid$	`let` x `=` u `in` t	Local binding
		$\mid$	`(`t `::` σ`)`	Type annotation (σ closed)
Patterns	p, q	$::=$	`_`	Wild card
		$\mid$	x	Variable
		$\mid$	`(`p `::` σ`)`	σ closed
Polytypes	σ	$::=$	$\forall\overline{a}.\rho$	
$\forall$-free polytypes	ρ	$::=$	...to be defined...	
Monotypes	τ	$::=$	`Int` $\mid \tau_1 \rightarrow \tau_2 \mid a$	
Type contexts	Γ	$::=$	$\Gamma, x : \sigma \mid \epsilon$	

Fig. 1: Syntax of simplified source types and terms

3.4 Summary

This concludes our informal summary of our language extensions, as seen by the programmer. Next, we turn our attention to type inference.

4 Type inference for Damas-Milner

The purpose of the paper is to show how to adjust a type-inference engine for the well-known Damas-Milner type system to accommodate the much richer type language of arbitrary-rank types. To that end, we first review the Damas-Milner type system and its inference algorithm. To make the discussion concrete, we will develop a complete, executable inference algorithm in Haskell. The source code is available online.

4.1 Notation

We will present all our type systems for a simple language, given in Figure 1. The language of terms is very simple: the lambda calculus augmented with non-recursive `let` bindings, simple pattern matching and type annotations on both terms and binders. This language is carefully chosen to allow us to present the key structural aspects of type inference for higher-rank types with as few constructs as possible. For example, we omit recursive bindings because they introduce no new problems.

The type annotations, on both terms and patterns, are the main unusual feature; indeed one of the points of this paper is to show how they can be used to direct type inference in a simple and predictable way. For the present, we will assume that the type annotations are always *closed*, which is the case in Haskell 98. There are strong reasons to want open type annotations, which require lexically-scoped type variables, but we will avoid that complication in this paper (Shields & Peyton Jones, 2002).

So far as types are concerned, Figure 1 shows only the syntax of *monotypes*, τ, and *polytypes*, σ. Missing is the syntax of $\forall$*-free polytypes*, ρ, which will vary as we progress through the paper.

A type context, Γ, binds a term variable, x, to its polytype σ.

We define $tyvars(\sigma)$ to be the free type variables of σ, and extend the function to type contexts in the obvious way: $tyvars(\Gamma) = \{tyvars(\sigma) \mid (x : \sigma) \in \Gamma\}$.

4.2 The Damas-Milner system

Figure 2 shows the type checking rules for the well-known Damas-Milner type system (Damas & Milner, 1982). In this system, polytypes have rank 1 only, so a ρ type is simply a monotype τ, and hence a polytype σ takes the form $\forall \overline{a} \,.\, \tau$. There are two judgement forms on terms:

$$\Gamma \vdash^{wtr} t : \tau \qquad\qquad \Gamma \vdash^{wts} t : \sigma$$

They mean that in context Γ the term t has monotype τ (respectively polytype σ). ("wtr" is short for "well-typed term with a rho-type" and "wts" for "well-typed term with a sigma-type".) There is a similar judgement form for patterns:

$$\vdash^{pat} p : \tau, \Gamma$$

which means that pattern p has type τ and binds the variables given by Γ. In our little language, Γ will either be empty (for a wild-card pattern) or have just one element (a variable pattern), but of course patterns could easily be more elaborate.

The Figure shows the rules for $\vdash^{wtr}$ in their *syntax-directed* form; that is, there is one rule for each form of term, and the shape of a type derivation is therefore completely defined by the syntax of the term.

In particular, the places where type *generalisation* and *instantiation* take place are completely specified by the rules. Generalisation is handled by the auxiliary judgement $\vdash^{wts}$ which infers a polytype for a term. It is used in rule LET to type the right-hand side of the `let`, and in ANNOT to check that a term $(t\texttt{::}\sigma)$, with a type annotation given by the programmer, indeed has the claimed type. Dually, instantiation is handled by the auxiliary judgement $\vdash^{inst}$, where

$$\Gamma \vdash^{inst} \sigma \leq \rho$$

means that σ is more polymorphic than ρ, or (equivalently) that σ can be instantiated to give ρ. Instantiation is used in rule VAR to specialise the type of a polymorphic variable at its occurrence sites.

Rule INST quietly makes another very important point: *type variables may range only over monotypes.* We can see this from the fact that the type variables in INST are instantiated by τ types, not σ types. This is a difference from System F, which allows quantification over arbitrary types, and it is essential to support inference (Girard, 1990).

$$\forall\text{-free polytypes} \quad \rho \; ::= \; \tau$$

$$\boxed{\Gamma \vdash^{wtr} t : \rho}$$

$$\frac{}{\Gamma \vdash^{wtr} i : \texttt{Int}}\ \text{INT} \qquad \frac{(x:\sigma) \in \Gamma \quad \vdash^{inst} \sigma \le \rho}{\Gamma \vdash^{wtr} x : \rho}\ \text{VAR}$$

$$\frac{\vdash^{pat} p : \tau, \Gamma' \quad \Gamma, \Gamma' \vdash^{wtr} t : \rho}{\Gamma \vdash^{wtr} (\backslash p\,.\,t) : (\tau \to \rho)}\ \text{ABS} \qquad \frac{\Gamma \vdash^{wtr} t : \tau \to \rho \quad \Gamma \vdash^{wtr} u : \tau}{\Gamma \vdash^{wtr} t\ u : \rho}\ \text{APP}$$

$$\frac{\Gamma \vdash^{wts} u : \sigma \quad \Gamma, x:\sigma \vdash^{wtr} t : \rho}{\Gamma \vdash^{wtr} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \text{LET} \qquad \frac{\Gamma \vdash^{wts} t : \sigma' \quad \vdash^{subs} \sigma' \le \sigma \quad \vdash^{inst} \sigma \le \rho}{\Gamma \vdash^{wtr} (t::\sigma) : \rho}\ \text{ANNOT}$$

$$\boxed{\Gamma \vdash^{wts} t : \sigma} \qquad \boxed{\vdash^{inst} \sigma \le \rho}$$

$$\frac{\overline{a} \notin fv(\Gamma) \quad \Gamma \vdash^{wtr} t : \rho}{\Gamma \vdash^{wts} t : \forall \overline{a}\,.\,\rho}\ \text{FORALL} \qquad \frac{\overline{a} \notin ftv(\overline{\tau})}{\vdash^{inst} \forall \overline{a}\,.\,\rho \;\le\; [\overline{a \mapsto \tau}]\,\rho}\ \text{INST}$$

$$\boxed{\vdash^{pat} p : \tau, \Gamma}$$

$$\frac{}{\vdash^{pat} \_ : \tau, \{\}}\ \text{PWILD} \qquad \frac{}{\vdash^{pat} x : \tau, \{x : \tau\}}\ \text{PVAR} \qquad \frac{\vdash^{pat} p : \tau, \Gamma}{\vdash^{pat} (p::\tau) : \tau, \Gamma}\ \text{PANN}$$

$$\boxed{\vdash^{subs} \sigma \le \sigma'}$$

$$\frac{\overline{a} \notin fv(\sigma) \quad \vdash^{subs} \sigma \le \rho}{\vdash^{subs} \sigma \le \forall \overline{a}\,.\,\rho}\ \text{SKOL} \qquad \frac{\overline{a} \notin fv(\overline{\tau}) \quad \vdash^{subs} [\overline{a \mapsto \tau}]\,\rho_1 \le \rho_2}{\vdash^{subs} \forall \overline{a}\,.\,\rho_1 \le \rho_2}\ \text{SPEC} \qquad \frac{}{\vdash^{subs} \tau \le \tau}\ \text{MONO}$$

Fig. 2: The Damas-Milner type system, with type annotations

4.3 Type annotations

Rule ANNOT does not form part of most presentations of the Damas-Milner system, because it deals with type annotations. The term $(t::\sigma)$ is a term that has been annotated by the programmer with a polytype σ. For example, consider the term `(\x -> x) :: (forall a. [a] -> [a])`. This term is well typed, because the most general type of `(\x -> x)` is $\forall a \,.\, a \to a$, and that is certainly more general than $\forall a \,.\, \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]}$. The annotation is a type *restriction*, because the annotated term must only be used at the specified type. For example, this term is illegal:

```
((\x -> x) :: (forall a. [a] -> [a])) (True,False)
```

because, while `(\x -> x)` is applicable to `(True,False)`, the type restriction makes it inapplicable.

Haskell 98 includes this type annotation construct, but we introduce it here mainly as an expository device. It turns out that the typing judgements and inference algorithm for a type-annotated term involves much of the machinery that we will need later for higher-ranked types. Discussing type-annotated terms here allows us to introduce this machinery in the well-understood context of Damas-Milner inference.

Rule ANNOT type-checks a type-annotated term in three stages:

- find t's most general type σ', using $\vdash^{wts}$;
- check that σ' is at least as polymorphic as the programmer-specified type σ, using a new judgement form $\vdash^{subs}$;
- instantiate σ, using $\vdash^{inst}$.

The new judgement form

$$\vdash^{subs} \sigma_{off} \leq \sigma_{req}$$

means "the offered type σ_{off} is more polymorphic than the required type σ_{req}". Unlike $\vdash^{inst}$, it compares two *polytypes*. For example[3]:

$$\begin{array}{rcl}
\texttt{Int} & \leq & \texttt{Int} \\
\texttt{Int} \to \texttt{Bool} & \leq & \texttt{Int} \to \texttt{Bool} \\
\forall a \,.\, a \to a & \leq & \texttt{Int} \to \texttt{Int} \\
\forall a \,.\, a \to a & \leq & \forall b \,.\, \texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]} \\
\forall a \,.\, a \to a & \leq & \forall bc \,.\, (b, c) \to (b, c) \\
\forall ab \,.\, (a, b) \to (b, a) & \leq & \forall c \,.\, (c, c) \to (c, c)
\end{array}$$

The third example involves only simple instantiation, but the last three illustrate the general case. Notice that the number of quantified type variables in the left-hand type can be the same, or more, or fewer, than the in the right-hand type, as the last three examples demonstrate.

It is worth studying carefully the rules for $\vdash^{subs}$, in Figure 2, because they play a central role in this paper. Rule SKOL skolemises the universally-quantified type variables on the right of the $\leq$, while rule SPEC instantiates quantified type variables on the left. Notice that one has to apply SKOL before SPEC, because the latter assumes a ρ type to the right of the $\leq$. For example, to prove that

$$\forall a \,.\, a \to a \quad \leq \quad \forall bc \,.\, (b, c) \to (b, c)$$

[3] Here, and in subsequent examples, we assume we have list and tuple types, written $\texttt{[}\tau\texttt{]}$ and (τ_1, τ_2) respectively.

first use SKOL to skolemise b and c, extending Γ with the new type constants β and γ (say), and *then* use SPEC to instantiate a with the type (β, γ).

Rule PANN only deals with a *montype* annotation in a pattern, whereas the language syntax of Figure 1 allows a *polytype*. This is deliberate: Damas-Milner only attributes a monotype to a lambda-bound variable, so a program with a polytype annotation will simply be rejected as ill-typed.

5 Damas-Milner type inference

The main claim of this paper is that a rather modest overhaul of a vanilla Damas-Milner type inference engine will suffice to support arbitrary-rank polymorphism. To demonstrate this claim convincingly, we now describe how to transcribe the typing rules of Figure 2 into a type inference algorithm. Then, in later sections we will show how to modify this algorithm to support higher-rank polymorphism. Our implementations are written in Haskell. and we assume that the reader is familiar with Haskell including, in particular, the use of monads and `do`-notation. We also assume some familiarity with type inference using unification. The complete source code of our implementations is available in the Appendix, and online.

5.1 Terms and types

The data types `Term` and `Type` are the representation for terms and types respectively, shown in Figure 3, taken from the syntax of Figure 1.

The data types `Sigma` and `Rho` deserves a little more explanation. Here are their definitions, copied from Figure 3:

```
data Sigma = ForAll [Name] Rho

data Rho   = TyVar Name          -- Always bound by a for-all
           | TcTv  TcTv          -- Internal type variable
           | Arrow Rho Rho       -- Functions
           | TcApp TyCon [Type] -- Excludes function types

type Tau = Rho      -- No foralls anywhere
```

We use a distinct data type for σ-types and ρ-types but, in the Damas-Milner system, ρ-types are the same as τ-types. (They will be distinguished when we move to higher rank types.)

It is very convenient to distinguish between *quantified* type variables and *internal* type variables:

A quantified type variable (a) occurs with constructor `TyVar` in a `Rho`, (b) is always bound by an enclosing `ForAll`, (c) may appear in (the type annotations of) a source program, and (d) is represented by a simple `String`. No well-formed `Rho` ever has a free quantified type variable.

An internal type variable is simply a temporary place-holder for an as-yet-unknown monotype: (a) it occurs with constructor `TcTv` in a `Rho`, (b) it is never

```
---------------- Terms -------------------
data Term = Var Name            -- x
          | Lit Int             -- 3
          | App Term Term       -- f x
          | Lam Pat Term        -- \ x -> x
          | Let Name Term Term  -- let { x = f y } in x+1
          | Ann Term Sigma      -- (f x) :: Int

data Pat = PWild                -- _ (wild card)
         | PVar Name            -- x
         | PAnn Pat Sigma       -- x :: Int

---------------- Types -------------------
data Sigma = ForAll [Name] Rho

data Rho   = TyVar Name         -- Always bound by a for-all
           | TcTv  TcTv         -- Internal type variable
           | Arrow Rho Rho      -- Functions
           | TcApp TyCon [Rho]  -- Excludes function types

-- For documentation, it is sometimes nice to say 'Tau'
type Tau = Rho      -- No foralls anywhere

data TyCon = Int | Bool | List | Tuple


(-->) :: Sigma -> Rho -> Rho -- Build a function arrow type
arg --> res = Arrow arg res

---------------- Helper types -------------
type Name = String
```

Fig. 3: The `Term` and `Type` data types

quantified by a `ForAll`, (c) it is created only by the type inference engine itself, and (d) its representation is more complicated as we shall see in Section 5.8. For now, we will treat internal type variables as abstract, of type `TcTv`.

We will use Roman letters, a, b, to denote quantified type variables, and (occasonally) Greek letters, α, β, to denote internal type variables.

Function types are represented using the `TyApp` (type application) constructor, with an `Arrow` type constructor. For notational convenience, we use Haskell's infix notation to define a function `(-->)` which builds function types, so that if `t1` and `t2` are types, then `t1 --> t2` is the type of functions from `t1` to `t2` (see Figure 3). Similarly `intType` is the `Rho` representing the type `Int`.

5.2 The typechecker monad

The type constructor `Tc` is the type-checker monad, whose primitive operations given in Figure 4. The monad serves the following roles:

```
        -- Control flow
fail :: String -> Tc a            -- Type inference can fail

        -- The type environment
lookupVar    :: Name -> Tc Sigma   -- Look up in the envt (may fail)
extendVarEnv :: [(Name,Sigma)] ->  -- Extend the envt
                Tc a -> Tc a

        -- Fresh type variables
newTyVar :: Tc Rho                    -- Make a Type of form (TcTv tv),
                                      -- where tv is a fresh internal tyvar
instTy :: Sigma -> Tc Rho             -- Instantiate a Sigma type with
                                      -- fresh internal type variables
skolTy :: Sigma -> Tc ([TcTv], Rho)   -- Instantiate a Sigma type
                                      -- with fresh skolem constants

        -- Unification
unify     :: Tau -> Tau -> Tc ()    -- Unification (may fail)
unifyFun  :: Rho -> Tc (Sigma,Rho)  -- Decompose a function type
zonkSigma :: Sigma -> Tc Sigma      -- Apply the current substitution

        -- Free type variables
envTvs :: Tc [TcTv]          -- Find the free TcTvs of the envt
rhoTvs :: Rho -> Tc [TcTv]   -- Find the free TcTvs of a type

        -- Quantification
quantify   :: [TcTv] -> Rho -> Tc Sigma
```

Fig. 4: The `TcMonad` module

- It supports exceptions, when type inference fails (see `fail` in Figure 4).
- It carries the environment Γ (`lookupVar` and `extendVarEnv`).
- It allocates fresh type variables (`newTyVar`).
- It maintains a global, ever-growing substitution that supports unification (`unify`, `unifyFun`, and `zonkTy`).

The substitution maps internal type variables (`TcTvs`) to monotypes — it does not affect quantified type variables at all. Unification extends the substitution; for example, `unify t1 t2` extends the substitution so that `t1` and `t2` are identical. Unification can, of course, fail. For example `unify intType (intType --> intType)` will fail. The monad handles the propagation of such failures behind the scenes.

The function `zonkTy` applies the substitution to a type, which can be useful when printing error messages. The remaining functions are introduced in Section 5.6.

5.3 Simple inference

Figure 2 expresses Damas-Milner type system in *syntax-directed form*, which is crucial for eliminating search in type inference. When expressed in this way the rules are tantamount to an algorithm. For each judgement form we have a corresponding

Haskell function, thus:

$$\begin{array}{lrcl}
\vdash^{wtr} & \texttt{inferRho} & :: & \texttt{Term -> Tc Rho} \\
\vdash^{wts} & \texttt{inferSigma} & :: & \texttt{Term -> Tc Sigma} \\
\vdash^{inst} & \texttt{instTy} & :: & \texttt{Sigma -> Tc Rho} \\
\vdash^{subs} & \texttt{subsCheck} & :: & \texttt{Sigma -> Sigma -> Tc ()}
\end{array}$$

We begin by looking at `inferRho`, derived from $\vdash^{wtr}$. Its simplest rule is INT, and its translation is trivial:

```
inferRho (Lit i) = return intType
```

The `return` is necessary to lift `intType` into the `Tc` monad. The rule for applications (APP) is a little more interesting:

```
inferRho (App fun arg)
  = do { fun_ty <- inferRho fun
       ; arg_ty <- inferRho arg
       ; res_ty <- newTyVar
       ; unify fun_ty (arg_ty --> res_ty)
       ; return res_ty }
```

That is, we typecheck the function and argument, create a fresh type variable for the result type, and check that the function type has the right shape. Though the rules are syntax directed, they frequently conjure up monotypes τ out of thin air, in this case the τ for the type of the result. In the implementation we create a fresh internal type variable (using `newTyVar`), relying on unification to fill out its value later. *Remember that these internal type variables each stand for an monotype*; as inference proceeds, unification extends an ever-growing substitution, which maps `TcTv`s to monotypes. When inference is complete, no `TcTv`s remain whatsoever.

Before dealing with the other cases for `inferRho`, we first discuss an important variation of the algorithm.

5.4 Propagating types inward

A type inference engine written in this way turns out to produce absolutely horrible error messages. For example, suppose that the context contains:

```
f :: (Int -> Int) -> Bool
```

and we perform type inference on the application `f (\x.True)`. The inference engine will infer the type `a -> Bool` for the argument `(\x.True)`, and then it will attempt the following unification:

```
((Int -> Int) -> Bool)  =  ((a -> Bool) -> r)
```

The unification will fail, but with a rather opaque error message.

No human would do this when doing mental type inference. We know the type of `f`, and we use that information when performing inference on `f`'s argument. This simple intuition leads to a very well-known technique for improving the error messages, namely to *propagate the expected type inwards.* More concretely, we make a variant of `inferRho`, called `checkRho`, thus:

```
checkRho :: Term -> Rho -> Tc ()
```

Instead of returning the inferred type as its result, `checkRho` now *takes the expected type as an argument.* We can recover the old `inferRho` by passing in a type variable:

```
inferRho :: Term -> Tc Rho
inferRho expr = do { exp_ty <- newTyVar
                   ; checkRho expr exp_ty
                   ; return res_ty }
```

The type variable `exp_ty` (short for "expected type") plays the role of a `var` parameter in Pascal: it serves as a location in which `checkRho` can return its result. This inward-propagation technique is well known to implementors as "Algorithm M"(Lee & Yi, 1998). We review it here because it will play a fundamental role in our type system; it is necessary not just to improve error messages, but to make the system work at all.

Here, for example is the `App` case for `checkRho`, to compare with the code for the same case of `inferRho` in the previous section:

```
checkRho (App fun arg) exp_ty
  = do { fun_ty <- inferRho fun
       ; (arg_ty, res_ty) <- unifyFun fun_ty
       ; checkRho arg arg_ty
       ; unify res_ty exp_ty }
```

First, we infer the type of the function. We expect it to return a function type, which we split up using `unifyFun` (to be defined shortly), yielding the argument and result type of the function. Now we type-check the argument *passing in the expected type of the argument, derived from the function*; and finally we unify the function's result type with the expected result type `exp_ty`. Not only are the error messages better, but the code is shorter too[4]!

The function `unifyFun` splits a function type, returning the argument and result types of the function; it may fail, raising an exception, if the argument is not a function type.

```
unifyFun :: Rho -> Tc (Rho, Rho)
unifyFun (Arrow arg_ty res_ty) = return (arg_ty,res_ty)
unifyFun fun_ty = do { arg_ty <- newTyVar
                     ; res_ty <- newTyVar
                     ; unify fun_ty (arg_ty --> res_ty)
                     ; return (arg_ty,res_ty) }
```

[4] Exercise: rewrite the `App` case of `checkRho` to use one line fewer, and without using `unifyFun`. We chose to use the form given here because it anticipates what we need in Section 7.

First, it checks whether `fun_ty` is already of the form `(arg_ty -> res_ty)`, in which case it returns the pair. If not, `unifyFun` creates fresh type variables for `arg_ty` and `res_ty` and attempts to unify `(arg_ty -> res_ty)` with the `fun_ty`. The first equation is only present for efficiency reasons; it could be omitted without affecting correctness.

5.5 Abstraction and patterns

The code for lambda abstraction uses `unifyFun` in a dual manner to split the expected type into the type of the pattern in the abstraction and the type of the body:

```
checkRho (Lam pat body) exp_ty
  = do { (pat_ty, body_ty) <- unifyFun exp_ty
       ; binds <- checkPat pat pat_ty
       ; let binds' = [(n, ForAll [] rho) | (n,rho) <- binds]
       ; extendVarEnv binds' (checkRho body body_ty) }
```

Then we call `checkPat` to typecheck the pattern, returning a list of `(Name,Rho)` bindings which bind all the variables in the pattern; extend the type environment with these new bindings (`extendVarEnv`); and then type-check the body. All this follows directly from LAM of Figure 2.

The function `checkPat` corresponds to $\vdash^{pat}$ just as `checkRho` corresponds to $\vdash^{wtr}$. It takes an expected type for the pattern (like `checkRho`), and returns a list of `(Name,Rho)` bindings. In our little language, a pattern always binds one variable or none, but if we were to add pairs, say, it could bind any number, so a list of bindings is the appropriate return type. Here, then, is `checkPat`:

```
checkPat :: Pat -> Rho -> TcM [(Name,Rho)]
checkPat PWild       exp_ty = return []
checkPat (PVar v)    exp_ty = return [(v, exp_ty)]
checkPat (PAnn p ty) exp_ty = do { unify ty exp_ty
                                 ; checkPat p ty }
```

For the Damas-Milner system we assume that the type annotation on a lambda-bound variable is a monotype, and so can be given to `unify`. For example, this is legal:

```
(\(x::Int). x+1)
```

but this is not:

```
(\(x::forall a. a->a). x True)
```

We will lift this restriction later in the paper.

5.6 Instantiation and generalisation

When we reach a variable (rule INST), we look it up in the environment (failing if it is not in scope), instantiate its type with fresh internal type variables, and then check that the resulting type is compatible with the expected type `exp_ty`:

```
checkRho (Var v) exp_ty = do { v_sigma <- lookupVar v
                             ; v_rho   <- instTy v_sigma
                             ; unify v_rho exp_ty }
```

Now we consider `let` bindings, which is where type generalisation occurs (LET):

```
checkRho (Let v rhs body) exp_ty
  = do { v_sigma <- inferSigma rhs
       ; extendVarEnv [(v,v_sigma)] (checkRho body exp_ty) }
```

We use `inferSigma` to infer the (polymorphic) type of `rhs`. Here is its implementation, which can be read directly from the $\vdash^{wts}$ judgement in Figure 2:

```
inferSigma :: Term -> Tc Sigma
inferSigma e = do { res_ty  <- inferRho e
                  ; res_tvs <- rhoTvs res_ty
                  ; env_tvs <- envTvs
                  ; let forall_tvs = res_tvs \\ env_tvs
                  ; quantify forall_tvs res_ty }
```

The functions `rhoTvs` and `envTvs` find the free internal variables of `exp_ty` and the (monad-carried) environment Γ, respectively (Figure 4). They both take account of the current substitution, returning a set of `TcTvs` (here represented as a list). We quantify over `forall_tvs`, the difference of these two sets, computed using the list-difference operator `(\\)`:

```
quantify :: [TcTv] -> Rho -> Tc Sigma
```

When we quantify, we can turn an internal `TcTv` into a quantified type variable. For example, consider the `Rho`

```
Arrow (TcTv t) (TcTv t)
```

where `t::TcTv`, and suppose we decide to quantify over `t`. Then `quantify` will return the `Sigma`

```
ForAll ["t"] (Arrow (TyVar "t") (TyVar "t"))
```

where the name `"t"` is chosen arbitrarily[5].

Why does `quantify` have a monadic type? Because `res_ty` only makes sense in the context of the substitution, which is carried by the monad. `quantify` guarantees to return a type that is fully substituted; this makes it easier to instantiate later, because the proper type variables can all be found without involving the substitution.

[5] Well, almost arbitrarily: it must not conflict with any proper type variable names already inside the type we are quantifying over. This is not an issue for Damas-Milner, since all the for-alls are at the top.

5.7 Subsumption

The code for a type-annotated expression can be read off Figure 2 just like the other cases:

```
checkRho (Ann body ann_ty) exp_ty
   = do { body_sigma_ty <- inferSigma body
        ; subsCheck body_sigma_ty ann_ty
        ; rho_ty <- instTy ann_ty
        ; unify rho_ty exp_ty }
```

The interesting part is the implementation of `subsCheck`, which implements the $\vdash^{subs}$ judgement (Figure 2). Here is the implementation:

```
subsCheck :: Sigma -> Sigma -> Tc ()
subsCheck sigma1 sigma2   -- Rule SKOL
  = do { (skol_tvs, rho2) <- skolTy sigma2
       ; subsCheck1 sigma1 rho2 }

subsCheck1 :: Sigma -> Rho -> Tc ()
subsCheck1 sigma1 rho2    -- Rule INST
  = do { rho1 <- instTy sigma1
       ; subsCheckRho rho1 rho2 }

subsCheckRho :: Rho -> Rho -> Tc ()
subsCheckRho rho1 rho2    -- Rule MONO
  = unify rho1 rho2
```

(We have written a function for each rule, to keep them clearly separated, but in fact we can inline both `subsCheck1` and `subsCheckRho` into the single function `subsCheck`.) The functions `subsCheck1` and `subsCheckRho`, corresponding to rules INST and MONO are entirely straightforward, but `subsCheck` itself (corresponding to rule SKOL) is a little tricky.

In `subsCheck`, we want to instantiate σ_2 with fresh type variables, but (unlike the normal case) these type variables *do not stand for arbitrary monotypes.* In fact, they stand for fresh "skolem constants", new type constants distinct from all other types. There are several ways to implement this idea. The one we choose here is to define a variant of `instTy`, namely `skolTy`:

```
skolTy :: Sigma -> Tc ([TcTv], Rho)
```

The idea is that `skolTy` instantiates the outer `ForAll` of its argument, replacing the quantified type variables with "rigid" internal type variables. These rigid type variables still have type `TcTv`, *but they will not unify with anything except themselves*, so they are, in effect, skolem constants. The `skolTy` function returns the newly-made rigid type variables, along with the instantiated type. Then all we need do is to call `subsCheck1` and we are done.

An extremely alert reader will realise the correctness of this implementation of rule SKOL depends on the fact that type annotations in our source program are *closed* (have no free type variables). In reality, there are strong reasons to support lexically-scoped type variables, which allow us to write open type annotations (Shields & Peyton Jones, 2002), and in any case the same problem shows up when we move to higher rank. However, with the source language as currently defined everything is OK; we will return to the issue in Section 7.3.

Before leaving `subsCheck`, it is worth noting that it has the same type as `unify`, except that it applies to σ-types, and degenerates to `unify` when applied to monotypes. So we can think of `subsCheck` as a kind of super-unifier.

5.8 Internal type variables and the `Tc` *monad*

So far we have not said how internal type variables are represented, nor how the `Tc` monad works. In this section we briefly describe them; the full code is in the Appendix.

An internal type variable, of type `TcTv` is represented like this:

```
data TcTv = Tv Uniq TRef TvFlavour

type TRef = IORef (Maybe Tau)
        -- 'Nothing' means the type variable is not substituted
        -- 'Just ty' means it has been substituted by 'ty'

data TvFlavour = Flexible      -- Can unify with any tau-type

               | Rigid Name    -- Must not unify with anything
                               -- except a flexible type variable
                               -- The Name just improves error messages
type Uniq = Int
```

A `TcTv` has a unique identity, which is just an `Int`; it has a "flavour", which distinguishes ordinary internal type variables from rigid ones (see Section 5.7); and it has a mutable slot of type `TRef`.

The `TRef` is a mutable reference cell that either contains `Nothing`, indicating that type variable is not in the domain of the substitution, or contains `Just ty`, indicating that the type variable is mapped to the type `ty` by the substitution, where `ty` is a monotype. We use an `IORef` for the mutable cell, so any operations that read or write this cell must be in the `IO` monad. (See (Peyton Jones, 2001) for a tutorial on the `IO` monad.) That is why many of our operations over types — for example `rhoTvs` — are in the `Tc` monad.

The `Tc` monad, then, is the `IO` monad augmented with (a) an environment, and (b) a way to report failure[6]

```
newtype Tc a = Tc (TcEnv -> IO (Either ErrMsg a))
```

[6] The latter could be done via an exception in the `IO` monad, but we have elected to make failure more explicit here.

Throughout, we maintain the following invariant:

- *An internal type variable can only be substituted to a τ-type*

This invariant is really important. For example, suppose `f` :: α`->`α, where α is an internal type variable. If we see the expression `(f 'c', f True)`, we will first unify α with `Char` and then with `Bool`, and will fail with a type error. But if α might subsequently be unified with $\forall b$. b`->`b, this failure would have been premature. Our invariant keeps things simple.

[SPJ: I'm beginning to wonder what would go wrong if we weakened the invariant, so that an internal type variable could only be substituted by a rho-type??]

6 Higher-rank types

We now turn our attention from the well-established Damas-Milner type system to the system of arbitrary-rank types proposed by Odersky and Laufer (Odersky & Läufer, 1996). Figure 5 presents the Odersky/Laufer type checking rules for our term language. To save clutter, in this figure and the next we omit wild-card patterns, and collapse the $\vdash^{pat}$ pattern-checking judgement into $\vdash^{wtr}$.

The figure begins by defining $\forall$-free polytypes, ρ, to complete the syntax of types:

$$\forall\text{-free polytypes} \quad \rho \quad ::= \quad \tau \mid \sigma_1 \to \sigma_2$$

Crucially, a polytype may appear in both the argument and result positions of a function type, and hence polytypes may be of *arbitrary rank*. Providing this freedom is the whole point of this paper.

The rules look rather different to those of the Damas-Milner system in Figure 2:

- There is just one judgement form, $\vdash^{wts}$, which checks whether a term has a *polytype*, σ.
- The rules are not syntax directed: any sub-term may be generalised by FORALL, and specialised by SUBS.
- A function may accept an argument of polytype if its λ-bound variable is annotated accordingly (rule AABS). Otherwise, functions are from monotypes to polytypes (rule ABS).

The type-subsumption judgement, $\vdash^{subs}$, was discussed in Section 3.2, and appears again here, but with one crucial generalisation: it has an extra rule (FUN) which allows it to "look inside" functions in the usual co- and contra-variant manner. Adding this single rule allows us to instantiate deeply nested quantifiers, rather than only outermost quantifiers. For example, we can deduce that:

$$\begin{array}{rcl}
\texttt{Int} \to (\forall a \ . \ a \to a) & \leq & \texttt{Int} \to \texttt{Int} \\
\texttt{Int} \to \texttt{Int} & \leq & (\forall a \ . \ a \to a) \to \texttt{Int} \\
(\forall b \ . \ \texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]}) \to \texttt{Bool} & \leq & (\forall a \ . \ a \to a) \to \texttt{Bool}
\end{array}$$

None of these types would have been syntactically legal in the Damas-Milner system.

$$\forall\text{-free polytypes} \quad \rho \ ::= \ \tau \mid \sigma \rightarrow \sigma'$$

$$\boxed{\Gamma \vdash^{wts} t : \sigma} \quad \text{(V1)}$$

$$\frac{}{\Gamma \vdash^{wts} i : \texttt{Int}}\ \text{INT} \qquad \frac{}{\Gamma, (x : \sigma) \vdash^{wts} x : \sigma}\ \text{VAR}$$

$$\frac{\Gamma, x : \tau \vdash^{wts} t : \sigma}{\Gamma \vdash^{wts} (\backslash x.\ \ t) : (\tau \rightarrow \sigma)}\ \text{ABS} \qquad \frac{\Gamma, x : \sigma \vdash^{wts} t : \sigma'}{\Gamma \vdash^{wts} (\backslash (x \texttt{::} \sigma)\texttt{.}\ \ t) : (\sigma \rightarrow \sigma')}\ \text{AABS}$$

$$\frac{\Gamma \vdash^{wts} t : (\sigma \rightarrow \sigma') \quad \Gamma \vdash^{wts} u : \sigma}{\Gamma \vdash^{wts} t\ u : \sigma'}\ \text{APP} \qquad \frac{\Gamma \vdash^{wts} u : \sigma \quad \Gamma, x : \sigma \vdash^{wts} t : \sigma'}{\Gamma \vdash^{wts} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \sigma'}\ \text{LET}$$

$$\frac{\Gamma \vdash^{wts} t : \sigma}{\Gamma \vdash^{wts} (t\ \texttt{::}\ \sigma) : \sigma}\ \text{ANNOT}$$

$$\frac{\overline{a} \notin fv(\Gamma) \quad \Gamma \vdash^{wts} t : \rho}{\Gamma \vdash^{wts} t : \forall \overline{a}\ .\ \rho}\ \text{FORALL} \qquad \frac{\Gamma \vdash^{wts} t : \sigma' \quad \vdash^{subs} \sigma' \leq \sigma}{\Gamma \vdash^{wts} t : \sigma}\ \text{SUBS}$$

$$\boxed{\vdash^{subs} \sigma \leq \sigma'}$$

$$\frac{\overline{a} \notin fv(\sigma) \quad \vdash^{subs} \sigma \leq \rho}{\vdash^{subs} \sigma \leq \forall \overline{a}\texttt{.}\rho}\ \text{SKOL} \qquad \frac{\overline{a} \notin fv(\overline{\tau}) \quad \vdash^{subs} [\overline{a \mapsto \tau}]\ \rho_1 \leq \rho_2}{\vdash^{subs} \forall \overline{a}\texttt{.}\rho_1 \leq \rho_2}\ \text{INST}$$

$$\frac{\vdash^{subs} \sigma_3 \leq \sigma_1 \quad \vdash^{subs} \sigma_2 \leq \sigma_4}{\vdash^{subs} (\sigma_1 \rightarrow \sigma_2) \leq (\sigma_3 \rightarrow \sigma_4)}\ \text{FUN} \qquad \frac{}{\vdash^{subs} \tau \leq \tau}\ \text{MONO}$$

Fig. 5: The Odersky-Laufer type system

6.1 Avoiding Generalisation

There is a serious practical problem with using the typing rules of Figure 5 as a basis for type inference: *they are not syntax-directed.* In particular, rule FORALL allows us to generalise anywhere, and rule SUBS allows us to specialise anywhere.

The Damas-Milner idea is to specialise at variable occurrences, and generalise at `let`s, but that does not work at all. Suppose that `f` has the following type:

$$\texttt{f} :: (\forall a\ .\ a \rightarrow (\forall b\ .\ b \rightarrow (a, b))) \rightarrow \texttt{Int}$$

and consider the application `f (\x.\y. (x,y))`. Is this well typed? The obvious type one might infer for `f`'s argument is this:

$$\texttt{(\textbackslash x.\textbackslash y. (x, y))} :: a \rightarrow b \rightarrow (a, b)$$

for some types `a` and `b`. That obviously will not do because `f` needs a polymorphic argument. Let's try generalising, thus:

$$\texttt{(\textbackslash x.\textbackslash y. (x, y))} :: \forall a, b \,.\, a \rightarrow b \rightarrow (a, b)$$

The argument certainly has this type but alas, under the subsumption rules of Figure 5, it is simply not the case that

$$\forall a, b \,.\, a \rightarrow b \rightarrow (a, b) \quad \leq \quad \forall a \,.\, a \rightarrow (\forall b \,.\, b \rightarrow (a, b))$$

Our mistake was to generalise too late, at the outside of the argument term. Instead, we must generalise as early as possible. In our example, we must generalise the type of the sub-term:

$$\texttt{(\textbackslash y. (x,y))} :: \forall b \,.\, a \rightarrow b \rightarrow (a, b)$$

Then we generalise further outside the `\x` abstraction:

$$\texttt{(\textbackslash x.\textbackslash y. (x,y))} :: \forall a \,.\, a \rightarrow (\forall b \,.\, b \rightarrow (a, b))$$

The only way to turn the rules of Section 4 into a type inference algorithm is to *generalise eagerly, at every node in the syntax tree*, and then use rule SUBS at every function application, to match up the type expected by the function with the offered argument. Odersky and Laufer give just such an inference algorithm in their paper (Odersky & Läufer, 1996). Unfortunately, once we have scaled the calculus up to all of Haskell, generalization turns out to be quite expensive at compile-time. It requires types and constraints to be traversed, the constraint simplifier to be invoked, and a coercion term to be constructed that converts a term of one type to a term of the other. Alternating generalisation and specialisation at every node of the syntax tree would lead to ridiculously large[7] coercion terms.

One might be prepared to accept this high cost in return for a more expressive type system, but such is not the case. In the absence of type annotations, *no program context can distinguish the above two types.* Even if we were to partially apply the term to a single argument, we could only distinguish the two types if the partial application is in turn applied twice, to terms of different types. But that implies the partial application is let-bound, in which case the partial application will have been re-generalized, in which case either type assignment leads to the same type for the partial application.

6.2 Prenex-form polytypes

This observation gives a useful clue. If there is no useful difference between the type $(\sigma_1 \rightarrow (\forall a \,.\, \sigma_2))$ and $(\forall a \,.\, (\sigma_1 \rightarrow \sigma_2))$, then let us simply outlaw the former. *Our*

[7] Ridiculous because most of them would turn out to be the identity function.

$$\forall\text{-free polytypes} \quad \rho \quad ::= \quad \tau \mid \sigma \to \rho$$

$$\boxed{\Gamma \vdash^{wtr} t : \rho}$$

$$\frac{}{\Gamma \vdash^{wtr} i : \texttt{Int}}\ \text{INT} \qquad \frac{\vdash^{inst} \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash^{wtr} x : \rho}\ \text{VAR}$$

$$\frac{\Gamma, x : \tau \vdash^{wtr} t : \rho}{\Gamma \vdash^{wtr} (\backslash x.\ t) : (\tau \to \rho)}\ \text{ABS} \qquad \frac{\begin{array}{c}\vdash^{wks} \sigma \\ \Gamma, x : \sigma \vdash^{wtr} t : \rho\end{array}}{\Gamma \vdash^{wtr} (\backslash (x::\sigma).\ t) : (\sigma \to \rho)}\ \text{AABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{wtr} t : \sigma \to \rho \\ \Gamma \vdash^{wts} u : \sigma' \\ \vdash^{subs} \sigma' \leq \sigma\end{array}}{\Gamma \vdash^{wtr} t\ u : \rho}\ \text{APP} \qquad \frac{\begin{array}{c}\Gamma \vdash^{wts} u : \sigma \\ \Gamma, x : \sigma \vdash^{wtr} t : \rho\end{array}}{\Gamma \vdash^{wtr} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \text{LET}$$

$$\frac{\begin{array}{c}\vdash^{wks} \sigma \\ \Gamma \vdash^{wts} t : \sigma' \\ \vdash^{subs} \sigma' \leq \sigma \\ \vdash^{inst} \sigma \leq \rho\end{array}}{\Gamma \vdash^{wtr} (t\ ::\ \sigma) : \rho}\ \text{ANNOT}$$

$$\boxed{\Gamma \vdash^{wts} t : \sigma} \qquad \boxed{\vdash^{inst} \sigma \leq \rho}$$

$$\frac{\begin{array}{c}\overline{a} \notin fv(\Gamma) \\ \Gamma \vdash^{wtr} t : \rho\end{array}}{\Gamma \vdash^{wts} t : \texttt{forall}\ \overline{a}.\rho}\ \text{FORALL} \qquad \frac{\overline{a} \notin fv(\overline{\tau})}{\vdash^{inst} \texttt{forall}\ \overline{a}.\rho\ \leq\ [\overline{a \mapsto \tau}]\ \rho}\ \text{INST}$$

Fig. 6: Modified Odersky-Laufer system, using prenex types

first refinement is to restrict the syntax of polytypes to disallow quantification in the right-hand side of a function type. That is, a ρ type takes the form:

$$\forall\text{-free polytypes} \quad \rho \quad ::= \quad \tau \mid \sigma \to \rho$$

As before, a ρ type has no top-level $\forall$ quantifiers, but (unlike Figure 5) *nor can it have a $\forall$ to the right of a function arrow.* A polytype is therefore syntactically forced to be in *prenex form*, with all its quantifiers at the front. For example, the type

$$\forall a\ .\ a \to (\forall b\ .\ a \to b)$$

is now illegal; instead we must move the "$\forall b$" to the front, to give

$$\forall ab\ .\ a \to a \to b$$

In general, we can always move the $\forall$'s to the front, to transform a general arbitrary-rank type into the prenex form of this section.

The resulting type structure embodies almost exactly the same structure as System Λ_2^- described by Kfoury and Tiuryn (Kfoury & Tiuryn, 1992). Their goal was to precisely characterise the decidablity or otherwise of type inference for finite rank fragments of System F, but they found Λ_2^- more convenient to work with. And so do we!

Since we only consider a *predicative* system, so that we can only instantiate a type variable with a monotype, we do not need to worry that instantiating (say) the identity function, of well-formed type $\forall a\ .\ a \to a$ at the well-formed type $\forall b\ .\ b \to b$ might lead to the ill-formed type $(\forall b\ .\ b \to b) \to (\forall b\ .\ b \to b)$. Kfoury and Tiuryn have a looser restriction than ours: they allow a type variable to be instantiated only with a $\forall$-free polytype, ρ, which may have nested quantifiers.

Kfoury and Tiuryn prove that Λ_2^- has the same expressive power as Λ_2 (the rank-2 fragment of System F). We conjecture, but have not proved, that this result transfers to our system (which has arbitrary rank, but is predicative). In short, restricting the syntax of types does not change programs we can write.

6.3 Type checking rules for prenex types

Figure 6 presents the type checking rules for the refined system, apart from the subsumption judgement, $\vdash^{subs}$, which remains unchanged from Figure 5. These rules already look far more like the rules of the Damas-Milner type system in Figure 2. The main judgement form $\vdash^{wtr}$ checks whether a term has a type ρ which has no top-level $\forall$ quantifiers, just like Figure 2. Another similarity is the separate judgements $\vdash^{inst}$ and $\vdash^{wts}$ which specify exactly where specialisation and generalisation take place, just like the Damas-Milner system. Notice that $\vdash^{inst}$ strips off, and $\vdash^{wts}$ glues on, only *top-level* quantifiers. In contrast, the (unchanged) judgement $\vdash^{subs}$ deals with nested quantification.

6.4 Propagating types inwards

So much for similarities to the Damas-Milner system. The key difference is in rule APP. Since the argument type of the function is a polytype, σ, we first use $\vdash^{wts}$ to deduce a polymorphic type, σ', for u, and then $\vdash^{subs}$ to check that σ' is polymorphic enough to match σ.

In concrete algorithmic terms, rule APP is still far from satisfactory. To transliterate it directly in to an algorithm one would (a) infer the most polymorphic type for the argument, and then (b) call a subsumption checker to check that the argument was polymorphic enough. Suppose we have

$$\begin{array}{lll} \texttt{f} & :: & (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \\ \texttt{id} & :: & \forall a\ .\ a \to a \end{array}$$

and we encounter the call (f id). Then rule VAR will instantiate id, $\vdash^{wts}$ will immediately re-generalise it, and $\vdash^{subs}$ will specialise it again! (This is also true, incidentally of the handling of type annotation in Figure 2 but it seemed less important there, because type annotations are relatively rare. Function application, however, is rather common!)

In a nutshell, our problem is this: the call to $\vdash^{wts}$ in rule APP is ignorant of the argument type required by the function, so it must generalise the argument as much as possible, just in case the function needs a very polymorphic argument. But most of the time the function will want a boring monotyped argument, so the generalisation turns out to be in vain.

The obvious solution is this: re-arrange $\vdash^{wts}$ so that it "knows" the expected type. Stated like this, there is an immediate analogy with Section 5.4, where we describe how advantageous it can be to push types inwards during inference. Perhaps we could re-use the same technique? Indeed, there are other advantages of doing so. Consider this definition:

```
foo :: (forall a. a -> a) -> (Int, Bool)
foo = \i -> (i 3, i True)
```

(Here, we informally extend our tiny calculus with separate type signatures for functions.) It is "obvious" that the type of "`i`" is $(\forall a\ .\ a \rightarrow a)$, but it is not explicitly annotated as such. It is obvious to us because we propagate the known type for `f` into `f`'s body. Aha! Pushing types inwards again!

6.5 Typing rules for pushing types inwards

With this in mind, Figure 7 gives typing rules that express the idea of propagating types inwards. We previously treated the idea as an implementation issue, but now it affects what programs are well typed — for example, we want `foo` to be well typed despite the absence of a type annotation on `i` — so the apparatus of typing judgements is appropriate.

Figure 7 describes two very similar typing judgements.

$$\Gamma \vdash^{wtr}_{\Uparrow} t : \rho$$

means "in context Γ the term t can be *inferred* to have type ρ", whereas

$$\Gamma \vdash^{wtr}_{\Downarrow} t : \rho$$

means "in context Γ, the term t can be *checked* to have type ρ". The up-arrow $\Uparrow$ suggests pulling a type up out of a term, whereas the down-arrow $\Downarrow$ suggests pushing a type down into a term. The judgement $\vdash^{wts}$ is generalised in the same way.

Some of the rules are "polymorphic" in the direction δ. For example, the rules LET and INT are insensitive to δ, and can be seen as shorthand for two rules that differ only in the arrow direction. In a real language there are quite a few such constructs (`case` and `if` are other examples) so the notational saving is quite worth while.

All the other rules split into two, one for each direction. Rule VAR1, for example, deals with the inference case: we just instantiate the outer $\forall$ quantifiers using $\vdash^{inst}$ as before. The other case, VAR2, is more interesting. Here, we are pushing inward

$$\forall\text{-free polytypes} \quad \rho \ ::= \ \tau \mid \sigma \ \texttt{->} \ \rho$$

$$\boxed{\Gamma \vdash^{wtr}_{\delta} \ t : \rho} \qquad \delta ::= \Uparrow \mid \Downarrow$$

$$\frac{\vdash^{inst} \ \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash^{wtr}_{\Uparrow} \ x : \rho}\ \textsc{var1} \qquad \frac{\vdash^{subs} \ \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash^{wtr}_{\Downarrow} \ x : \rho}\ \textsc{var2}$$

$$\frac{\begin{array}{c}\vdash^{wtr}_{\Uparrow} \ p : \sigma, \Gamma' \\ \Gamma, \Gamma' \vdash^{wtr}_{\Uparrow} \ t : \rho\end{array}}{\Gamma \vdash^{wtr}_{\Uparrow} \ (\backslash p \ \ . \ \ t) : (\sigma \ \texttt{->} \ \rho)}\ \textsc{abs1} \qquad \frac{\begin{array}{c}\vdash^{wtr}_{\Downarrow} \ p : \sigma, \Gamma' \\ \Gamma, \Gamma' \vdash^{wtr}_{\Downarrow} \ t : \rho\end{array}}{\Gamma \vdash^{wtr}_{\Downarrow} \ (\backslash p \ \ . \ \ t) : (\sigma \ \texttt{->} \ \rho)}\ \textsc{abs2}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{wtr}_{\Uparrow} \ t : \sigma \ \texttt{->} \ \rho \\ \Gamma \vdash^{wts}_{\Downarrow} \ u : \sigma\end{array}}{\Gamma \vdash^{wtr}_{\Uparrow} \ t \ u : \rho}\ \textsc{app1} \qquad \frac{\begin{array}{c}\Gamma \vdash^{wtr}_{\Uparrow} \ t : \sigma \ \texttt{->} \ \rho \\ \Gamma \vdash^{wts}_{\Downarrow} \ u : \sigma \Gamma \vdash^{subs} \ \rho \leq \rho'\end{array}}{\Gamma \vdash^{wtr}_{\Downarrow} \ t \ u : \rho'}\ \textsc{app2}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{wts}_{\Downarrow} \ t : \sigma \\ \vdash^{inst} \ \sigma \leq \rho\end{array}}{\Gamma \vdash^{wtr}_{\Uparrow} \ (t \ \texttt{::} \ \sigma) : \rho}\ \textsc{annot1} \qquad \frac{\begin{array}{c}\Gamma \vdash^{wts}_{\Downarrow} \ t : \sigma \\ \Gamma \vdash^{subs} \ \sigma \leq \rho\end{array}}{\Gamma \vdash^{wtr}_{\Downarrow} \ (t \ \texttt{::} \ \sigma) : \rho}\ \textsc{annot2}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{wts}_{\Uparrow} \ u : \sigma \\ \Gamma, x : \sigma \vdash^{wtr}_{\delta} \ t : \rho\end{array}}{\Gamma \vdash^{wtr}_{\delta} \ \texttt{let} \ x \ \texttt{=} \ u \ \texttt{in} \ t : \rho}\ \textsc{let} \qquad \frac{}{\Gamma \vdash^{wtr}_{\delta} \ i : \texttt{Int}}\ \textsc{int}$$

$$\boxed{\vdash^{pat} \ p : \sigma, \Gamma}$$

$$\frac{}{\vdash^{pat}_{\Uparrow} \ \_ : \tau, \{\}}\ \textsc{pwild1} \qquad \frac{}{\vdash^{pat}_{\Uparrow} \ x : \tau, \{x : \tau\}}\ \textsc{pvar1} \qquad \frac{\vdash^{pat}_{\Downarrow} \ p : \sigma, \Gamma}{\vdash^{pat}_{\Uparrow} \ (p \texttt{::} \sigma) : \sigma, \Gamma}\ \textsc{pann1}$$

$$\frac{}{\vdash^{pat}_{\Downarrow} \ \_ : \sigma, \{\}}\ \textsc{pwild2} \qquad \frac{}{\vdash^{pat}_{\Downarrow} \ x : \sigma, \{x : \sigma\}}\ \textsc{pvar2} \qquad \frac{\begin{array}{c}\vdash^{subs} \ \sigma' \leq \sigma \\ \vdash^{pat}_{\Downarrow} \ p : \sigma, \Gamma\end{array}}{\vdash^{pat}_{\Uparrow} \ (p \texttt{::} \sigma) : \sigma', \Gamma}\ \textsc{pann2}$$

$$\boxed{\Gamma \vdash^{wts}_{\delta} \ t : \sigma} \qquad \boxed{\vdash^{inst} \ \sigma \leq \rho}$$

$$\frac{\begin{array}{c}\overline{a} \notin tyvars(\Gamma) \\ \Gamma +\!\!\!+ \, \overline{a} \vdash^{wtr}_{\delta} \ t : \rho\end{array}}{\Gamma \vdash^{wts}_{\delta} \ t : \texttt{forall} \ \overline{a}.\rho}\ \textsc{forall} \qquad \frac{\overline{a} \notin fv(\overline{\tau})}{\vdash^{inst} \ \texttt{forall} \ \overline{a}.\rho \ \leq \ [\overline{a \mapsto \tau}] \, \rho}\ \textsc{inst}$$

Fig. 7: Well-typed simple source terms - annotation propagation version.

a type ρ, and it meets a variable of known type σ. The right thing to do is simply to check that σ is more polymorphic than ρ, which is done by $\vdash^{subs}$. The rules for type annotations, ANNOT1 and ANNOT2, are very similar to the rules for variables.

Lambda abstractions are dealt with by rules ABS1 and ABS2. The inference case (ABS1) is just as before, but the checking case (ABS2) is more interesting. To check that `\x.`t has type $\sigma \rightarrow \rho$, we bind `x` to the polytype σ, *even though* `x` *is is not explicitly annotated*, before checking that the body has type ρ. In this way, we take advantage of contextual information, in a simple and precisely-specified way, to reduce the necessity for type annotations. The rules for annotated abstractions, AABS1 and AABS2, follow easily.

The really interesting rules are the ones for applications. The inference case, APP1, illustates how the two directions can interact. To infer the type for the application (t u) we first *infer* the type $\sigma \rightarrow \rho$ for t, and then *check* the type for u. In this way we take advantage of the function's type (which is often directly extracted from Γ), to provide the type context for the argument. Notice that, unlike Figure 6, there is no use of subsumption or generalisation (in the form of $\vdash^{wts}$), so we have achieved the efficiency gain we sought.

In the checking case for applications, rule APP2 ignores the required type ρ, and uses inference to infer a type ρ' for the application; then it checks that $\rho' \leq \rho$. Although this is a use of subsumption, which we earlier excoriated, it compares two ρ types, which will almost always be monotypes τ. A subsumption test on monotypes amounts to simple equality — or, in an implementation, unification — which is very efficient.

By discarding contextual information, APP2 makes a strategic choice, that in an application we are likely to find better information from the function's type than from the context's type. Other choices are possible, and the framework we have described makes it possible to express them, but we believe this choice is a good one. *[SPJ: Do we want to say more here?]*

7 Inference for higher rank

Having now fixed the type system, we are now ready to develop a type inference algorithm for it. We begin, as before, with the main judgement $\vdash^{wtr}$. At first it seems that we might need two `tcRho` functions, one for each direction:

```
inferRho :: Term -> Tc Rho
checkRho :: Term -> Rho -> Tc ()
```

Doing this would be very burdensome, because when we scale to a real language `tcRho` will have many, many equations. A neat alternative is this:

```
tcRho :: Term -> Expected Rho -> Tc ()

data Expected t = Check t
                | Infer (IORef t)
```

When *checking* that an expression has a particular type `ty` (the $\Downarrow$ direction) we pass `(Check ty)` as the second parameter, in exactly the way that we discussed in Section 5.4. When *inferring* the type of an expression (the $\Uparrow$ direction) we pass `(Infer ref)` as the second parameter, expecting `tcRho` to return the result type by writing to the reference `ref`. This corresponds exactly to the common technique of passing as a parameter the address of the result location – a `var` parameter, in Pascal terminology.

Unlike the reference cells in a `TcTv`, which can be instantiated only to a τ-type, the reference passed in an `Infer` can be filled in by a ρ-type. We will still maintain the previous invariant, that an internal type variable can only be substituted to a τ-type, for the reasons discussed in Section 5.8.

Now we can write `inferRho` and `checkRho` in terms of `tcRho`:

```
checkRho :: Term -> Rho -> Tc ()
checkRho expr ty = tcRho expr (Check ty)

inferRho :: Term -> Tc Rho
inferRho expr = do { ref <- newTcRef (error "inferRho: empty result")
                   ; tcRho expr (Infer ref)
                   ; readTcRef ref }
```

The interesting one is `inferRho`, which creates a new mutable cell, calls `tcRho` (which should write to the cell), and reads the result. The cell is initialised with an error value, so that if `tcRho` erroneously fails to write to the cell any attempt to look at the result will cause the system to halt with a runtime error.

7.1 Basic rules

Now we can look at the definition of `tcRho`. Literals do not change of course. The code for variables changes very slightly:

```
tcRho (Var v) exp_ty
  = do { v_sigma <- lookupVar v
       ; v_rho   <- instTy v_sigma
       ; subsCheckR v_rho exp_ty }    -- Was: unify v_rho exp_ty
```

As before, we look up the variable in the environment, and instantiate its outer for-alls to make a ρ type. But now comes the difference: instead of *unifying* `v_rho` with the expected type `exp_ty`, we call our “super-unifier”, `subsCheck`, to check that `v_rho` is more polymorphic than `exp_ty`. To a first approximation, replacing calls to `unify` with calls to `subsCheck` is the way to move to higher rank!

We do need to modify `subsCheck` slightly, so that it takes an `Expected Rho`, rather than a `Rho` as its second argument, which is why we called it `subsCheckR`. The implementation is easy:

```
subsCheckR :: Rho -> Expected Rho -> Tc ()
subsCheckR t1 (Check t2) = subsCheckRho t1 t2
subsCheckR t1 (Infer r)  = writeTcRef r t1
```

In the checking case, we call `subsCheckRho`, while in the inference case we update the reference with the result type. You might wonder whether we could simplify the `Var` case a little, thus:

```
tcRho (Var v) exp_ty = do { v_sigma <- lookupVar v
                          ; subsCheckR v_sigma exp_ty }
```

After all, `subsCheck` is capable of instanatiating the σ-type obtained by looking up `v`, and this code would correspond more closely to rule VAR2. But `tcRho` guarantees to return a ρ-type, not a σ-type, and our implementation of `subsCheckR` will simply write `v_sigma` into the `r` of an `Infer r` argument, which is wrong. We must instantiate `v_sigma` first.

Next, we deal with applications:

```
tcRho (App fun arg) exp_ty
  = do { fun_ty <- inferRho fun
       ; (arg_ty, res_ty) <- unifyFun fun_ty
       ; checkSigma arg arg_ty        -- Was: checkRho
       ; subsCheckR res_ty exp_ty }   -- Was: unify
```

We infer the type of the function, split its type into its argument and result parts, use `checkSigma` to check that the argument has the right type, and check that the result type of the function is more polymorphic than the expected type. Again, this code looks almost exactly like it did in the Damas-Milner case (Section 5.4), except that we use `checkSigma` instead of `checkRho` for the argument type, and `subsCheckR` instad of `unify` for the result type.

We will discuss `checkSigma` in Section 7.3, but before moving on, we note that `checkSigma` can be used directly in the case for type annotations:

```
tcRho (Ann body ann_ty) exp_ty
   = do { checkSigma body ann_ty
        ; rho_ty <- instTy ann_ty
        ; subsCheckR rho_ty exp_ty }
```

We have to instantiate the `ann_ty` for exactly the same reason as in the `Var` case above. (Indeed, one might be tempted to move this call to `instTy` into `subsCheckR`.)

7.2 Abstractions

The only tricky case is that for abstractions, because here we really do have to treat the inference and checking cases separately:

```
tcRho (Lam pat body) (Check exp_ty)
  = do { (pat_ty, body_ty) <- unifyFun exp_ty
       ; binds <- checkPat pat pat_ty
       ; extendVarEnv binds (checkRho body body_ty) }

tcRho (Lam pat body) (Infer ref)         -- Rule ABS1
```

```
  = do { (pat_ty, binds) <- inferPat pat
         body_ty <- extendVarEnv binds (inferRho body)
       ; writeTcRef ref (pat_ty --> body_ty) }
```

The first case, where we pass in a type, is just as before (Section 5.5). In the inference case, we must first infer the type of the pattern, using `inferPat`, then extend the environment and infer the type of the body, and finally update the incoming reference with the function type `(pat_ty --> body_ty)`.

The functions `checkPat` and `inferPat` are quite straightforward.

```
checkPat :: Pat -> Sigma -> TcM [(Name,Sigma)]
checkPat PWild       exp_ty = return []
checkPat (PVar v)    exp_ty = return [(v, exp_ty)]
checkPat (PAnn p ty) exp_ty = do { subsCheck exp_ty ty
                                 ; checkPat p ty }

inferPat :: Pat -> TcM (Sigma, [(Name,Sigma)])
inferPat PWild    = do { ty <- newTyVar
                       ; return (ty, []) }
inferPat (PVar v) = do { ty <- newTyVar
                       ; return (ty, [(v,ty)]) }
inferPat (PAnn p ty) = do { binds <- checkPat p ty
                          ; return (ty, binds) }
```

There are a couple of points to notice, though. First, `checkPat` is just like the `checkPat` of Section 5.5 except that we use `subsCheck` instead of `unify` as usual. *But notice the back-to-front argument order to* `subsCheck`. If we meet a pattern with a type annotation, we must check that the expected type is more polymorphic than the type annotation, not the other way round! For example, this expression is well typed:

```
(\(f::Int->Int). f 3) :: (forall a. a->a) -> Int
```

Second, notice that `checkPat` takes a `Sigma` (not a `Rho`) as its expected type, and returns a list of `(Name,Sigma)` bindings. (Similarly `inferPat`.) This is essential, because the argument type of a function can be a σ-type, and the type annotations in the pattern can bind the variables to σ-types.

Third, notice that we chose a different strategy to implement the `checkPat`/`inferPat` pair than we did for `checkRho`/`inferRho`. We could have written a single function `tcPat`, taking an `Expected Sigma` argument, as we did for `tcRho`, but instead we chose to duplicate the cases. The single-function version would scale better when we increase the richness of patterns, but duplicating the code here serves nicely to illustrate the alternative strategy, and is a little less abstract.

7.3 Subsumption and generalisation

We need `inferSigma` just as before, to infer the type of the right-hand side of a `let` expression, and its code is unchanged from the Damas-Milner version in

Section 5.6. However, we also need `checkSigma`, which we used when type-checking an application in Section 7.1

```
checkSigma :: Term -> Sigma -> Tc ()
```

One possible implementation is just to infer the most general type, and perform a subsumption check:

```
checkSigma expr exp_ty = do { actual_ty <- inferSigma expr
                            ; subsCheck actual_ty exp_ty }
```

The `subsCheck` function, our "super-unifier", is the heart of the higher-rank type-inference engine. We need to extend the implementation described in Section 5.7 in two ways:

- We must deal with function types (Section 7.3.1).
- We must refine the implementation of skolemisation (Section 7.3.2).

The full code for `subsCheck` can be found in the Appendix.

7.3.1 Functions

At first, dealing with functions seeems simple, just by applying the FUN rule of Figure 5:

```
subsCheckRho (Arrow a1 r1) (Arrow a2 r2)
  = subsFun a1 r1 a2 r2

subsFun a1 r1 a2 r2
  = do { subsCheck a2 a1 ; subsCheck r1 r2 }
```

We have separated out the "business end" of this case into an auxiliary function `subsFun`, because we will need it again shortly.

What if the first argument (say) to `subsCheck` is an internal type variable, but the second is a function type? Then we can use `unifyFun` to persuade the first to look like a function, and then invoke `subsFun` as before. Indeed, this will will work regardless of what the first argument is:

```
subsCheckRho t1 (Arrow a2 r2)
  = do { (a1,r1) <- unifyFun t1; subsFun a1 r1 a2 r2 }
```

This case, plus its symmetric case, actually replaces the `Arrow/Arrow` case we started with.

This approach neatly avoids one hefalump trap. If `t1` happened to be an as-yet-unsubstituted internal type variable, we might be tempted just to unify it to the second argument, *but that might break the invariant that an internal type variable can only be substituted by a monotype.* Instead `unifyFun` will unify the type variable with a freshly-created function type, and `subsFun` will do the rest.

7.3.2 Skolemisation revisited

There is a second, rather more subtle, way in which `subsCheck` needs to be revised. Consider the types of `runST` and `newRef`:

```
runST :: forall a. (forall s. ST s a) -> a
newRef :: forall s a. a -> ST s (Ref s a)
```

(It does not matter exactly what these functions do, but they are described in (Launchbury & PeytonJones, 1995).) Now, is this expression well typed?

```
runST (newRef 'c')
```

Certainly not, because the `(newRef 'c')` has type `ST s (Ref s Char)`; so we would have to instantiate `runST`'s type variable `a` to `(Ref s Char)`, and then the `s` would appear in the result type of `runST`, which it should not do.

Now consider what will happen during inference. First, we will instantiate `runST`'s type with a fresh internal type variable α, giving the type

$$(\forall s\,.\,\mathtt{ST}\ s\ \alpha) \rightarrow \alpha$$

Next, we will call `checkSigma` on the expression `(newRef 'c')`, with expected type $\forall s\,.\,\mathtt{ST}\ s\ \alpha$. In turn, `checkSigma` will infer the most general type for `(newRef 'c')`, namely $\forall t\,.\,\mathtt{ST}\ t\ (\mathtt{Ref}\ t\ \mathtt{Char})$. Now we call `subsCheck`, to check the judgement

$$\forall t\,.\,\mathtt{ST}\ t\ (\mathtt{Ref}\ t\ \mathtt{Char}) \quad \leq \quad \forall s\,.\,\mathtt{ST}\ s\ \alpha$$

As it stands, `subsCheck` will succeed, whereas actually it should fail. It will succeed by first skolemising s, then instantiating t to a fresh type variable β, and then unifying $\mathtt{ST}\ \beta\ (\mathtt{Ref}\ \beta\ \mathtt{Char})$ with $ST\ s\ \alpha$. This unification will succeed, binding β to the skolem constant s, and α to $(\mathtt{Ref}\ s\ \mathtt{Char})$. The trouble is that our skolem constant s has "escaped" into the free type variable α. This difficulty does not show up in the rules for $\vdash^{subs}$; it arises because we are doing type inference, and the internal type variable α has been refined by the subsumption check.

There is another way in which the refinement of an internal type variable can break the curent `subsCheck` code. Consider a call to `subsCheck` to check this judgement:

$$\alpha \quad \leq \quad \forall s\,.\,s \rightarrow s$$

where α is an intenal type variable. Again, we can skolemise s, and then unify, which binds α to the type $s \rightarrow s$. But rule SKOL (Figure 5) requires that the skolem constant not be free in the type on the left of the $\leq$. It wasn't to begin with, but after the unification it may be! So, even though we might make up completely fresh skolem constants in `skolTy`, we must still check the side condition $\overline{a} \notin fv(\sigma)$ in rule SKOL.

Fortunately, the solution to both problems is simple. Here is the code in Section 5.7 that implemented skolemisation:

```
subsCheck sigma1 sigma2        -- Rule SKOL
  = do { (skol_tvs, rho2) <- skolTy sigma2
       ; subsCheck1 sigma1 rho2 }
```

The new thing is that after the subsumption check, *we need to check that the skolemised type variables have not escaped into one of the internal type variables free in* `sigma1` *or* `sigma2`:

```
subsCheck sigma1 sigma2      -- Rule SKOL
  = do { (skol_tvs, rho2) <- skolTy sigma2
       ; subsCheck1 sigma1 rho2
       ; tvs1 <- sigmaTvs sigma1
       ; tvs2 <- sigmaTvs sigma2
       ; let bad_tvs = skol_tvs `intersect` (tvs1 `union` tvs2)
       ; if null bad_tvs then
              return ()
         else
              fail "Type not polymorphic enough" }
```

[SPJ: I think the right way to think about this is to think of the free internal type variables of `sigma2` *as existentially quantified. That does not help me, but it might help others.]* This check is, frankly, distressingly subtle. Its subtlety contradicts our general claim that we can "read off" an algorithm from the typing rules. Nevertheless, it is unavoidable, and it arises in every implementation of subsumption in a type-inference system. Nor is it fundamentally related to higher-rank types. If the Damas-Milner system of Section 4 were enhanced to support open type annotations – i.e. type annotations with free type varibles, bound in some outer scope – then exactly the same problem arises.

7.4 Generalisation revisited

The implementation of `checkSigma` in the previous section is a bit inefficient: `inferSigma` uses `quantify` to generalise the result type, only for `subsCheck` to immediately instantiate it again. Furthermore, since we know the expected type of the term, we would get better error messages from propagating that information inward. Here is a more direct alternative:

```
checkSigma expr exp_ty
  = do { (skol_tvs, rho) <- skolTy exp_ty
       ; checkRho expr rho
       ; env_tvs <- envTvs
       ; tvs2    <- sigmaTvs exp_ty
       ; let bad_tvs = skol_tvs `intersect` (env_tvs `union` tvs2)
       ; if null bad_tvs then
              return ()
         else
              fail "Type not polymorphic enough" }
```

This function amounts to fusing the `quantify` from `inferSigma` with the `instTy` of the `subsCheck`. This time we check whether the skolemised type variables intersect `env_tvs`, the type variables free in the environment, because they are a superset of the free variables of the type that `quantify` would have generated.

7.5 Summary

[SPJ: Summary to be written]

8 Other elaborations

We designed the syntax of our language to be as small as possible while still demonstrating all the interesting effects. In this section we discuss how various extra constructs can readily be added to the core language.

8.1 Multi-branch constructs

Our syntax does not include conditional or `case` expressions. They are easy to add, but they do introduce a small but important wrinkle to the typing rules, and hence the implementation. Suppose the syntax included `if`-expressions:

$$e ::= \ldots \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

Now imagine that we want to infer the type of an `if`-expression. We can infer the type of e_2 and of e_3, but then we need to check that the two types are the same. *But we cannot unify ρ-types,* so we had better infer a τ-type for the two, not a ρ type[8]! The typing rules are therefore:

$$\frac{\Gamma \vdash^{wtr}_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash^{wtr}_{\Uparrow} e_2 : \tau \quad \Gamma \vdash^{wtr}_{\Uparrow} e_3 : \tau}{\Gamma \vdash^{wtr}_{\Uparrow} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \textsc{if}1 \qquad \frac{\Gamma \vdash^{wtr}_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash^{wtr}_{\Downarrow} e_2 : \rho \quad \Gamma \vdash^{wtr}_{\Downarrow} e_3 : \rho}{\Gamma \vdash^{wtr}_{\Downarrow} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho}\ \textsc{if}2$$

In the implementation, how can we guarantee to infer only a monotype for e_2 and e_3 in the inference case? By passing in a fresh internal type variable, thus:

```
tcRho (If e1 e2 e3) exp_ty
  = do { tcCheckRho e1 boolType
       ; exp_ty' <- zapToMonoType exp_ty
       ; tcRho e2 exp_ty'
       ; tcRho e3 exp_ty' }

zapToMonoType :: Expected Rho -> Tc (Expected Rho)
zapToMonoType (Check ty)  = return (Check ty)
zapToMonoType (Infer ref) = do { ty <- newTyVar
                               ; writeTcRef ref ty
                               ; return (Check ty) }
```

This works because we guarantee only to bind an internal type variable to a monotype.

[8] An alternative would be to *infer* a ρ-type for e_2 and *check* that e_3 had this type, or vice versa, but one way round might succeed where the other fails, so it seems unsatisfactory.

8.2 Recursive bindings

Stuff about `letrec`.

8.3 Higher kinds

Stuff about Haskell's higher kinds. Main message: it's entirely orthogonal to higher rank; there are no awkward interactions.

8.4 Translation

GHC does type-directed translation of a term, so the real type of `tcRho` is more like:

```
tcRho :: Term -> Expected Rho -> TcM Term
```

where the returned `Term` has all the type abstractions and applications that are implicit in the source term. I'm very unsure about whether to discuss this:

- It's very nice that it's relatively easy to generate the translation. Once you get the idea, it's pretty easy. So the entire technology developed in this paper scales smoothly to embrace type-directed translation.
- But this is already a long paper. Would this material be too much?
- A sub-question is whehter to introduce the standard type-directed translation judgements, which look like $\Gamma \vdash^{wtr} t \rightsquigarrow t' : \rho$, meaning term t is translated to t'. I suspect the answer is 'yes'.

8.5 Type classes

It's also a feature of the structure of the type checker that it smoothly extends to embrace type classes, including their (non-trivial) type-directed translation. I'm even more unsure about how much to say about this.

9 Summary

[SPJ: Not yet written.]

A Appendix

In the Appendix we give the complete code for the higher-rank type-inference engine.

A.1 Type inference

```
module TcTerm where

import BasicTypes
import TcMonad
import List( (\\) )
import Text.PrettyPrint.HughesPJ

------------------------------------------
--      The top-level wrapper           --
------------------------------------------

typecheck :: Term -> Tc Sigma
typecheck e = do { ty <- inferSigma e
                 ; zonkSigma ty }

------------------------------------------
--      tcRho, and its variants         --
------------------------------------------

checkRho :: Term -> Rho -> Tc ()
checkRho expr ty = tcRho expr (Check ty)

inferRho :: Term -> Tc Rho
inferRho expr
  = do { ref <- newTcRef (error "inferRho: empty result")
       ; tcRho expr (Infer ref)
       ; readTcRef ref }

tcRho :: Term -> Expected Rho -> Tc ()

tcRho (Lit i) exp_ty
  = subsCheckR intType exp_ty

tcRho (Var v) exp_ty
  = do { v_sigma <- lookupVar v
       ; v_rho   <- instTy v_sigma
       ; subsCheckR v_rho exp_ty }

tcRho (App fun arg) exp_ty
  = do { fun_ty <- inferRho fun
       ; (arg_ty, res_ty) <- unifyFun fun_ty
       ; checkSigma arg arg_ty
       ; subsCheckR res_ty exp_ty }

tcRho (Lam pat body) (Check exp_ty)
```

```
  = do { (pat_ty, body_ty) <- unifyFun exp_ty
       ; binds <- checkPat pat pat_ty
       ; extendVarEnv binds (checkRho body body_ty) }

tcRho (Lam pat body) (Infer ref)
  = do { (pat_ty, binds) <- inferPat pat
       ; body_ty <- extendVarEnv binds (inferRho body)
       ; writeTcRef ref (pat_ty --> body_ty) }

tcRho (Let var rhs body) exp_ty
  = do { var_ty <- inferSigma rhs
       ; extendVarEnv [(var,var_ty)] (tcRho body exp_ty) }

tcRho (Ann body ann_ty) exp_ty
   = do { checkSigma body ann_ty
        ; rho_ty <- instTy ann_ty
        ; subsCheckR rho_ty exp_ty }


------------------------------------------
--      tcSigma, and its variants       --
------------------------------------------

inferSigma :: Term -> Tc Sigma
inferSigma e
   = do { exp_ty <- inferRho e
        ; res_tvs <- rhoTvs exp_ty
        ; env_tvs <- envTvs  -- Guaranteed zonked
        ; let forall_tvs = res_tvs \\ env_tvs
        ; quantify forall_tvs exp_ty }

checkSigma :: Term -> Sigma -> Tc ()
checkSigma expr exp_ty
  = do { (skol_tvs, rho) <- skolTy exp_ty
       ; checkRho expr rho
       ; env_tvs <- envTvs
       ; tvs2    <- sigmaTvs exp_ty
       ; let esc_tvs = env_tvs ++ tvs2
             bad_tvs = filter (`elem` esc_tvs) skol_tvs
       ; if null bad_tvs then
           return ()
         else
           fail "Type not polymorphic enough" }

------------------------------------------
--      Typechecking patterns           --
------------------------------------------

checkPat :: Pat -> Sigma -> Tc [(Name,Sigma)]
checkPat PWild       exp_ty = return []
checkPat (PVar v)    exp_ty = return [(v, exp_ty)]
checkPat (PAnn p ty) exp_ty = do { subsCheck exp_ty ty
                                 ; checkPat p ty }
```

```
inferPat :: Pat -> Tc (Sigma, [(Name,Sigma)])
inferPat PWild       = do { ty <- newSigmaTyVar
                          ; return (ty, []) }
inferPat (PVar v)    = do { ty <- newSigmaTyVar
                          ; return (ty, [(v,ty)]) }
inferPat (PAnn p ty) = do { binds <- checkPat p ty
                       ; return (ty, binds) }


------------------------------------------
--      Subsumption checking            --
------------------------------------------

subsCheck :: Sigma -> Sigma -> Tc ()
-- (subsCheck off exp) checks that
--     'off' is more polymorphic than 'exp'

subsCheck off_ty exp_ty         -- Rule SKOL
   = do { (skol_tvs, exp_ty') <- skolTy exp_ty
        ; subsCheck1 off_ty exp_ty'
        ; tvs1 <- sigmaTvs off_ty
        ; tvs2 <- sigmaTvs exp_ty
        ; let esc_tvs = tvs1 ++ tvs2    -- Duplicates don't matter
              bad_tvs = filter (`elem` esc_tvs) skol_tvs
        ; if (null bad_tvs) then
             return ()
          else
             failTc (vcat [text "Subsumption check failed:",
                           nest 2 (ppr off_ty),
                           text "is not as polymorphic as",
                           nest 2 (ppr exp_ty)])
     }

subsCheck1 :: Sigma -> Rho -> Tc ()

subsCheck1 off_ty exp_ty         -- Rule INST
   = do { off_ty' <- instTy off_ty
        ; subsCheckRho off_ty' exp_ty }

subsCheckRho :: Rho -> Rho -> Tc ()
        -- Rule FUN
subsCheckRho t1 (Arrow a2 r2)
  = do { (a1,r1) <- unifyFun t1; subsCheckFun a1 r1 a2 r2 }

subsCheckRho (Arrow a1 r1) t2
  = do { (a2,r2) <- unifyFun t2; subsCheckFun a1 r1 a2 r2 }

subsCheckRho off_ty exp_ty      -- Otherwise revert to
  = unify off_ty exp_ty         -- ordinary unification

subsCheckFun :: Sigma -> Rho -> Sigma -> Rho -> Tc ()
subsCheckFun a1 r1 a2 r2
```

```
  = do { subsCheck a2 a1 ; subsCheckRho r1 r2 }

subsCheckR :: Rho -> Expected Rho -> Tc ()
subsCheckR t1 (Check t2) = subsCheckRho t1 t2
subsCheckR t1 (Infer r)  = writeTcRef r t1
```

A.2 The monad and its operations

```
module TcMonad(
        Tc,      -- The monad type constructor
        failTc, runTc, ErrMsg, lift,

        -- Environment manipulation
        extendVarEnv, lookupVar, envTvs, sigmaTvs, rhoTvs,

        -- Types and unification
        newTyVar, newSigmaTyVar,
         instTy, skolTy, zonkSigma, quantify,
        unify, unifyFun,

        -- Ref cells
        newTcRef, readTcRef, writeTcRef

    ) where

import BasicTypes
import Data.FiniteMap
import Text.PrettyPrint.HughesPJ
import Data.IORef
import Control.Monad( sequence_ )
import List( nub, (\\) )

------------------------------------------
--      The monad itself                --
------------------------------------------

data TcEnv
  = TcEnv { uniqs   :: IORef Uniq,            -- Unique supply
            var_env :: FiniteMap Name Sigma, -- Type environment for term variables
            tv_env  :: [Uniq]                -- In-scope type variables
    }

newtype Tc a = Tc (TcEnv -> IO (Either ErrMsg a))
unTc (Tc a)  = a

type ErrMsg = Doc

instance Monad Tc where
   return x = Tc (\env -> return (Right x))
   fail err = Tc (\env -> return (Left (text err)))
   m >>= k  = Tc (\env -> do { r1 <- unTc m env
                              ; case r1 of
```

```
                                  Left err -> return (Left err)
                                  Right v  -> unTc (k v) env })

failTc :: Doc -> Tc a   -- Fail unconditionally
failTc d = fail (docToString d)

tcError :: Doc -> Tc a  -- Report an error in the type inference engine itself
tcError d = fail (docToString (text "Panic!" <+> d))

runTc :: [(Name,Sigma)] -> Tc a -> IO (Either ErrMsg a)
-- Run type-check, given an initial environment
runTc binds (Tc tc)
  = do { ref <- newIORef 0
       ; let { env = TcEnv { uniqs = ref,
                             var_env = listToFM binds,
                             tv_env = [] } }
       ; tc env }
  where

lift :: IO a -> Tc a
-- Lift a state transformer action into the typechecker monad
-- ignores the environment and always succeeds
lift st = Tc (\env -> do { r <- st; return (Right r) })

newTcRef :: a -> Tc (IORef a)
newTcRef v = lift (newIORef v)

readTcRef :: IORef a -> Tc a
readTcRef r = lift (readIORef r)

writeTcRef :: IORef a -> a -> Tc ()
writeTcRef r v = lift (writeIORef r v)


--------------------------------------------------
--      Dealing with the type environment       --
--------------------------------------------------

extendVarEnv :: [(Name,Sigma)] -> Tc a -> Tc a
extendVarEnv binds (Tc m)
  = Tc (\env -> m (extend env))
  where
    extend env = env { var_env = addListToFM (var_env env) binds }

getEnv :: Tc (FiniteMap Name Sigma)
getEnv = Tc (\ env -> return (Right (var_env env)))

lookupVar :: Name -> Tc Sigma    -- May fail
lookupVar n = do { env <- getEnv
                 ; case lookupFM env n of
                     Just ty -> return ty
                     Nothing -> failTc (text "Not in scope:" <+> quotes (pprName n)) }
```

```
extendTvEnv :: [Uniq] -> Tc a -> Tc a
extendTvEnv tvs (Tc m)
  = Tc (\env -> m (extend env))
  where
    extend env = env { tv_env = tvs ++ tv_env env }

--------------------------------------------------
--      Creating, reading, writing TcTvs        --
--------------------------------------------------

newTyVar :: Tc Rho
newTyVar = do { tv <- newTcTv Flexible; return (TcTv tv) }

newSigmaTyVar :: Tc Sigma
newSigmaTyVar = do { ty <- newTyVar; return (ForAll [] ty) }

newFlexiTyVar, newRigidTyVar :: Name -> Tc TcTv
newFlexiTyVar n = newTcTv Flexible
newRigidTyVar n = newTcTv (Rigid n)

newTcTv :: TvFlavour -> Tc TcTv
newTcTv f = Tc (\ (TcEnv { uniqs = ref }) ->
             do { uniq <- readIORef ref ;
                ; writeIORef ref (uniq + 1)
                ; tref <- newIORef Nothing
                ; return (Right (Tv uniq tref f)) })

readTv  :: TcTv -> Tc (Maybe Tau)
writeTv :: TcTv -> Tau -> Tc ()

readTv  (Tv _ ref _)    = readTcRef ref
writeTv (Tv _ ref _) ty = writeTcRef ref (Just ty)


------------------------------------------
--      Instantiation                   --
------------------------------------------

instTy :: Sigma -> Tc Rho
-- Instantiate the topmost for-alls of the argument type
-- with flexible type variables
instTy (ForAll [] ty) = return ty
instTy (ForAll ns ty) = do { tvs <- mapM newFlexiTyVar ns
                           ; return (substTy ns tvs ty) }

skolTy :: Sigma -> Tc ([TcTv], Rho)
-- Instantiate the topmost for-alls of the argument type
-- with rigid type variables, and return those type variables
skolTy (ForAll [] ty) = return ([], ty)
skolTy (ForAll ns ty) = do { tvs <- mapM newRigidTyVar ns
                           ; return (tvs, substTy ns tvs ty) }

------------------------------------------
```

```
--      Quantification                  --
------------------------------------------

quantify :: [TcTv] -> Rho -> Tc Sigma
-- Quantify over the specified type variables
quantify tvs ty
  = do { mapM_ bind (tvs `zip` new_bndrs) -- 'bind' is just a cunning way
       ; ty' <- zonkRho ty                -- of doing the substitution
       ; return (ForAll new_bndrs ty') }
  where
    used_bndrs = tyVarBndrs ty  -- Avoid quantified type variables in use
    new_bndrs  = take (length tvs) (allBinders \\ used_bndrs)
    bind (tv, name) = writeTv tv (TyVar name)

allBinders :: [Name]    -- a,b,..z, a1, b1,... z1, a2, b2,...
allBinders = [ [x]          | x <- ['a'..'z'] ] ++
             [ (x : show i) | i <- [1..], x <- ['a'..'z']]

------------------------------------------
--      Getting the free tyvars         --
------------------------------------------

envTvs :: Tc [TcTv]
  -- Get the type variables that are mentioned in the environment
  -- This function takes account of zonking, and returns a set (no
  -- duplicates) of type variables that are not substituted
envTvs = do { env <- getEnv; sigmasTvs (eltsFM env) }

sigmaTvs :: Sigma -> Tc [TcTv]
  -- Takes account of the substitution, and returns
  -- a set (no duplicates) of unbound TcTvs
sigmaTvs ty = do { tvs <- get_sigma_tvs ty; return (nub tvs) }

rhoTvs :: Rho -> Tc [TcTv]
rhoTvs ty = do { tvs <- get_rho_tvs ty; return (nub tvs) }

sigmasTvs :: [Sigma] -> Tc [TcTv]      -- Takes account of the substitution
sigmasTvs tys = do { tvs_s <- mapM get_sigma_tvs tys
                   ; return (nub (concat tvs_s)) }

get_sigma_tvs (ForAll ns ty) = get_rho_tvs ty
  -- TcTvs are never quantified

get_rho_tvs (TyVar _)       = return []
get_rho_tvs (Arrow arg res) = do { tvs1 <- get_sigma_tvs arg
                                 ; tvs2 <- get_rho_tvs res
                                 ; return (tvs1 ++ tvs2) }
get_rho_tvs (TyApp tc tys)  = do { tvs_s <- mapM get_rho_tvs tys
                                 ; return (concat tvs_s) }
get_rho_tvs (TcTv tv)       = do { mb_ty <- readTv tv
                                 ; case mb_ty of
                                     Just ty -> get_rho_tvs ty
                                     Nothing -> return [tv] }
```

```
-----------------------------------------
--      Zonking                        --
-----------------------------------------

zonkRho :: Rho -> Tc Rho
-- Eliminate any substitutions in the type
zonkRho (Arrow arg res) = do { arg' <- zonkSigma arg
                             ; res' <- zonkRho res
                             ; return (Arrow arg' res') }
zonkRho (TyApp tc tys)  = do { tys' <- mapM zonkRho tys
                             ; return (TyApp tc tys') }
zonkRho (TyVar n)       = return (TyVar n)
zonkRho (TcTv tv)       -- A mutable type variable
  = do { mb_ty <- readTv tv
       ; case mb_ty of
           Nothing -> return (TcTv tv)
           Just ty -> do { ty' <- zonkRho ty
                         ; writeTv tv ty'       -- "Short out" multiple hops
                         ; return ty' } }

zonkSigma :: Sigma -> Tc Sigma
zonkSigma (ForAll ns ty) = do { ty' <- zonkRho ty
                              ; return (ForAll ns ty') }

-----------------------------------------
--      Unification                    --
-----------------------------------------

unify :: Tau -> Tau -> Tc ()

unify ty1 ty2
  | badType ty1 || badType ty2  -- Compiler error
  = failTc (text "Panic! Unexpected types in unification:" <+>
            vcat [ppr ty1, ppr ty2])

unify (TcTv tv) ty = unifyVar tv ty
unify ty (TcTv tv) = unifyVar tv ty

unify (Arrow (ForAll [] arg1) res1)
      (Arrow (ForAll [] arg2) res2)
  = do { unify arg1 arg2; unify res1 res2 }

unify (TyApp tc1 tys1) (TyApp tc2 tys2)
  | tc1 == tc2 && length tys1 == length tys2
  = sequence_ (zipWith unify tys1 tys2)

unify ty1 ty2 = failTc (text "Cannot unify types:" <+> vcat [ppr ty1, ppr ty2])

-----------------------------------------
unifyVar :: TcTv -> Tau -> Tc ()
unifyVar (Tv u1 _ _) (TcTv (Tv u2 _ _))
```

```
  | u1 == u2            -- Important special case.  If we don't have this
  = return ()           -- we'll get bogus occurs-check errors

unifyVar tv@(Tv uniq ref f) ty2
  = do { mb_ty <- readTcRef ref
       ; case mb_ty of
           Just ty1 -> unify ty1 ty2
           Nothing  -> do { tvs2 <- get_rho_tvs ty2
                          ; if tv `elem` tvs2 then
                                occursCheckErr tv ty2
                            else
                                writeTcRef ref (Just ty2) }
    }


-----------------------------------------
unifyFun :: Rho -> Tc (Sigma, Rho)
--      (arg,res) <- unifyFunTy fun
-- unifies 'fun' with '(arg -> res)'
unifyFun (Arrow arg res) = return (arg,res)
unifyFun fun_ty          = do { arg_ty <- newSigmaTyVar
                              ; res_ty <- newTyVar
                              ; unify fun_ty (arg_ty --> res_ty)
                              ; return (arg_ty, res_ty) }

-----------------------------------------
occursCheckErr :: TcTv -> Tau -> Tc ()
-- Raise an occurs-check error
occursCheckErr tv ty
  = failTc (text "Occurs check for" <+> quotes (ppr tv) <+>
            text "in:" <+> ppr ty)

badType :: Tau -> Bool
-- Tells which types should never be encountered during unification
badType (TyVar _)    = True
badType other        = False
```

A.3 Basic types

```
module BasicTypes where

-- This module defines the basic types used by the type checker
-- Everything defined in here is exported

import Text.PrettyPrint.HughesPJ
import Control.Monad.ST
import Data.IORef
import List( nub, (\\) )
import Maybe( fromMaybe )

infixr 4 -->,--->       -- The arrow type constructor
infixl 4 `App`          -- Application
```

```
-----------------------------------
--      Ubiquitous types         --
-----------------------------------

type Name = String      -- Names are very simple


-----------------------------------
--      Expressions              --
-----------------------------------
                                -- Examples below
data Term = Var Name            -- x
          | Lit Int             -- 3
          | App Term Term       -- f x
          | Lam Pat Term        -- \ x -> x
          | Let Name Term Term  -- let { x = f y } in x+1
          | Ann Term Sigma      -- (f x) :: Int

data Pat = PWild                -- _ (wildcard)
         | PVar Name            -- x
         | PAnn Pat Sigma       -- x :: Int

atomicTerm :: Term -> Bool
atomicTerm (Var v) = True
atomicTerm (Lit i) = True
atomicTerm other   = False


-----------------------------------
--      Types                    --
-----------------------------------

data Sigma = ForAll [Name] Rho

data Rho = TyVar Name         -- Always bound by a ForAll
          | Arrow Sigma Rho    -- Function type
         | TyApp TyCon [Tau]  -- Excludes function types
         | TcTv  TcTv         -- A temporary, typechecker type var

-- A synonym that imply a syntactic invariant on the type
-- Namely, no non-trivial foralls anywhere
type Tau = Rho

data TcTv = Tv Uniq TRef TvFlavour

type TRef = IORef (Maybe Tau)
        -- 'Nothing' means the type variable is not substituted
        -- 'Just ty' means it has been substituted by 'ty'

data TvFlavour = Flexible       -- Can unify with any tau-type

               | Rigid Name     -- Must not unify with anything
```

```
                                -- except a type variable
                                -- The Name just improves error messages

instance Eq TcTv where
  (Tv u1 _ _) == (Tv u2 _ _) = u1 == u2

type Uniq = Int

data TyCon = IntT | BoolT | ListT | TupleT
           deriving( Eq )


---------------------------------
--      Constructors

(-->) :: Sigma -> Rho -> Rho
arg --> res = Arrow arg res

(--->) :: Rho -> Rho -> Rho
arg ---> res = Arrow (ForAll [] arg) res

intType, boolType :: Rho
intType  = TyApp IntT  []
boolType = TyApp BoolT []


---------------------------------
--      Substitution

substTy :: [Name] -> [TcTv] -> Rho -> Rho
-- Replace the specified quantified type variables by
-- given internal type variables
substTy ns tvs ty = subst (ns `zip` map TcTv tvs) ty

subst env (Arrow arg res) = Arrow (subst_sigma env arg) (subst env res)
subst env (TyApp tc tys)  = TyApp tc (map (subst env) tys)
subst env (TyVar n)       = fromMaybe (TyVar n) (lookup n env)
subst env (TcTv tv)       = TcTv tv

subst_sigma env (ForAll ns ty)
  = ForAll ns (subst env' ty)
  where
    env' = [(n,ty) | (n,ty) <- env, not (n `elem` ns)]


---------------------------------
tyVarBndrs :: Rho -> [Name]
-- Get all the binders used in ForAlls in the type, so that
-- when quantifying an outer for-all we can avoid these inner ones
tyVarBndrs ty = nub (bndrs ty)
  where
    bndrs (TyApp _ tys)                 = concatMap bndrs tys
    bndrs (Arrow (ForAll tvs arg) res) = tvs ++ bndrs arg ++ bndrs res
```

```
    bndrs other                               = []


-----------------------------------
--      The expected type       --
-----------------------------------

data Expected a = Infer (IORef a) | Check a


-----------------------------------
--      Pretty printing class   --
-----------------------------------

class Outputable a where
  ppr :: a -> Doc

docToString :: Doc -> String
docToString = render

dcolon = text "::"
dot    = char '.'

-------------- Pretty-printing names ---------------------

pprName :: Name -> Doc
pprName n = text n

-------------- Pretty-printing terms ---------------------

instance Outputable Term where
   ppr (Var n)       = pprName n
   ppr (Lit i)       = int i
   ppr (App e1 e2)   = pprApp (App e1 e2)
   ppr (Lam p e)     = sep [char '\\' <> ppr p <+> text "->", ppr e]
   ppr (Let v rhs b) = sep [text "let {",
                            nest 2 (pprName v <+> equals <+> ppr rhs <+> char '}') ,
                            text "in",
                            ppr b]
   ppr (Ann e ty)    = pprParendTerm e <+> dcolon <+> pprParendSigma ty

instance Show Term where
   show t = docToString (ppr t)

pprParendTerm :: Term -> Doc
pprParendTerm e | atomicTerm e = ppr e
                | otherwise    = parens (ppr e)

pprApp e = go e []
  where
    go (App e1 e2) es = go e1 (e2:es)
    go e es           = pprParendTerm e <+> sep (map pprParendTerm es)
```

```
instance Outputable Pat where
   ppr PWild         = text "_"
   ppr (PVar n)      = pprName n
   ppr (PAnn pat ty) = parens (dcolon <+> ppr ty)

instance Show Pat where
   show t = docToString (ppr t)


-------------- Pretty-printing types ---------------------

instance Outputable Sigma where
   ppr ty = pprSigma topPrec ty

instance Outputable Rho where
   ppr ty = pprType topPrec ty

instance Outputable TcTv where
   ppr (Tv u _ Flexible)  = text "$" <> int u
   ppr (Tv u _ (Rigid n)) = text "$" <> text n <> int u

instance Show Sigma where
   show t = docToString (ppr t)

instance Show Rho where
   show t = docToString (ppr t)

type Precedence = Int
topPrec    = 0  -- Top-level precedence
arrPrec    = 1  -- Precedence of (a->b)
tcPrec     = 2  -- Precedence of (T a b)
atomicPrec = 3  -- Precedence of t

precSigma (ForAll [] rho) = precType rho
precSigma (ForAll ns _)   = topPrec

precType (Arrow _ _) = arrPrec
precType other       = atomicPrec
         -- All the TyApps happen to be atomic

pprParendSigma :: Sigma -> Doc
pprParendSigma ty = pprSigma tcPrec ty


pprSigma :: Precedence -> Sigma -> Doc
-- Print with parens if precedence arg > precedence of type itself
pprSigma p ty | p >= precSigma ty = parens (ppr_sigma ty)
              | otherwise         = ppr_sigma ty

pprType :: Precedence -> Rho -> Doc
-- Print with parens if precedence arg > precedence of type itself
pprType p ty | p >= precType ty = parens (ppr_type ty)
             | otherwise        = ppr_type ty
```

```
ppr_sigma :: Sigma -> Doc         -- No parens
ppr_sigma (ForAll [] ty) = ppr ty
ppr_sigma (ForAll ns ty) = sep [text "forall" <+>
                                  hcat (punctuate comma (map pprName ns)) <> dot,
                               ppr ty]

ppr_type :: Rho -> Doc         -- No parens
ppr_type (Arrow arg res) = sep [pprSigma arrPrec arg <+> text "->",
                                pprType (arrPrec-1) res]
ppr_type (TyApp tc tys) = ppr_tc_app tc tys
ppr_type (TyVar n)      = pprName n
ppr_type (TcTv tv)      = ppr tv

ppr_tc_app IntT  []   = text "Int"
ppr_tc_app BoolT []   = text "Bool"
ppr_tc_app ListT [ty] = brackets (ppr ty)
ppr_tc_app TupleT tys = parens (hsep (punctuate comma (map ppr tys)))
```

References

Damas, Luis, & Milner, Robin. (1982). Principal type-schemes for functional programs. *Pages 207–12 of: Conference record of the 9th annual acm symposium on principles of programming languages.* New York: ACM Press.

Girard, J-Y. (1990). The system F of variable types: fifteen years later. Huet, G (ed), *Logical foundations of functional programming.* Addison-Wesley.

Kfoury, AJ, & Tiuryn, J. (1992). Type reconstruction in finite rank fragments of second-order lambda calculus. *Information and computation*, **98**(2), 228–257.

Laemmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: a practical approach to generic programming. *Pages 26–37 of: Acm sigplan international workshop on types in language design and implementation (tldi'03).* New Orleans: ACM.

Launchbury, J, & Peyton Jones, SL. (1995). State in Haskell. *Lisp and symbolic computation*, **8**(4), 293–342.

Lee, Oukseh, & Yi, Kwangkeun. (1998). Proofs about a folklore let-polymorphic type inference algorithm. *Acm transactions on programming languages and systems*, **20**(4), 707–723.

Odersky, M, & Läufer, K. (1996). Putting type annotations to work. *Pages 54–67 of: 23rd acm symposium on principles of programming languages (popl'96).* St Petersburg Beach, Florida: ACM.

Peyton Jones, Simon. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Pages 47–96 of:* Hoare, CAR, Broy, M, & Steinbrueggen, R (eds), *Engineering theories of software construction, marktoberdorf summer school 2000.* NATO ASI Series. IOS Press.

Pierce, Benjamin C., & Turner, David N. (1998). Local type inference. *Pages 252–265 of: 25th acm symposium on principles of programming languages (popl'98).* San Diego: ACM.

Shields, Mark, & Peyton Jones, Simon. (2002). *Lexically scoped type variables.* Tech. rept. Microsoft Research.