Practical type inference for arbitrary-rank types

5 January 2004

Simon Peyton Jones

Mark Shields

Abstract

Haskell's popularity has driven the need for ever more expressive type system features, most of which threaten the decidability and practicality of Damas-Milner type inference. One such feature is the ability to write functions with higher-rank types — that is, functions that take polymorphic functions as their arguments.

Complete type inference is known to be undecidable for higher-rank type systems, but in practice programmers are more than willing to add type annotations to guide the type inference engine, and to document their code. However, the choice of just what annotations are required, and what changes are required in the type system and its inference algorithm, 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-rank polymorphism through the exploitation of type annotations on λ-bound arguments and arbitrary sub-terms. Though elegant, and more convenient than some 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 Damas-Milner, and of comparable compile-time cost. To substantiate these claims, we describe a type inference algorithm in detail.

Note to the reader.

This is a draft of a paper that we plan to submit to the Journal of Functional Programming. We 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 Introduction

Consider the following Haskell program:

```
foo :: ([Bool], [Char])
foo = let
         f x = (x [True, False], x ['a','b'])
      in
      f reverse

main = print foo
```

In the body of `f`, the function `x` is applied both to a list of booleans and to a list of characters — but that should be fine, because the function passed to `f`, namely `reverse`, works equally well on lists of any type. If executed, therefore, one might think that the program would run without difficulty, to give the result `([False,True], ['b','a'])`.

Nevertheless, the expression is rejected by Haskell's type checker (and would be rejected by ML as well), because Haskell implements the Damas-Milner rule that *a lambda-bound argument (such as* `x`*) can only have a monomorphic type.* The type checker can assign to `x` the type `[Bool]` $\rightarrow$ `[Bool]`, or `[Char]` $\rightarrow$ `[Char]`, but not $\forall a$. `[`a`]` $\rightarrow$ `[`a`]`.

It turns out that one can do a great deal of programming in Haskell or ML without every finding this restriction irksome. For a minority of programs, however, so-called *higher-rank types* turn out to be desirable, a claim we elaborate in Section 2. The following question then arises: is it possible to enhance the Damas-Milner type system to allow higher-rank types, but without making the type system, or its inference algorithm, much more complicated? We believe that the answer is an emphatic "yes".

The main contribution of this paper is to present a practical type system and inference algorithm for arbitrary-rank types; that is, types in which universal quantifiers can occur nested. For example, our system will accept the program:

```
foo :: ([Bool], [Char])
foo = let
         f :: (forall a. [a] -> [a]) -> ([Bool], [Char])
         f x = (x [True, False], x ['a','b'])
      in
      f reverse
```

Notice the type signature for `f`, which expresses the polymorphic type of `f`'s argument. Our work draws together and applies Odersky & Läufer's type system for arbitrary-rank types (Odersky & Läufer, 1996), and Pierce & Turner's idea of local type inference (Pierce & Turner, 1998). The resulting type system, which we describe in Section 4 has the following properties:

- It is a conservative extension of Damas-Milner: any program typeable with Damas-Milner remains typeable.
- The system accommodates types of arbitrary finite rank; it is not, for example, restricted to rank 2. We define the *rank* of a type in Section 3.1.
- Programmer annotations may be required to guide the type inference engine, but the type system specifies precisely which annotations are required, and which are optional.
- The annotations required are quite modest, more so than in the system of Odersky and Läufer.
- The inference algorithm is very little more complicated than the Damas-Milner algorithm.

The main claim of this paper is *simplicity*. In the economics of language design and compiler development, one cannot pay too much to solve a rare problem. Language users only have so much time to spend on learning a language and on understanding compiler error messages. Compiler implementors have a finite time budget to spend on implementing language features, or improving type error messages. There is a real cost to complexity.

We claim, however, that a suitably-designed system of higher-rank types represents an extremely modest addition to a vanilla Damas-Milner type system. First, the language is extended in a simple way: we simply permit the programmer to write explicitly-quantified types, such as the type of `f` above. Second, the implementation changes are also extremely modest. Contrary to our initial intuition, a type-inference engine for Damas-Milner can be modified very straightforwardly to accommodate arbitrary-rank types. This is particularly important in a full-scale compiler like GHC, because the type checker is already extremely complicated. It supports Haskell's overloading mechanism, implicit parameters, functional dependencies, records, scoped type variables, existential types, and more besides. Anything that complicates the main type-inference fabric, on which all this is based, would be hard to justify.

To make this latter claim concrete, we first present a complete implementation of Damas-Milner for a small language (Section 5), and then give all the changes needed to make it work at higher rank (Section 6). The implementation is structured using a monad to carry all the plumbing needed by the type-inference engine, so the code is remarkably concise and is given in full in the Appendix. We hope that, *inter alia*, this implementation of type inference may serve as a convincing example of the utility of monadic programming.

As well as this pedagogical implementation, we have built what we believe is the first full-scale implementation of the Odersky/Läufer idea, in a compiler for Haskell, the Glasgow Haskell Compiler (GHC).

2 Motivation

The introduction showed a rather artificial example in which the argument of a function needed a polymorphic type. Here is another, more realistic, example. Haskell comes with a built-in type class called `Monad`:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

One can easily write monad combinators; for example, `mapM f` applies a monadic function `f` to each element of its argument list[1]:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (x:xs) = f x           >>= \ y ->
                mapM f xs     >>= \ ys ->
                return (y:ys)
```

Now suppose instead that one wanted to do the same thing using an explicit data structure. A value of data type `Monad m` would be a record of two functions, which we write using Haskell's record notation:

```
data Monad m = Mon { return :: a -> m a,
                     bind   :: m a -> (a -> m b) -> m b }
```

We rename `(>>=)` to `bind`, because `bind` is now a selector function that extracts the function from the record. This type declaration would be illegal in Haskell 98, because the type of `return`, for example, mentions a type variable `a` that is not a parameter of the type `Monad`. The idea is, of course, that the data structure contains a *polymorphic* function of type $\forall a \,.\, a \to \mathtt{m}\ a$.

The function `mapM` now takes an explicit argument record of type `Monad m`, from which it extracts the relevant fields by pattern matching[2]:

```
mapM :: Monad m -> (a -> m b) -> [a] -> m [b]
mapM m@(Monad { return = ret, bind = bnd }) f xs
  = case xs of
      []     -> ret []
      (x:xs) -> f x           `bnd`  \y ->
                mapM m f xs   `bnd`  \ys ->
                ret (y:ys)
```

Notice that in this function, `bnd` is used at two different types within a single right-hand side, so it is crucial that it is polymorphic. In this way, we can use a data

[1] In Haskell, lambda abstractions extend as far to the right as possible; in this case, both lambdas extend to the end of the definition.
[2] In Haskell, back-quotes turn a function such as `bnd` into an infix operator.

type whose constructor has a rank-2 type to simulate the effect of type classes — indeed, this is precisely the way in which type classes are implemented internally.

Functions and constructors with higher-rank types now appear quite regularly in the functional programming literature. For example:

Data structure fusion. Short-cut deforestation (Gill *et al.*, 1993) makes use of `build` with type

```
build :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
```

Encapsulation. The encapsulated state monad `ST` (Launchbury & Peyton Jones, 1995) requires a function `runST` with type:

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

The idea is that `runST` ensures that a stateful computation, of type `ST s a`, can be securely encapsulated to give a pure result of type `a`.

Dynamic types. Baars and Swierstra describe the following data type

```
data Equal a b = Equal (forall f . f a -> f b)
```

as a key part of their approach to dynamic typing (Baars & Swierstra, 2002).

Generic programming. Various approaches to generic, or polytypic, programming make essential use of higher-rank types. For example, the "scrap your boilerplate" approach to generic programming (Lämmel & Peyton Jones, 2003) has functions such as:

```
gmapT :: forall a. Data a => (forall b. Data b => b -> b)
                          -> a -> a
```

Hinze's work on generic programming also makes extensive use of higher-rank types (Hinze, 2000).

Invariants. Several authors have explored the idea of using the type system to encode data type invariants, via so-called *nested data types* (Bird & Paterson, 1999; Okasaki, 1999; Hinze, 2001). For example, Paterson and Bird use the following data type to encode lambda terms, in which the nesting depth is reflected in the type:

```
data Term v = Var v | App (Term v) (Term v) | Lam (Term (Incr v))
data Incr v = Zero | Succ v
```

Then the fold over `Term` has type:

```
foldT :: (forall a. a -> n a)
      -> (forall a. n a -> n a -> n a)
      -> (forall a. n (Incr a) -> n a)
      -> Term b -> n b
```

All of these examples use rank-2 types, but rank-3 types are occasionally useful too. Here is an example that defines a `map` function over the `Term` type above, using a fixpoint function `fixMT`:

```
type MapT = forall a b. (a->b) -> Term a -> Term b

fixMT :: (MapT -> MapT) -> MapT
fixMT f = f (fixMT f)

mapT :: MapT
mapT = fixMT (\mt -> \f t ->
              case t of
                Var x     -> Var (f x)
                App t1 t2 -> App (mt f t1) (mt f t2)
                Lam t     -> Lam (mt (mapI f) t))
```

Notice that `fixMT` has a rank-3 type. In order to make the type readable we abbreviate the polymorphic type $\forall a\ b\ .\ (a \to b) \to$ `Term` $a \to$ `Term` b using a type synonym `MapT`. Haskell 98 does not allow polymorphic types as the right hand side of a type synonym, but it is tremendously useful, as this example shows, so GHC permits it.

These cases are not all that common, but there are usually no workarounds; if you need higher-rank types, you *really* need them! Taken together, we believe they make a compelling case that adding higher-rank types adds genuinely-useful expressive power to the language.

3 The key ideas

Motivated by the previous Section, we now present a brief, informal account of our approach to typing higher-ranked programs. The next Section will give a formal description, while Sections 5 and 6 describe the implementation. There is a considerable amount of related work which we allude to only in passing, leaving a more thorough treatment for Section 9.

3.1 Higher-ranked types

We begin with terminology. The *rank* of a type describes the way in which its universal quantifiers are nested. A rank-0 type has no universal quantifiers anywhere; it is a *monotype*, while a function that takes an argument of rank n itself has rank $n + 1$. More precisely, here is the syntax of types, stratified by rank (Kfoury & Tiuryn, 1992):

$$\begin{array}{rrcl} \text{Monotypes} & \tau, \sigma^0 & ::= & a \mid \tau_1 \to \tau_2 \\ \text{Polytypes} & \sigma^{n+1} & ::= & \sigma^n \mid \sigma^n \to \sigma^{n+1} \mid \forall a\ .\ \sigma^{n+1} \end{array}$$

Here are some examples:

$\texttt{Int} \to \texttt{Int}$	Rank 0
$\forall a \,.\, a \to a$	Rank 1
$\texttt{Int} \to (\forall a \,.\, a \to a)$	Rank 1
$(\forall a \,.\, a \to a) \to \texttt{Int}$	Rank 2

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

3.2 Exploiting type annotations

Haskell and ML are both based on the classic Damas-Milner type system (Damas & Milner, 1982), which we review in Section 4.2. This type system has the remarkable property that a compiler can infer the principal type for a polymorphic function, *without any help from the programmer*. Furthermore, the type inference algorithm is not unduly complicated. But Damas-Milner stands on a delicate cusp: almost any extension of the type system either destroys this unaided-type-inference property, or greatly complicates the type-inference algorithm.

The Damas-Milner type system permits $\forall$ quantifiers only at the outermost level of a type scheme, so the examples in Section 2 would all be ill-typed, and it turns out that type inference becomes difficult or intractable if one permits richer, higher-ranked types (Section 9).

An obvious alternative is to abandon the goal of unaided type inference, at least for programs that use higher-ranked types, and instead require the programmer to supply some type annotations to guide type inference, as we did for function `f` in the Introduction. Odersky and Läufer do precisely this, in a paper that is one of the main inspirations of our work (Odersky & Läufer, 1996). Our intuition is that programmers are not only willing to provide explicit type annotations; they are positively eager to do so, as a form of machine-checked documentation, especially as the types become more complicated.

One problem with the Odersky/Läufer approach is that the annotation burden is quite heavy, as we shall see in Section 4.5. Often, though, the context makes a type annotation redundant. For example, consider again our example:

```
f :: (forall a. [a] -> [a]) -> ([Bool], [Char])
f x = (x [True, False], x ['a','b'])
```

The type signature for `f` makes the type of `x` clear, without explicitly annotating the latter. Similarly, in our `Monad` example (Section 2), the local variables `ret` and `bnd` were given polymorphic types somehow inferred from the data type declaration for `Monad`. The idea of propagating type information around the program, to avoid redundant type annotations, is called *local type inference* (Pierce & Turner, 1998).

The original paper used local type inference to stretch the type system in the direction of sub-typing, but we apply the same technique to support higher-rank types, as we shall see in Section 4.8.

3.3 Subsumption

Suppose that we have variables bound with the following types:

$$\begin{array}{lll} \texttt{k} & :: & \forall ab \,.\, a \to b \to b \\ \texttt{f1} & :: & (\texttt{Int} \to \texttt{Int} \to \texttt{Int}) \to \texttt{Int} \\ \texttt{f2} & :: & (\forall x \,.\, x \to x \to x) \to \texttt{Int} \end{array}$$

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`.

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 accepted. Why? Because `k` is more polymorphic than the function `f2` requires. The former is independently polymorphic in `a` and `b`, while the latter is less flexible.

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 Läufer use the term *subsumption* for this "more polymorphic than" relation. When extended to arbitrary rank, the usual co/contra-variance phenomenon occurs; that is, $\sigma_1 \to \texttt{Int}$ is more polymorphic than $\sigma_2 \to \texttt{Int}$ if σ_1 is *less* polymorphic than σ_2. For example, consider

$$\begin{array}{lll} \texttt{g} & :: & ((\forall x \,.\, [x] \to [x]) \to \texttt{Int}) \to \texttt{Int} \\ \texttt{k1} & :: & (\forall a \,.\, a \to a) \to \texttt{Int} \\ \texttt{k2} & :: & ([\texttt{Int}] \to [\texttt{Int}]) \to \texttt{Int} \end{array}$$

Since $(\forall a \,.\, a \to a)$ is more polymorphic than $(\forall x \,.\, [x] \to [x])$, it follows that

$$((\forall a \,.\, a \to a) \to \texttt{Int})$$

is *less* polymorphic than

$$((\forall x \,.\, [x] \to [x]) \to \texttt{Int})$$

and hence the application `(g k1)` is ill-typed. In effect, `k1` requires to be given an argument of type $(\forall a \,.\, a \to a)$, whereas `g` only promises to pass it a (less polymorphic) argument of type $(\forall x \,.\, [x] \to [x])$. On the other hand, the application `g k2` is well typed.

3.4 Predicativity

Once one allows polytypes nested inside function types, it is natural to ask whether one can also call a polymorphic function at a polytype. For example, consider the following two functions:

```
revapp :: a -> (a->b) -> b
revapp x f = f x

poly :: (forall v. v -> v) -> (Int, Bool)
poly f = (f 3, f True)
```

Would the application `(revapp (\x->x) poly)` be legal? The application would require us to instantiate the type variable a from `revapp`'s type with the polytype $\forall v \,.\, v \to v$. The function `fixMT` in Section 2 is a more practical example. It is a specialised instance of an "ordinary" `fix` function:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

However, using `fix` in place of `fixMT` would mean instantiating `fix` at the polymorphic type `MapT`. The same issue arises in the context of data structures. Suppose we have a data type:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

It is legal to have the type (`Tree` $(\forall a \,.\, a \to a)$); that is, a `Tree` whose leaves hold polymorphic functions? Doing so would require us to instantiate the `Leaf` constructor at a polymorphic type.

A type system that allows a polymorphic function to be instantiated at a polytype is called *impredicative*, while a *predicative* system only allow a polymorphic function to be instantiated with a monotype.

The Damas-Milner type system is predicative, of course, and so is the Odersky/Läufer system. Type inference is much easier in a predicative type system, as we discuss in Section 5.7, so we adopt predicativity in our type system too. Remarkably, it is possible to support both type inference and impredicativity, as MLF shows (Le Botlan & Rémy, 2003), but doing so adds significant new complications to both the type system and the implementation — see Section Section 9.2.

3.5 Higher-kinded types

Haskell allows abstraction over *higher-kinded types*, as we have already seen. For example, our `Monad` type was defined like this:

```
data Monad m = Mon { return :: a -> m a,
                     bind   :: m a -> (a -> m b) -> m b }
```

Here, the type variable `m` ranges over type *constructors*, such as `Maybe` or `Tree`, rather than over *types*. A Haskell compiler will infer that `m` has kind $* \to *$; that is, `m` maps types (written "$*$") to types.

The question of type inference for higher kinds is an interesting one. Happily, it turns out that the solution adopted by Haskell for higher kinds extends smoothly

	Rank 1	Arbitrary rank
Not syntax-directed Does not lead to an algorithm	Damas-Milner $\rho ::= \tau$ Figure 3	Odersky-Läufer $\rho ::= \tau \mid \sigma \to \sigma$ Figure 5
Syntax-directed Algorithm can be read off	Damas-Milner $\rho ::= \tau$ Figure 4	Our system $\rho ::= \tau \mid \sigma \to \rho$ Figures 6 and 7

Type contexts $\Gamma ::= \Gamma, x : \sigma \mid \epsilon$

Polytypes $\sigma ::= \forall \overline{a}.\rho$
Monotypes $\tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid a$
Type variables a, b

Fig. 1: Road map

to work in the presence of higher-rank types, as we know from our experience of implementing both in GHC. The two features are almost entirely orthogonal. We therefore do not discuss higher kinds at all in the rest of the paper.

3.6 Summary

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

4 Type systems for higher-rank types

In this section we give a precise specification of the type system we sketched informally in Section 3. In fact, we will discuss five type systems in all, using the road-map shown in Figure 1.

The first column, headed "Rank 1" deals with the conventional rank-1 ML-style type system. There are two standard presentations of this type system, which correspond to the two cells of this column. Type systems are often specified initially in a *non-syntax-directed* style. This style is terse, and well-adapted for proving properties, but does not usually suggest a type inference algorithm. A standard idea is to recast the rules in *syntax-directed* form, so that the structure of the typing derivation is determined by the syntactic structure of the program. With a bit of practice, it is usually possible to "read off" an inference algorithm from a set of typing rules in syntax-directed form. We present the textbook Damas-Milner system in both forms (left-hand column of the table in Figure 1), to introduce in a familiar context our language, and to review the idea of syntax-directed rules.

Then we will follow exactly the same development for the arbitrary-rank system

Term variables	x, y, z			
Integers	i			
Terms	t, u	::=	i	Literal
		|	x	Variable
		|	$\backslash x \,.\, t$	Abstraction
		|	$\backslash (x :: \sigma) \,.\, t$	Typed abstraction (σ closed)
		|	$t\ u$	Application
		|	`let` x `=` u `in` t	Local binding
		|	$t :: \sigma$	Type annotation (σ closed)

Fig. 2: Syntax of simplified source types and terms

(right-hand column of the table). The top right-hand corner is a non-syntax-directed system, developed by Odersky and Läufer, on which our work is based. From this we derive a syntax-directed system in the bottom right-hand corner — or rather two of them, for reasons that will become apparent.

These systems differ in their type structure. They all share a common definition for polytypes (σ) and monotypes (τ), also given in Figure 1. In this Figure, and elsewhere, we use the notation $\overline{a}$ to mean a sequence of zero or more type variables $a_1, \ldots, a_n$. The systems differ in their definition of the intermediate *rho-types* (ρ), whose distinguishing feature is that they have no top-level quantifiers. The syntax of rho-types is given, for each system, in Figure 1.

That will then leave us ready to develop an implementation in Sections 5 and 6.

4.1 Notation

We will present all our type systems for a simple language, given in Figure 2. The language of terms is very simple: it is the lambda calculus augmented with non-recursive `let` bindings, and type annotations on both terms and lambda abstractions. 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 abstractions, 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* — that is, they have no free type variables — which is the case in Haskell 98. There are strong reasons to want open type annotations, which require lexically-scoped type variables (Shields & Peyton Jones, 2002), but we will avoid that complication in this paper.

Figure 1 defines the syntax of types. A minor point is that in the definition of polytypes we quantify over a vector of zero or more type variables, $\overline{a}$, rather than quantifying one variable at a time with a recursive definition. These quantifiers are

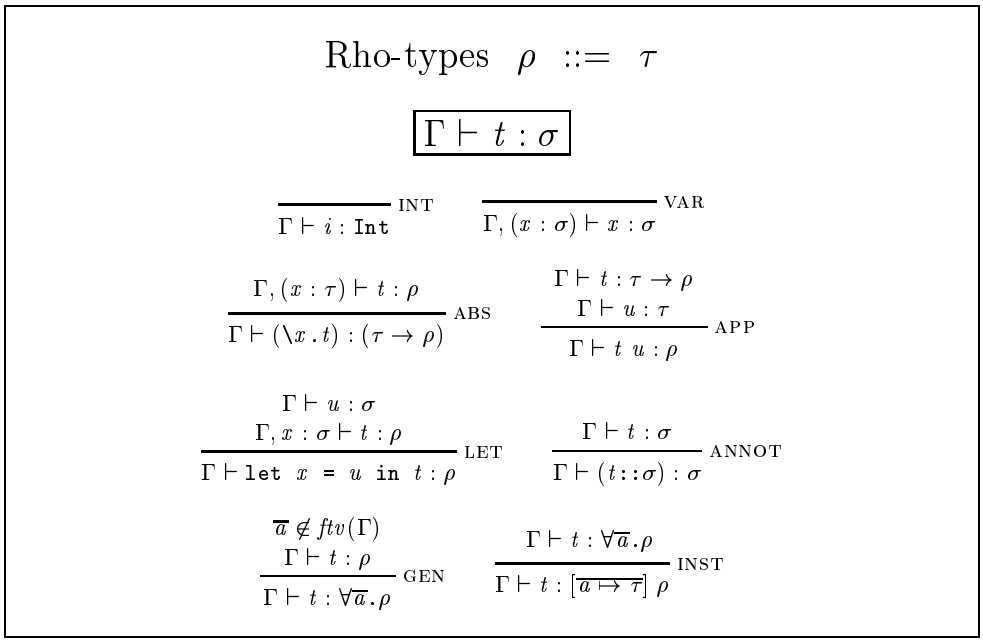

Fig. 3: The non-syntax-directed Damas-Milner type system

not required to bind all the free type variables of ρ; that is, a polytype σ can have free type variables. Otherwise it would not be possible to write higher-rank types, such as $\forall a \,.\, (\forall b \,.\, (a, b) \to (b, a)) \to [a] \to [a]$. (Here, and in subsequent examples, we assume we have list and pair types, written $[\tau]$ and (τ_1, τ_2) respectively; but we will not introduce any terms with these types.) In our syntax, a σ-type always has a $\forall$, even if there are no bound variables, but we will sometimes abbreviate the degenerate case $\forall \,.\, \rho$ as simply ρ.

The same Figure also shows type contexts, Γ, which convey the typings of in-scope variables; Γ binds a term variable, x, to its type σ.

We define $ftv(\sigma)$ to be the free type variables of σ, and extend the function to type contexts in the obvious way: $ftv(\Gamma) = \bigcup\{ftv(\sigma) \mid (x : \sigma) \in \Gamma\}$. We use the notation $[\overline{a \mapsto \tau}]\rho$ to mean the capture-avoiding substitution of type variables $\overline{a}$ by monotypes $\overline{\tau}$ in the type ρ.

4.2 The non-syntax-directed Damas-Milner system

Figure 3 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 $\forall \overline{a} \,.\, \tau$. The main judgement takes the form:

$$\Gamma \vdash t : \sigma$$

which means "in environment Γ the term t has type σ". The alert reader will nevertheless notice several judgements of the form $\Gamma \vdash t : \rho$, for example in the conclusion of rule APP. As mentioned in Section 4.1, this is just shorthand for the σ-type $\forall\ .\ \rho$, namely a type with no quantifiers.

Rule INST quietly makes a very important point: the system is predicative (Section 3.4), so *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. Efficient type inference depends crucially on this restriction, a point that we amplify in Section 5.7.

4.3 The syntax-directed Damas-Milner system

Each rule in Figure 3 has a distinct syntactic form in its conclusion, except for two: GEN (generalisation) and INST (instantiation). Because these two have the same syntactic form in their premise as in their conclusion, on can apply them pretty much anywhere; for example, one could alternate GEN and INST indefinitely. This flexibility makes it hard to turn the rules into a type-inference *algorithm*. For example, given a term, say `\x.x`, it is not clear which rules to use, in which order, to derive a judgement $\vdash$ `\x.x` $: \sigma$ for some σ.

If *all* the rules had a distinct syntactic form in their conclusions, the rules would be in so-called *syntax-directed form*, and that would, in turn, fully determine the shape of the derivation tree for any particular term t. This is a very desirable state of affairs, because it means that the steps of a type inference algorithm can be driven by the syntax of the term, rather than having to search for a valid typing derivation.

Figure 4 shows an alternative form of the typing rules that *is* syntax-directed. The main judgement now takes the form:

$$\Gamma \vdash t : \rho$$

meaning that "in context Γ term t has type ρ". In contrast to Figure 3, the type ρ in the judgement is a *monotype* (recall that in the Damas-Milner system, ρ is the same as τ).

The places where type generalisation and instantiation take place are now completely specified by the syntax of the term. Instantiation is handled by the auxiliary judgement $\vdash^{inst}$, where

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

means that the outer quantifiers of σ can be instantiated to give ρ. Instantiation is used in rule VAR to instantiate the type of a polymorphic variable at its occurrence sites.

Dually, generalisation is handled by the auxiliary judgement $\vdash^{gen}$ which infers a polytype for a term. It is used in rule LET to type the right-hand side of the `let`.

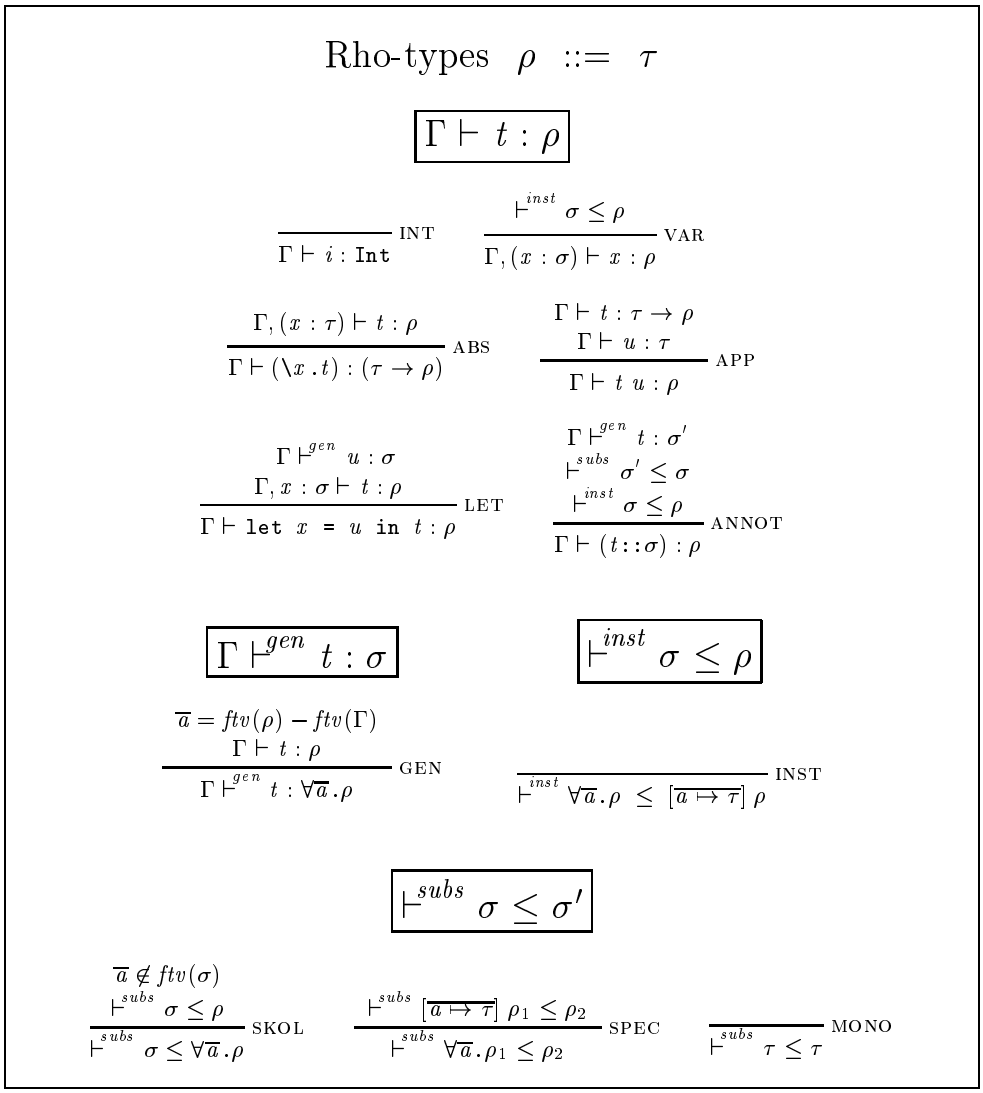

Fig. 4: The syntax-directed Damas-Milner type system

In the spirit of moving towards an algorithm, GEN also specifies that the quantified type variables $\overline{a}$ should be exactly the variables that are free in ρ but not in Γ (contrast rule GEN in Figure 3). There is no point in generalising over a variable that is not free in ρ; but otherwise it is useful to generalise as much possible, subject to $\overline{a} \notin \Gamma$. In this way, we have constrained the valid derivations still further — that is, moved closer to a deterministic algorithm — without reducing the set of typeable terms.

We can now very nearly regard the rules as an algorithm. Corresponding to the judgement $\vdash$ is an inference algorithm that, given a context Γ and a term t computes

a type τ such that $\Gamma \vdash t : \tau$; and similarly for the other judgements[3]. However, the rules still leave one big thing unspecified: in various rules an otherwise-unspecified τ appears out of nowhere. For example, in rule ABS, where does the τ come from? Given the empty context and the term `\x.x`, the following judgements all hold, by choosing the τ in rule ABS to be `Int`, $[a]$ and a respectively:

$$\begin{array}{l} \vdash \texttt{(\x.x)} : \texttt{Int} \to \texttt{Int} \\ \vdash \texttt{(\x.x)} : [a] \to [a] \\ \vdash \texttt{(\x.x)} : a \to a \end{array}$$

Of course, we want the last of these, because it is the *most general type* for `\x.x`, the one that is better than all the others, and in Section 5 we will see how to achieve this. A similar guess must be made in rule INST where we have choose the types $\overline{\tau}$ to use when instantiating σ.

The trade-off between syntax-directed and non-syntax-directed formulations of typing judgements is well known. However, although it is well known, it is not well documented; Clement *et al* (1986) is one of the few papers that discuss the matter, and has the merit of giving a proof of equivalence of the two systems.

4.4 Type annotations and subsumption

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:

$$\texttt{(\x.x)} \texttt{ :: } (\forall a \,.\, [a] \to [a])$$

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:

$$\texttt{((\x.x)} \texttt{ :: } (\forall a \,.\, [a] \to [a])) \texttt{ (True,False)}$$

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 involve 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.

In the non-syntax-directed system of Figure 3, type annotations are easy to handle.

[3] This is not the only possible way to regard the typing rules as an algorithm, as we discuss in Section 9.

Rule ANNOT simply requires that a type-annotated term $(t :: \sigma)$ does indeed have type σ.

Matters become more interesting in the syntax-directed system of Figure 4. There, rule ANNOT type-checks a type-annotated term in three stages:

- Find t's most general type σ', using $\vdash^{gen}$;
- This type might differ from the programmer-supplied annotation σ, because the latter is not necessarily the most general type of t. So the next step is to check that σ' is at least as polymorphic as σ, using a new judgement form $\vdash^{subs}$, shown in Figure 4;
- Finally, 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}". The judgement embodies precisely the subsumption relationship of Section 3.3. Unlike $\vdash^{inst}$, it compares two *polytypes*. For example:

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

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 in the right-hand type, as the last three examples demonstrate.

It is worth studying carefully the rules for $\vdash^{subs}$, in Figure 4, 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. The side condition in SKOL, $\overline{a} \notin \mathit{ftv}(\overline{\tau})$, ensures that the universally-quantified variables $\overline{a}$ are indeed not involved in σ.

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

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

first use SKOL to skolemise b and c, and *then* use SPEC to instantiate a with the type (β, γ). The derivation looks like this:

$$\dfrac{\dfrac{\dfrac{}{(\beta,\gamma) \to (\beta,\gamma) \quad \leq \quad (\beta,\gamma) \to (\beta,\gamma)}\,\text{MONO}}{\forall a \,.\, a \to a \quad \leq \quad (\beta,\gamma) \to (\beta,\gamma)}\,\text{INST}\ [a \mapsto (\beta,\gamma)]}{\forall a \,.\, a \to a \quad \leq \quad \forall bc \,.\, (b,c) \to (b,c)}\,\text{SKOL}$$

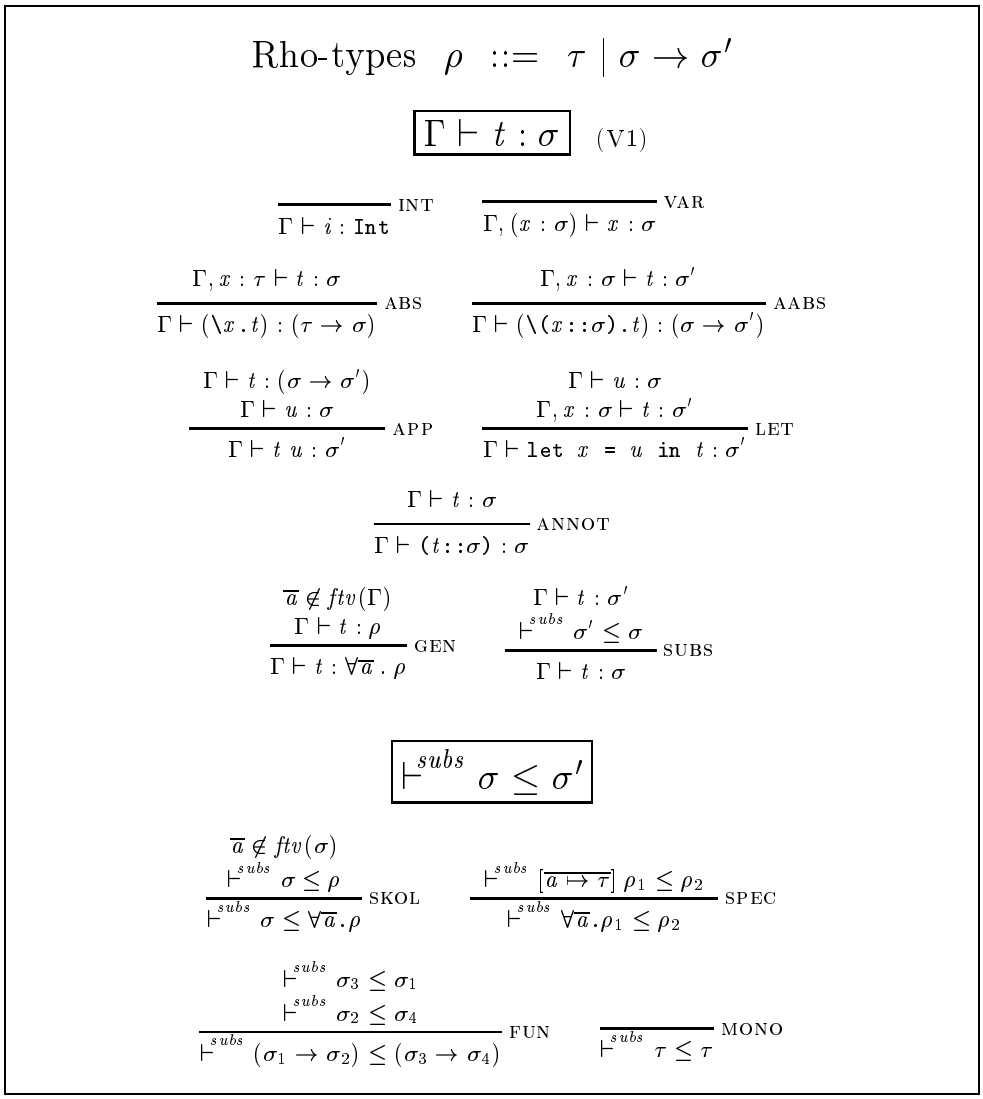

Fig. 5: The Odersky-Läufer type system

Strictly speaking there are three judgement forms: $\vdash^{subs} \sigma \le \sigma$, $\vdash^{subs} \sigma \le \rho$, and $\vdash^{subs} \rho \le \rho$, in which each makes use of the next in its premise. Furthermore, since ρ and τ are the same in the Damas-Milner system, $\vdash^{inst} \sigma \le \rho$ coincides with $\vdash^{subs} \sigma \le \rho$, but we prefer to keep them separate because they will not coincide in subsequent variants.

4.5 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 Läufer (1996).

Figure 5 presents the Odersky/Läufer type checking rules for our term language, in non-syntax-directed form.

The Figure begins by defining rho-types, ρ, to complete the syntax of types:

$$\text{Rho-types} \quad \rho \quad ::= \quad \tau \mid \sigma_1 \to \sigma_2$$

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.

Comparing these rules to those of the non-syntax-directed Damas-Milner system in Figure 3, the two significant differences are these:

- The syntax of terms is extended with a new form of lambda abstraction, $\backslash(x :: \sigma) \,.\, t$, in which the bound variable is explicitly annotated with a polytype, σ. The argument type of such an abstraction is σ (rule AABS) in contrast to an ordinary, unannotated lambda abstraction whose argument type is a mere monotype, τ (rule ABS).
- Rule GEN is unchanged, but instantiation (rule INST) is replaced by subsumption (rule SUBS). The idea is that if we know $(t : \sigma)$, then we also know $(t : \sigma')$ for any σ' that is less polymorphic than σ. Checking the "more polymorphic than" condition is done by the type-subsumption judgement, $\vdash^{subs}$, shown in Figure 5.

The definition of subsumption in Figure 5 is just like that in Figure 4, 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:

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

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

4.6 Avoiding Generalisation

The typing rules of Figure 5 have the same difficulty as those of the non-syntax-directed rules for Damas-Milner: they are not syntax-directed, and are far removed from an algorithm. In particular, rule GEN 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:

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

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:

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

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

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

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

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

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:

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

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

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

The straightforward way to turn the rules of Section 4.5 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 Läufer give just such an inference algorithm in their paper (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[4] coercion terms.

4.7 Prenex-form polytypes

One might be prepared to accept this high cost in return for a more expressive type system, but such need not be the case. Consider two functions

$$\begin{array}{lcl} \texttt{h1} & :: & \forall a \,.\, a \to (\forall b \,.\, b \to (a, b)) \\ \texttt{h2} & :: & \forall ab \,.\, a \to b \to (a, b) \end{array}$$

In the absence of type annotations, *no program context can distinguish* `h1` *from* `h2`. One might wonder if the partial application `(h1 True)` might be polymorphic, while `(h2 True)` is not. But we could only tell if that the latter is monomorphic by attempting to apply it twice, to terms of different types. But that implies that

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

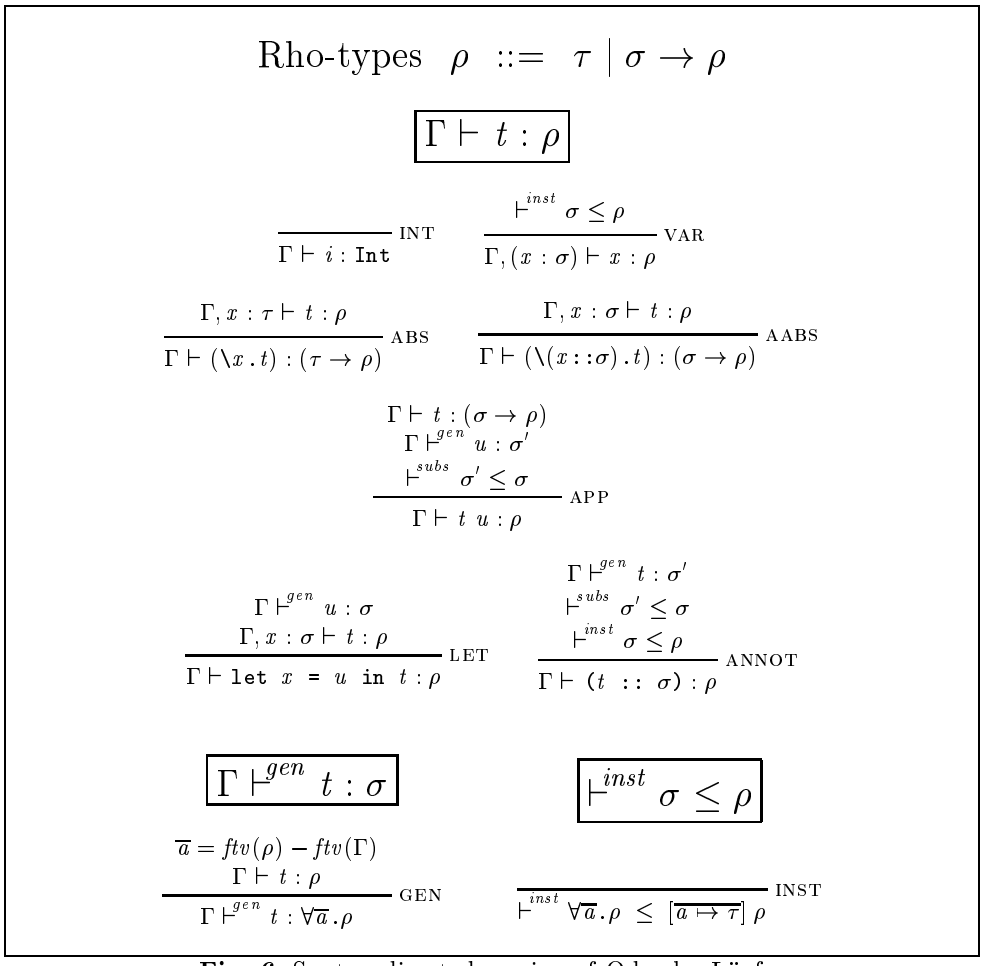

Fig. 6: Syntax-directed version of Odersky-Läufer

the partial application is let-bound (or passed to a function with a polymorphic argument type), in which case the partial application will have been re-generalized. In short, `h1` and `h2` are indistinguishable, so there is no reason to prefer the former over the latter.

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

$$\text{Rho-types} \quad \rho \ ::= \ \tau \mid \sigma \to \rho$$

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 beginning. 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 (1992). Their goal was to precisely characterise the decidability 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 ρ-type, 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.

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 4. The main judgement form $\vdash$ checks whether a term has a type ρ which has no top-level $\forall$ quantifiers, just like Figure 4. Another similarity is the separate judgements $\vdash^{inst}$ and $\vdash^{gen}$ which specify exactly where specialisation and generalisation take place, just like the Damas-Milner system. Notice that $\vdash^{inst}$ strips off, and $\vdash^{gen}$ glues on, only *top-level* quantifiers. In contrast, the (unchanged) judgement $\vdash^{subs}$ deals with nested quantification.

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

4.8 Propagating types inwards

The revised rules are now syntax-directed, but they share with the original Odersky/Läufer system the property that the type of a lambda abstraction can only have a higher-rank type (i.e. polytype on the left of the arrow) if the lambda-bound variable is explicitly annotated; compare rules ABS and AABS in Figure 6.

Often, though, this seems far too heavyweight. For example, suppose we have the definition:

```
foo = (\i. (i 3, i True)) :: (∀a . a → a) → (Int, Bool)
```

(In Haskell one would instead use a separate type signature:

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

but we use the one-line version to avoid adding declaration type signatures to our little language.) In this example it is plain as a pike-staff that `i` should have the type $(\forall a\ .\ a \to a)$, even though it is not explicitly annotated as such. Somehow we would like to "push the type annotation inwards", so that the type signature for `foo` can be exploited to give the type for `i`. The idea of taking advantage of type annotations in this way is not new (Pierce & Turner, 1998): it is called partial type inference, as we discuss in Section 9.3, and we use the Pierce/Turner formalism in what follows.

With this in mind, Figure 7 gives typing rules that express the idea of propagating types inwards. The Figure describes two very similar typing judgements.

$$\Gamma \vdash_{\Uparrow} t : \rho$$

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

$$\Gamma \vdash_{\Downarrow} t : \rho$$

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 judgements $\vdash^{gen}$ and $\vdash^{inst}$ are generalised in the same way.

Many of the rules are "polymorphic" in the direction δ. For example, the rules INT, VAR, ANNOT, and LET 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 even more such constructs (`case` and `if` are other examples), so the notational saving is quite worth while.

The other rules each split into two. For example, applications are dealt with by APP1 and APP2. The inference case, APP1, illustrates how the two directions can interact. To infer the type for the application $(t\ u)$ we first *infer* the type $\sigma \to \rho$ 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.

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 $\rho' \leq \rho$. There is clearly some information loss here. We know the result type, ρ, for the function term t, but we do not know its argument type. The rules provide no way to express this *partial* information about the t's type – but see the discussion in Section 9.3.

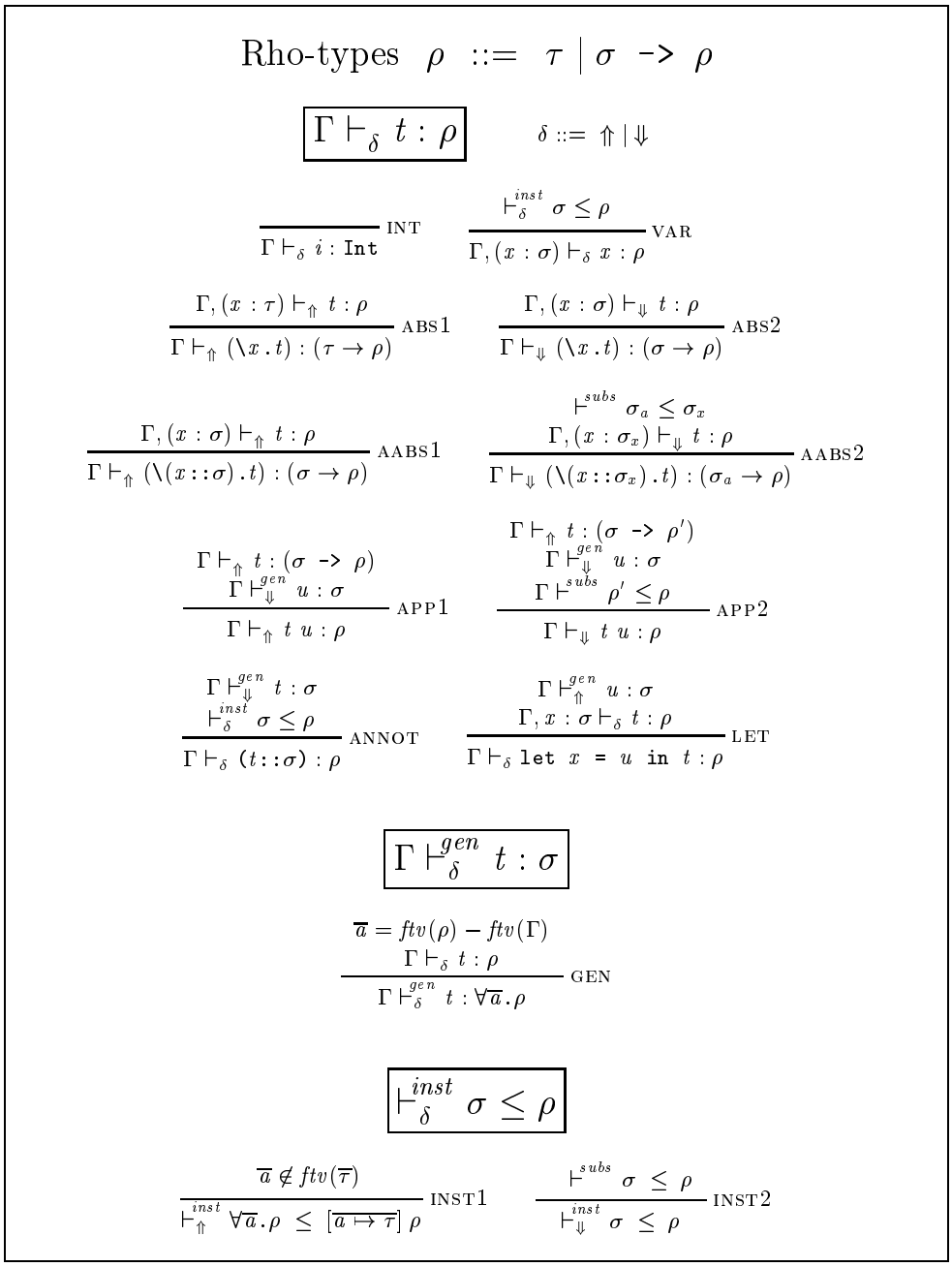

Fig. 7: Bidirectional version of Odersky-Läufer

Dually, ordinary (un-annotated) 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 $\backslash x\,.\,t$ has type $\sigma \to \rho$, we bind x to the polytype σ, *even though* `x` *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.

We also need two rules for annotated lambda abstractions. In the inference case, AABS1, we extend the environment with the σ-type specified by the annotation, and infer the type of the body. In the checking case, AABS2, we extend the environment in the same way, before checking that the body has the specified type — but we must also check that the argument type expected by the environment σ_a is more polymorphic than that specified in the type annotation σ_x. Notice the contravariance! For example, this expression is well typed:

$$\texttt{(\textbackslash(f::Int->Int). f 3)} \;::\; (\forall a\ .\ a \to a) \to \texttt{Int}$$

Finally the $\vdash^{inst}_{\delta}$ judgement form also splits into two. Rule INST1 deals with the inference case for $\vdash^{inst}_{\delta}$: we just instantiate the outer $\forall$ quantifiers just like the old INST rule. The other case, INST2, is more interesting. Here, we are pushing inward 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 $\vdash^{subs}$. Rule ANNOT and VAR both make use of the $\vdash^{inst}_{\delta}$, just as they did in the Damas-Milner case (Figure 4), but ANNOT becomes slightly simpler. In our Damas-Milner rules we inferred the most general type for t, and performed a subsumption check against the specified type; now we can simply push the specified type inwards, into t.

In summary, these type rules reduce the burden of type annotations by propagating type information inwards. As we shall see when we come to implementation in Section 5.4, the idea of propagating types inwards is desirable for reasons quite independent of higher-rank types, so the impact on implementation turns out to be rather modest.

4.9 Properties of the various type systems

What is the relationship between the uni-directional rules of Section 4.7 and the bi-directional rules of Section 4.8? Firstly, if a term is typeable by the former, then it is typeable by the latter in inference mode:

$$\Gamma \vdash e : \rho \quad \Rightarrow \quad \Gamma \vdash_{\Uparrow} e : \rho$$

The reverse is not true: the bidirectional rules, even in inference mode, may type more programs. For example, this term is only typeable using the bidirectional rules:

```
(\x -> (x True, x 'a')) :: (forall a. a->a) -> (Bool, Char)
```

Furthermore, if the bidirectional rules succeed in inference mode, then that type is principal, in the following sense:

$$\text{If } \Gamma \vdash_{\Uparrow} e : \rho \text{ and } \vdash^{subs} \rho \leq \rho' \text{then } \begin{array}{ll} (1) & \Gamma \vdash_{\Uparrow} e : \rho' \\ (2) & \Gamma \vdash_{\Downarrow} e : \rho' \end{array}$$

$$\text{If } \Gamma \vdash_{\Uparrow} e : \rho \text{ and } \vdash^{subs} \rho \leq \rho' \quad \text{then} \quad \Gamma \vdash_{\Downarrow} e : \rho'$$

However, a term might of course be typeable in checking mode when it is not typeable in inference mode; for example `(\x -> (x True, x 'a'))`.

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 4 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 type `Term` in Figure 8 is the representation for terms, whose syntax was given in Figure 2. The data types `Sigma` and `Rho` deserve a little more explanation. Here are their definitions, copied from Figure 8:

```
data Sigma = ForAll [Name] Rho

data Rho   = TyVar Name          -- Always bound by a forall
           | TcTv  TcTv          -- Internal type variable
           | Arrow Tau Tau       -- Functions
           | IntTy                   -- Integers

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 `Sigma` ever has a free quantified type variable.

```
---------------- Terms ------------------
data Term = Var Name            -- x
          | Lit Int             -- 3
          | App Term Term       -- f x
          | Lam Name Term       -- \x. x
          | Let Name Term Term  -- let x = f y in x+1
          | Ann Term Sigma      -- f x :: Int

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

data Rho   = TyVar Name         -- Always bound by a forall
           | TcTv  TcTv         -- Internal type variable
           | Arrow Tau Tau      -- Functions
           | IntTy              -- Integers

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

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

intType :: Tau
intType = IntTy

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

Fig. 8: The `Term` and `Type` data types

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 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 5.7. 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 (occasionally) Greek letters, α, β, to denote internal type variables.

In fact, much of the type inference engine is independent of the representation of types. The infix function `(-->)` helps to maintain this abstraction by allowing the inference engine to construct a function type using `(-->)` without knowing how function types are represented internally. Similarly `intType` is the `Tau` representing the type `Int`.

5.2 The type-checker monad

The type constructor `Tc` is the type-checker monad, whose primitive operations are given in Figure 9. The monad serves the following roles:

```
        -- Control flow
check :: Bool -> String -> Tc ()   -- 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
instantiate :: Sigma -> Tc Rho          -- Instantiate a Sigma type with
                                        -- fresh internal type variables
skolemise :: Sigma -> Tc ([TcTv], Rho)  -- Instantiate a Sigma type
                                        -- with fresh skolem constants

        -- Unification
unify     :: Tau -> Tau -> Tc ()    -- Unification (may fail)

        -- Free type variables...
envTvs   :: Tc [TcTv]            -- ...of the envt
sigmaTvs :: Sigma -> Tc [TcTv]   -- ...of a Sigma type
rhoTvs   :: Rho   -> Tc [TcTv]   -- ...of a Rho type

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

Fig. 9: The `TcMonad` module

- It supports exceptions, when type inference fails (`check`).
- 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` and `zonkSigma`).

The function `check` (Figure 9) is typically used in a context like this

```
do { ...
   ; check (..condition..) "Error message"
   ; ... }
```

It checks its boolean argument; if it is `True`, `check` returns `()`; but if it is `False`, `check` raises an exception in the monad, passing the specified string as an error message.

Environment extension (`extendVarEnv`) is scoped, not a side effect. There is no need to restore the old environment after a call to `extendVarEnv`. For example, one might write:

```
do { ...
   ; extendVarEnv "x" ty
       (do { ...; t <- lookupVar "x"; ... })
```

```
    ; ...this code does not see the binding... }
```

The monad also maintains a single, ever-growing substitution that maps internal type variables (`TcTvs`) to monotypes — it does not affect quantified type variables at all. Unification extends the substitution by side effect; 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.

We will introduce the remaining functions in Section 5.5.

5.3 Simple inference

Figure 4, on page 14, expresses the 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:

```
⊢        inferRho  ::  Term -> Tc Rho
⊢^gen    inferSigma ::  Term -> Tc Sigma
⊢^inst    instSigma ::  Sigma -> Rho   -> Tc ()
⊢^subs    subsCheck ::  Sigma -> Sigma -> Tc ()
```

We begin by looking at `inferRho`, derived from $\vdash$. 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 a monotype*; as inference proceeds, unification extends an ever-growing substitution, which maps `TcTvs` to monotypes.

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 exp_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 exactly the same technology will prove useful in Section 6, to implement the bidirectional type rules of Section 4.8.

Here, for example is the `Lit` case for `checkRho`, which uses `unify` to ensure that the expected type `exp_ty` is indeed equal to `intTy`:

```
checkRho (Lit i) exp_ty = unify intTy exp_ty
```

Similarly here is the `App` case, 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[5]!

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) }
```

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.

The code for lambda abstraction uses `unifyFun` in a dual manner to split the expected type into the type of the bound variable and the type of the body; then we extend the environment with a new binding, and check the body.

```
checkRho (Lam var body) exp_ty
  = do { (pat_ty, body_ty) <- unifyFun exp_ty
       ; extendVarEnv var (ForAll [] pat_ty)
                          (checkRho body body_ty) }
```

All this follows directly from rule ABS of Figure 4.

5.5 Instantiation and generalisation

When we reach a `Var` (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
                             ; instSigma v_sigma exp_ty }
```

The function `instSigma` implements the judgement $\vdash^{inst}$, thus:

[5] 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 6.

```
instSigma :: Sigma -> Rho -> Tc ()
instSigma sigma exp_ty = do { rho <- instantiate sigma
                            ; unify 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 $\vdash^{gen}$ judgement in Figure 4:

```
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 9). 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[6].

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. Furthermore, `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.

[6] 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.

5.6 Subsumption

The code for a type-annotated expression can be read off Figure 4 just like the other cases:

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

The interesting part is the implementation of `subsCheck`, which implements the $\vdash^{subs}$ judgement (Figure 4). Here is the implementation:

```
subsCheck :: Sigma -> Sigma -> Tc ()
subsCheck sigma1 sigma2   -- Rule SKOL
  = do { (skol_tvs, rho2) <- skolemise sigma2
       ; subsCheckSR sigma1 rho2
       ; tvs1 <- sigmaTvs sigma1
       ; let bad_tvs = skol_tvs `intersect` tvs1
       ; check (null bad_tvs)
               "Type not polymorphic enough" }

subsCheckSR :: Sigma -> Rho -> Tc ()
subsCheckSR sigma1 rho2   -- Rule INST
  = do { rho1 <- instantiate sigma1
       ; subsCheckRR rho1 rho2 }

subsCheckRR :: Rho -> Rho -> Tc ()
subsCheckRR 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 `subsCheckSR` and `subsCheckRR` into the single function `subsCheck`.) The function `subsCheckSR`, corresponding to rule INST, is quite straightforward. The function `subsCheckRR` corresponds to rule MONO, and simply invokes the ordinary unifier to make the two monotypes equal.

So far so easy, but `subsCheck` itself (corresponding to rule SKOL) requires more care. We want to instantiate `sigma2` 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 `instantiate`, namely `skolemise`:

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

The idea is that `skolemise` 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 and flexible type variables*, so they are, in effect, constants. The `skolemise` function returns the newly-made rigid type variables, along with the instantiated type.

After recursively calling `subsCheckSR`, we must check the side condition $\overline{a} \notin ftv(\sigma)$ for rule SKOL, namely that the skolemised variables `skol_tvs` are not free in `sigma1`. You might wonder how this could possibly be the case, since `skol_tvs` are freshly made, but the recursive call to `subsCheckSR` might have bound a flexible type variable in `sigma1` to one of the skolems. That is why we wait until *after* the call to `subsCheckSR` before making the test. For example, consider the term:

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

Rule ANNOT will invoke a subsumption check that tries to confirm that the type of the body of the annotated term (`x` in this case) is at least as polymorphic as the type signature $\forall a\ .\ a \to a$. By this time, `x` will be in the environment with type β, an internal type variable, so we end up checking this judgement

$$\beta \quad \leq \quad \forall a\ .\ a \to a$$

We skolemise a, and then unify, which binds β to the type $a \to a$. But rule SKOL 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! The function `sigmaTvs`, like `rhoTVs`, takes account of the substitution, which is why it has a monadic type.

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), so that `sigma2` is closed. 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 6.4.

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 mono-types. So we can think of `subsCheck` as a kind of super-unifier.

5.7 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 TvFlavour

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

type Uniq = Int
```

A `TcTv` has a unique identity, which is just an `Int`; and a "flavour", which distinguishes ordinary internal type variables from rigid ones (see Section 5.6). Rigid type variables only have a `Name`, but flexible ones have a a mutable slot of type `TyRef`.

The `TyRef` 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.) We need to "lift" the standard operations over `IORefs` (reading, writing, etc) to the `Tc` monad:

```
newTcRef   :: a -> Tc (IORef a)
readTcRef  :: IORef a -> Tc a
writeTcRef :: IORef a -> a -> Tc ()
```

The fact that types contain these mutable reference is the reason that many of our operations over types — for example `rhoTvs` — are in the `Tc` monad.

The `Tc` monad, then, is the `IO` monad augmented with an environment, and a way to report failure[7]:

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

Throughout, we maintain the following invariant:

An internal type variable can only be substituted by a τ-type.

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

Similarly, if `subsCheck` (Section 5.6) is passed two type variables as its arguments, it will simply unify them. But if a different order of type inference first unified

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

those type variables with polytypes, the call to `subsCheck` would need to do a full subsumption check rather than simple unification.

In short, the invariant that an internal type variable can only be substituted by a τ-type ensures that the *result* of type inference does not depend on the *order* of type-inference. The invariant is, in turn, a direct consequence of predicativity (Section 3.4).

6 Inference for higher rank

Having now completed type inference for Damas-Milner, we are ready to extend the type-inference engine for higher-rank types.

6.1 Changes to the basic structure

In moving to higher rank, we make the following changes to our basic data types:

- The `Term` data type gets a new constructor, `ALam` for an annotated lambda:

  ```
  data Term = ... | ALam Name Sigma Term
  ```

- The `Arrow` constructor for types takes a `Sigma` on the left and a `Rho` on the right:

  ```
  data Rho = ... | Arrow Sigma Rho
  ```

  The type of the function `(-->)` is changed in the same way:

  ```
  (-->)    :: Sigma -> Rho -> Rho
  ```

Next, we consider the main judgement $\vdash$. 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. Much more attractive is to exploit the symmetry implied by the many syntactic forms for which Figure 7 has only one "polymorphic" rule, mentioning δ. Here is a neat way to express this idea in code:

```
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 5.7. 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.

6.2 Basic rules

Now we can look at the definition of `tcRho`. The code for variables is unchanged:

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

The difference is in `instSigma`, which implements our new "polymorphic" judgement $\vdash^{inst}_{\delta}$. Here is one possible implementation:

```
instSigma :: Sigma -> Expected Rho -> Tc ()
instSigma s1 exp_rho  = do { t1 <- instantiate s1
                           ; instRho t1 exp_rho }

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

We begin by instantiating the σ-type, and then split into two cases. For inference (rule INST1), we are done – all we need to do is to write the ρ-type into the result.

In the checking case (rule INST2), we check that the two ρ types are compatible by calling `subsCheckRR`. This call may have real work to do if either type is higher rank, so we discuss `subsCheckRR` later, in Section 6.4.

One reason for defining an auxiliary function `instRho` is that it is useful in other rules. For example, the literal case looks like this:

```
tcRho (Lit i) exp_ty = instRho intType exp_ty        -- Was unify
```

The call to `unify` is replaced by a call to `instRho`. To a first approximation, to move to higher rank, we simply replace calls to `unify` with calls to `instRho`!

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
       ; instRho res_ty exp_ty }  -- Was: unify
```

We infer the type of the function, and split its type into its argument and result parts. The `arg_ty` returned by `unifyFun` is a `Sigma`, since a σ-type occurs to the left of an arrow:

```
unifyFun :: Rho -> Tc (Sigma, Rho)
unifyFun (Arrow arg_ty res_ty) = return (arg_ty, res_ty)
unifyFun fun_ty = do { arg_rho <- newTyVar
                     ; res_ty  <- newTyVar
                     ; let arg_ty = ForAll [] arg_rho
                     ; unify fun_ty (arg_ty --> res_ty)
                     ; return (arg_ty, res_ty) }
```

One might wonder whether this is correct in the case where `fun_ty` is an internal type variable, especially one that has already been unified with a function type. But all is well; since an internal type variable can only unify with a τ-type, any such function type could only have a τ-type to the left of the arrow, which will unify quite happily with `ForAll [] arg_ty`. (An efficient implementation might nevertheless test for an already-bound internal type variable, simply to avoid the redundant `newTyVar` calls.)

Returning to the `App` case of `checkRho`, after decomposing the function type with `unifyFun`, we use `checkSigma` to check that the argument has the right type. Finally we use `instRho` (in place of `unify`) to 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 `instRho` instead of `unify` for the result type.

We will discuss `checkSigma` in Section 6.4, 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
        ; instSigma ann_ty exp_ty }
```

6.3 Abstractions

The only tricky case is that for abstractions. For an un-annotated lambda, we treat the inference and checking cases separately (rules ABS1 and ABS2 respectively):

```
tcRho (Lam var body) (Infer ref)
  = do { var_rho <- newTyVar
       ; let var_ty = ForAll [] var_rho
       ; body_ty <- extendVarEnv var var_ty (inferRho body)
       ; writeTcRef ref (var_ty --> body_ty) }

tcRho (Lam var body) (Check exp_ty)
  = do { (var_ty, body_ty) <- unifyFun exp_ty
       ; extendVarEnv var var_ty (checkRho body body_ty) }
```

In the inference case, we invent a fresh internal type variable to stand for the τ-type of the bound variable, extend the environment, infer the type of the body, and update the incoming reference with the function type `(var_ty --> body_ty)`. The checking case has an incoming type that we can decompose with `unifyFun`, giving a `Sigma` we bind to `var` in the environment, before checking the body.

The new syntactic form, an annotated lambda, also requires two rules (AABS1 and AABS2):

```
tcRho (ALam var var_ty body) (Infer ref)
  = do { body_ty <- extendVarEnv var var_ty (inferRho body)
       ; writeTcRef ref (var_ty --> body_ty) }

tcRho (ALam var var_ty body) (Check exp_ty)
  = do { (arg_ty, body_ty) <- unifyFun exp_ty
       ; subsCheck arg_ty var_ty
       ; extendVarEnv var var_ty (checkRho body body_ty) }
```

6.4 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 5.5. However, we also need `checkSigma`, which we used when type-checking an application in Section 6.2:

```
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 5.6 in two ways:

- We must deal with function types (Section 6.4.1).
- We must refine the implementation of skolemisation (Section 6.4.2).

We also offer a more efficient implementation of `checkSigma` in Section 6.4.3. The full code for be found in the Appendix.

6.4.1 Subsumption for function types

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

```
subsCheckRR (Arrow a1 r1) (Arrow a2 r2)
  = subsCheckFun a1 r1 a2 r2

subsCheckFun :: Sigma -> Rho -> Sigma -> Rho -> Tc ()
subsCheckFun a1 r1 a2 r2
  = do { subsCheck a2 a1 ; subsCheckRR r1 r2 }
```

We have separated out the "business end" of this case into an auxiliary function `subsCheckFun`, 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 `subsCheckFun` as before. Indeed, this will will work regardless of what the first argument is:

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

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

This approach neatly avoids one heffalump 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 `subsCheckFun` will do the rest.

6.4.2 Skolemisation revisited

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

$$\texttt{runST} :: \forall a\ .\ (\forall s\ .\ \texttt{ST}\ s\ a) \rightarrow a$$
$$\texttt{newRef} :: \forall s\ a\ .\ a \rightarrow \texttt{ST}\ s\ (\texttt{Ref}\ s\ a)$$

It does not matter exactly what these functions do, but they are described by Peyton Jones and Launchbury (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

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

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

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

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 $\texttt{ST}\ \beta\ (\texttt{Ref}\ \beta\ \texttt{Char})$ with $ST\ s\ \alpha$. This unification will succeed, binding β to the skolem constant s, and α to $(\texttt{Ref}\ s\ \texttt{Char})$. The trouble is that our skolem constant s has "escaped" into the free type variable α.

The bottom line is this: in the implementation of `subsCheck` we must check that the skolem constants do not end up free in `sigma2`, in addition to the existing check that they are not free in `sigma1`:

```
subsCheck sigma1 sigma2      -- Rule SKOL
  = do { (skol_tvs, rho2) <- skolemise sigma2
       ; subsCheckSR sigma1 rho2
       ; tvs1 <- sigmaTvs sigma1
       ; tvs2 <- sigmaTvs sigma2         -- This is the new bit!
       ; let bad_tvs = skol_tvs `intersect` (tvs1 `union` tvs2)
       ; check (null bad_tvs)
               "Type not polymorphic enough" }
```

This modification 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: it arises when `sigma2` is not a closed type. In the Damas-Milner system of Section 5 we

assumed that user type annotations were closed, and the only use of `subsCheck` passed a user type annotation as `sigma2`; hence `tvs2` is always empty. However, if the language were enhanced to support *open* type annotations – i.e. type annotations with free type variables, bound in some outer scope – then exactly the same problem would arise.

6.4.3 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) <- skolemise exp_ty
       ; checkRho expr rho
       ; env_tvs <- envTvs
       ; tvs2    <- sigmaTvs exp_ty
       ; let bad_tvs = skol_tvs `intersect` (env_tvs `union` tvs2)
       ; check (null bad_tvs) "Type not polymorphic enough" }
```

This function amounts to fusing the `quantify` from `inferSigma` with the `instantiate` 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.

6.5 Summary

We have now concluded the changes required to adapt a Damas-Milner type-inference engine to support higher-rank types. A crude way to summarise the changes is to count lines of code. The implementation in the Appendix is broken into three modules, with line count (including comments) as follows:

Module	Damas-Milner	Higher rank
`BasicTypes`	236	236
`TcMonad`	302	302
`TcTerm`	112	144
Total	640	682

The only significant changes are around 15 lines of code required to implement subsumption checking in `TcTerm`, plus about another 10 to handle the `ALam` cases in `tcRho`. Of course, a number of calls to `unify` have been replaced with calls to `subsCheck`, and so more than 55 lines have changed. Furthermore, in a real type checker, some carefully-placed calls to `zapToMonoType` are necessary (Section 7.4).

Overall, the extra compiler complexity required to support higher-rank types is remarkably small, even for the tiny language treated here. In a larger, more realistic language, `TcTerm` would be much larger (because there would be more term forms) but the same 55 extra lines would suffice, so in percentage terms the addition seems even smaller.

7 Handling a larger language

We have concentrated so far on a very small language, to focus attention on the central ideas. In this section we sketch briefly how to extend the framework to handle a full programming language, such as Haskell. Mostly it is a routine matter, but there are some interesting corners.

7.1 Rich patterns

In a real programming language, lambda abstractions and case expressions can bind rich, nested patterns. To give the idea, we might extend the syntax for terms thus:

Terms	t, u	$::=$	$\ldots$	
		$\mid$	$\backslash p \,.\, t$	Pattern abstraction
Patterns	p	$::=$	x	Variable
		$\mid$	$_$	Wild card
		$\mid$	$(p :: \sigma)$	Type annotated pattern
		$\mid$	(p_1, p_2)	Pair
		$\mid$	$\ldots$	

Corresponding to these patterns, we have a new judgement form:

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

which reads "pattern p has type σ and binds variables described by environment Γ". We put the Γ on the right as a clue that it is expected to be an output, rather than an input — but that makes no difference to the mathematical meaning of the judgement, of course. The typing rule for a pattern abstraction looks like this:

$$\frac{\vdash^{pat}_{\delta} p : \sigma, \Gamma' \qquad \Gamma, \Gamma' \vdash_{\delta} t : \rho}{\Gamma \vdash_{\delta} (\backslash p \,.\, t) : (\sigma \to \rho)} \text{ABS}$$

We only need one rule, because the cases that were previously treated separately in ABS1, ABS2, AABS1, and AABS2, are now handled by $\vdash^{pat}$. The same judgement $\vdash^{pat}$ can be used by all constructs that use pattern-matching: `case` expressions, list comprehensions, `do` notation, and so on,

Rather than give the rules for $\vdash^{pat}$, we will jump straight to the code. The main function `tcPat` takes an `Sigma` (not a `Rho`) as its expected type (because the argument type of the function can be a σ-type), and returns a list of `(Name,Sigma)` bindings (because the pattern can bind type-annotated variables to σ-types):

```
tcPat :: Pat -> Expected Sigma -> Tc [(Name,Sigma)]
```

A wild-card pattern is trivial: succeed immediately, returning the empty environment:

```
tcPat PWild exp_ty = return []
```

The variable-pattern case splits into two, just like the non-type-annotated lambda (Section 6.3).

```
tcPat (PVar v) (Infer ref) = do { ty <- newTyVar
                                ; return [(v,ForAll [] ty)] }
tcPat (PVar v) (Check ty)  = return [(v, ty)]
```

The code for a type-annotated pattern looks similar to that for a type-annotated expression (Section 6.2):

```
tcPat (PAnn p pat_ty) exp_ty = do { checkPat p pat_ty
                                  ; instPatSigma pat_ty exp_ty }
```

The new function `instPatSigma` checks that the expected type `exp_ty` is more polymorphic than the pattern type `pat_ty`:

```
instPatSigma :: Sigma -> Expected Sigma -> Tc ()
instPatSigma pat_ty (Infer ref)    = writeTcRef ref pat_ty
instPatSigma pat_ty (Check exp_ty) = subsCheck exp_ty pat_ty
```

Patterns do not become really interesting until one adds pattern-matching over data constructors, but we postpone that to the next sub-section. Meanwhile, we can use the new `tcPat` function to implement rule ABS for a pattern-matching lambda (constructor `PLam`). Because we have to decompose the function type, it still takes two cases:

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

tcRho (PLam pat body) (Check ty)
  = do { (arg_ty, res_ty) <- unifyFun ty
       ; binds <- checkPat pat arg_ty
       ; extendVarEnvList binds (checkRho body res_ty) }
```

Here, `inferPat` and `checkPat` are simple wrappers for `tcPat`, just as `inferRho` and `checkRho` are wrappers for `tcRho` (Section 6.1); and `extendVarEnvList` is like `extendVarEnv`, but extends an environment with a *list* of bindings.

7.2 Higher-ranked data constructors

It is easy to extend `tcPat`, as new patterns are added to the language. A particularly important example is that of data constructors, especially if they have higher-ranked types. For example, consider the following data type declaration, in an extended version of Haskell supporting higher-rank types:

```
data T = MkT (forall a. a -> a)
```

When *constructing* values of type `T`, we can simply treat the constructor `MkT` as an ordinary function, albeit with a higher-rank type:

$$\texttt{MkT} :: (\forall a\ .\ a \to a) \to \texttt{T}$$

When *pattern-matching* over values of type `T`, however, we need to add something new. For example, if we see a `case` expression thus:

```
case x of
   MkT v -> (v 3, v True)
```

we would like `v` to be attributed the type $\forall a\ .\ a \to a$ without the programmer having to write an explicit annotation. The data type declaration should be enough!

This is easy to achieve. First we extend `Pat` with a new form:

```
data Pat = ... | PCon Name [Pat]
```

where the `Name` is the name of a data constructor that is presumably bound in the type environment.

```
tcPat (PCon con ps) exp_ty
   = do { (arg_tys, res_ty) <- instDataCon con
        ; envs <- mapM check_arg (ps `zip` arg_tys)
        ; instPatSigma (ForAll [] res_ty) exp_ty
        ; return (concat envs) }
   where
     check_arg (p,ty) = checkPat p ty
```

The auxiliary function `instDataCon` looks up the data constructor in the environment, instantiates its type using `instantiate`, and splits out the argument types and result type:

```
instDataCon :: Name -> Tc ([Sigma], Tau)
```

Just as with a function application, the argument types of the constructor is pushed into the argument patterns.

7.3 Data constructors and predicativity

In the preceding discussion, we have implicitly assumed that data types can only be instantiated with monomorphic types. For example, consider:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

One can construct values of types such as `(Tree Int)` or `(Tree (Tree Int))`, but what about `Tree` $(\forall a \,.\, a \to \texttt{Int})$? More generally, in a type, can the argument of a type constructor be a σ-type, or must it be a τ-type? Well, the constructor `Leaf` is a polymorphic function of type $(\forall a \,.\, a \to \texttt{Tree}\ a)$, and our restriction to predicativity therefore requires that we instantiate `Leaf` only at a τ-type (Section 3.4). By far the simplest solution is to require that type constructors are parameterised only by monotypes. Then, just as `instSigma` instantiates a polymorphic function with fresh internal (mono-)type variables, so `instDataCon` instantiates the data constructor's type with fresh internal (mono-)type variable.

This approach is consistent with our general assumption of predicativity, and it also finesses some awkward efficiency questions. If one could have (say) a list of polymorphic functions, when one might ask whether the type $[\forall a \,.\, a \to a]$ is more polymoprhic than $[\texttt{Int} \to \texttt{Int}]$. One might argue that it should certainly be so, but there are complications. First, in general, the direction of the relationship depends on the *variance* of the type parameter — consider types like

```
data Contra a = Contra (a -> Int)
```

Here, `Contra` $(\texttt{Int} \to \texttt{Int})$ would be more, rather than less, polymorphic than `Contra` $(\forall a \,.\, a \to a)$. The situation gets even more complicated when there are multiple type arguments, when a type argument appears several times on the right-hand side, or when a type argument does not appear at all on the right-hand side (so-called phantom types). Second, if the system does type-directed translation (which we discuss in Section 8), one would actually need to traverse the entire list, coercing each function in the list from type $(\forall a \,.\, a \to a)$ to $(\texttt{Int} \to \texttt{Int})$. List traversal is a rather expensive operation to happen "behind the scenes" as a result of type inference.

Still, one can make a case for special treatment for *tuples*, which are ubiquitous in functional programs. Tuples are all co-variant, of course, and they come with special syntax for construction and pattern-matching. So a possible syntax for types could be this:

$$\begin{array}{lllll} \text{Polytypes} & \sigma & ::= & \forall \overline{a}.\rho & \\ \text{Rho-types} & \rho & ::= & \sigma \to \tau \mid (\sigma_1, \ldots, \sigma_n) \mid \tau & \\ \text{Monotypes} & \tau & ::= & \tau_1 \to \tau_2 \mid (\tau_1, \ldots, \tau_n) \mid K\ \overline{\tau} \mid a \end{array}$$

so that types like $(\forall a \,.\, a \to a, \texttt{Int})$ would be legal. Along with this would come special typing judgements for tuples:

$$\frac{\Gamma \vdash_{\Uparrow} t_i : \rho_i \quad (1 \le i \le n)}{\Gamma \vdash_{\Uparrow} (t_1, \ldots, t_n) : (\rho_1, \ldots, \rho_n)}\ \text{TUP1} \qquad \frac{\Gamma \vdash_{\Downarrow} t_i : \sigma_i \quad (1 \le i \le n)}{\Gamma \vdash_{\Downarrow} (t_1, \ldots, t_n) : (\sigma_1, \ldots, \sigma_n)}\ \text{TUP2}$$

There would be similar extra typing judgements for patterns. Lastly, one could add an extra case to the subsumption judgement:

$$\frac{\vdash^{subs} \sigma_i \le \sigma_i' \quad (1 \le i \le n)}{\vdash^{subs} (\sigma_1, \ldots, \sigma_n) \le (\sigma_1', \ldots, \sigma_n')}\ \text{SKOL}$$

These new typing rules lead directly to new cases in the implementation.

These extensions would allow one to construct, pass around, and pattern-match tuples with polymoprhic components. However, a function such as `fst :: (`$\forall a\ b\ .\ (a, b) \to a$ *can still only be used predicatively*, becuase it is an ordinary polymorphic function. For example, the application

$$\texttt{fst (id :: } \forall a\ .\ a \to a\texttt{, Int)}$$

would be rejected. Still, the situation is no different with higher rank functions (one cannot apply `map` to a higher-rank function, for the same reason), so perhaps it is acceptable. GHC does not currently implement the impredicative-tuple extension, so we do not have any concrete experience to report on this question.

7.4 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:

$$\frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\Uparrow} e_2 : \tau \quad \Gamma \vdash_{\Uparrow} e_3 : \tau}{\Gamma \vdash_{\Uparrow} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}\ \text{IF1} \qquad \frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\Downarrow} e_2 : \rho \quad \Gamma \vdash_{\Downarrow} e_3 : \rho}{\Gamma \vdash_{\Downarrow} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \rho}\ \text{IF2}$$

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
```

[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.

```
                                    ; return (Check ty) }
```

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

8 Type-directed translation

In this paper we have discussed type inference, which tells whether a term is well-typed. In some compilers, the type inference engine also performs a closely-related task, that of *type-directed translation*. For example, consider:

$$\texttt{concat = (\ xs -> foldr (++) Nil xs) :: } \forall a\ .\ [[a]] \to [a]$$

where the types of `foldr`, `(++)` and `Nil` are:

$$\begin{array}{rcl} \texttt{foldr} & :: & \forall x\ y\ .\ (x \to y \to y) \to y \to [x] \to y \\ \texttt{(++)} & :: & \forall z\ .\ [z] \to [z] \to [z] \\ \texttt{Nil} & :: & \forall a\ .\ [a] \end{array}$$

In System F, the definition of a polymorphic function has a *type abstraction* (which we denote `/\`a`. ...`), and each use of a polymorphic function requires a *type application* (which we denote `f {`t`}`). Furthermore, every binder is explicitly typed. With explicit type abstractions and applications, `concat` would look like this:

```
concat = /\a. \(xs:[[a]]).
         foldr {[a],[a]} ((++) {a}) (Nil {a}) xs
```

The "`/\a`" binds the type variable `a`; and the type applications instantiate the polymorphic functions `foldr`, `(++)`, and the constructor `Nil`, whose types we give above for reference. A type-directed compiler, such as GHC, actually translates the source program into System F as part of type inference. The translated program can be regarded as encoding a proof that the original program is indeed well typed. Then the rest of compilation can deal with an explicitly-typed program, which gives the compiler helpful information. Furthermore, type-checking the System F program at a later stage (which is easy to do) gives a very strong consistency check that the intermediate stages have not performed an invalid transformation (Peyton Jones & Santos, 1998; Tarditi *et al.*, 1996; Shao, 1997).

Matters become even more interesting when we consider subsumption. For example, suppose we have:

$$\begin{array}{rcl} \texttt{f} & :: & (\forall c\ .\ (c, c) \to (c, c)) \to \texttt{Int} \\ \texttt{g} & :: & \forall ab\ .\ (a, b) \to (b, a) \end{array}$$

and we are faced with the application `(f g)`. Now `g` is indeed more polymorphic than the argument required by `f`, but that cuts no ice with System F! We need to add extra type abstractions and applications to make the types match exactly, like this:

```
f (/\c. g {c, c})
```

We abstract over `c`, and then instantiate both `g`'s type arguments with `c`. Now indeed,

$$\texttt{(/\c. g \{c, c\})} :: \forall c\,.\,(c, c) \to (c, c)$$

as required.

8.1 Implementing type-directed translation

The question is this: *how can we adapt our type inference engine so that it performs type-directed translation at the same time as type inference*? The good news is that performing type-directed translation is very straightforward. First, we must add type abstraction and application to the `Term` data type:

```
data Term = ...
          | TyLam Name Term   -- Type abstraction
          | TyApp Term Tau    -- Type application
```

The extra constructors are only used in the output of type inference, not the input. Notice that the argument of a type application is a τ-type; remember that the system is predicative. Next, we need to adjust the type of `tcRho` to return a translated term:

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

where the returned `Term` has all the type abstractions and applications that are implicit in the source term. Similarly, `checkRho`, `inferRho`, and `tcPat` all return translated terms and patterns respectively.

For the most part, the changes are routine. For example, the code for lambdas becomes:

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

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

Our other key function, `subsCheck`, gets the following very interesting type:

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

The call `(subsCheck s1 s2)` returns a coercion that transforms a `Term` of type `s1` into a `Term` of type `s2`. The way to think of it is this: `subsCheck` proves that a type

`s1` is more polymorphic than a type `s2`; it returns a proof of this claim, in the form of a funtion that when applied to a term of type `s1` returns a term of type `s2`. We will see how to write `subsCheck` shortly, but let us first consider a call, in the `Var` case of `tcRho`:

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

Recall that `instSigma` is a derivative of `subsCheck` (Section 6.2), and hence also returns a `Term->Term` coercion function. We simply apply the function returned by `instSigma` to `(Var v)`, to coerce it to the expected type `exp_ty`.

8.2 Implementing subsCheck

The implementation of `subsCheck` is a straightforward extension of the code we developed in Sections 5.6 and 6.4. First, `subsCheckRR` deals with two ρ-types:

```
subsCheckRR :: Rho -> Rho -> Tc (Term -> Term)
subsCheckRR t1 (Arrow a2 r2)
  = do { (a1,r1) <- unifyFun t1; subsCheckFun a1 r1 a2 r2 }

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

subsCheckRR off_ty exp_ty
  = do { unify off_ty exp_ty
       ; return (\t -> t) }
```

The first two cases hand off the real work to `subsCheckFun`. The third case calls `unify`, and returns the identity coercion.

The function `subsCheckFun` recursively calls `subsCheck` and composes the two coercions it gets back:

```
subsCheckFun :: Sigma -> Rho -> Sigma -> Rho -> Tc (Term -> Term)
subsCheckFun a1 r1 a2 r2
  = do { co_arg <- subsCheck a2 a1
       ; co_res <- subsCheckRR r1 r2
       ; return (\f -> Lam "x" (co_res (App f (co_arg (Var "x"))))) }
```

The coercion function it returns takes a function-typed term, `f`, and produces the function-typed term

```
\x. co_res (f (co_arg x))
```

That is, first apply the argument coercion `co_arg` to `x`; then apply `f`, then coerce the result with the result coercion `co_res`[9].

In a similar way, type abstractions are generated by `subsCheck`, and type applications by `subsCheckSR` (see Section 5.6), but we omit the details here.

8.3 Patterns

One complication is that in principle *patterns* must be translated as well as *terms*. For example, consider:

```
f = (\(t::Int->Int). \x. t (t x)) :: (∀a . a → a) → Int → Int
```

This is well-typed in our system. The outer type signature gives a rather restrictive type to `f`, requiring `f` to be applied to a polymorphic argument, but the signature on `t` is more generous: any `Int->Int` function will do. When type-checking the pattern `(t::Int->Int)`, the call to `subsCheck` inside `checkPat` (Section 6.3) will generate a non-trivial coercion, *which must be recorded in the translated pattern*.

GHC does exactly this, and uses the coercions, recorded in the pattern, during the desugaring of nested pattern-matching, subsequent to type inference. Again, we omit the details.

8.4 Type classes

One of Haskell's most distinctive features is its *type class system*. Again, it turns out that the type inference engine we have described extends smoothly to embrace type classes, including their (non-trivial) type-directed translation. All that is needed is a mechanism to gather type constraints, which can conveniently be handled by the `Tc` monad, a constraint solver (which is entirely new), and a way to record the solution in the translated term (which works in much the same way as the type-directed translation we have already seen). The whole mechanism is almost entirely orthogonal to higher-rank types.

8.5 Summary

In this section we have briefly sketched how the type inference engine can be extended to support type-directed translation, including that required by Haskell's type classes. We have only given sketchy details, for reasons of space, but GHC uses precisely the scheme we sketch, so we know that it scales up without difficulty.

[9] The alert reader will notice that this formulation is not quite right, because the `Lam "x"` might capture a free variable `"x"` in `f`, but that is easily fixed by generating a fresh varible name, or by using an extra `let` binding.

9 Related work

In this section we discuss how our work fits into the wider context of research in type inference algorithms.

9.1 Finite-rank fragments of System F

System F is a very well-studied language whose type system is impredicative, and has arbitrary-rank types (Girard, 1990). It is extremely expressive: indeed, we take System F as the "gold standard" for expressiveness, to which we aspire. From a programming point of view, however, System F is extremely verbose and burdensome to write, because it is explicitly typed. Here is an example:

$$\Lambda a \;.\; \lambda(g : \forall b \;.\; b \to a) \;.\; (g\ [\texttt{Char}]\ \texttt{'x'},\ g\ [\texttt{Bool}]\ \texttt{True})$$

Every binder must be annotated with its type (e.g. $(g : \forall b \;.\; b \to a)$). Furthermore, the terms must include explicit type abstractions and type applications — the forms $\Lambda a \;.\; e$ and $e\ [\sigma]$ respectively.

Many people have studied the question: *if we erased from System F all the type abstractions, type applications, and binder annotations, could they be reconstructed by type inference*? The answer is a definite "no". Even the question "is any type at all derivable for this expression" is undecidable (Wells, 1999).

Well, then, perhaps there is a useful subset of System F for which we can perform type inference? This question has been studied by stratifying System F by rank; the rank-K subset of System F consists of all expressions that can be typed using types of rank $\leq K$. Kfoury and Wells show that typeability is decidable for rank ≤ 2, and undecidable for all ranks ≥ 3 (Kfoury & Wells, 1994). For the rank-2 fragment, the same paper gives a type inference algorithm. This inference algorithm is somewhat subtle, and has not, to our knowledge, been implemented in a production compiler.

9.2 MLF

A big disadvantage of the Kfoury/Wells approach is that the finite-rank fragments of System F do not have principal types. Given a typeable expression, their inference algorithm will find a type for it, but it cannot guarantee to find a *principal* type – that is, one that is more general than any other derivable type for the same expression. This is a serious problem in practice, where we want to infer the type of a function and expect that type to be compatible with all possible call sites for that function. This desire is especially pressing when we want to support separate compilation with stable interfaces.

Recently, Le Botlan and Rémy have described a new and ingenious type system, MLF, that avoids this problem (Le Botlan & Rémy, 2003). Like us, they accept that

the programmer will have to guide the type system, using annotations called "*guess points*". They recover principal types, for the annotated program, by defining a type system that is actually *more* expressive than System F, but in which every typeable System F program is typeable. Amazingly, they do all this while still retaining impredicativity. And, of course, every program typeable by Damas-Milner can be typed in MLF without any annotations at all.

The price they pay for these remarkable results is that (a) the type system, which the programmer must encounter, is pretty complicated, and (b) the type inference algorithm is likewise complicated[10].

Our system also has principal types for the annotated program, and can type ordinary Damas-Milner programs without annotations. But, like Haskell or ML, our system is predicative. Is the additional power of impredicativity worth the extra complexity? We have found it hard to find convincing examples that require impredicativity — but a few years ago no one thought much about higher-ranked types either. At least we can observe that there is a clear cost/benefit trade-off to be made, with our system and MLF occupying interestingly different points on the design spectrum.

9.3 Partial type inference

The idea of employing type annotations written by the programmer to guide type inference is well known. Pierce and Turner call it *partial type inference*[11] in their influential paper (1998): "the job of a partial type inference algorithm should be to eliminate especially those type annotations that are both *common* and *silly* – i.e. those that can neither be justified on the basis of their value as checked documentation, nor ignored because they are rare".

Their paper presents a particular instantiation of partial type inference, which they call *local type inference*, to which our work has many similarities. They employ the idea of pushing types inward to reduce the annotation burden; and we adopted their presentation of the type system using two judgements (one for inference and one for checking). However, the focus of their work is on type systems that allow sub-typing, such as System $F_{\leq}$. Even inferring type arguments (which is relatively simple in our work) then becomes tricky! These difficulties led them to a "fully un-curried" style of function application and abstraction, which is not necessary for us, as well as an interesting constraint solver that we do not need. Furthermore, in their system, no type can be inferred for a lambda abstraction, unless its binder is annotated; that is, they lack a rule like ABS2 from Figure 7.

[10] "Complicated" does not mean "inefficient". The Damas-Milner algorithm has complexity exponential in the program size, but it works perfectly well in practice, and so apparently does MLF.

[11] "Partial" in the sense that not every program that can be typed will be accepted by the inference algorithm, rather than in the sense that type inference may diverge.

One shortcoming of local type inference is that it only pushes *completely known* types inwards. For example, suppose `g` has type $\forall a\ .\ (\texttt{Int} \to a) \to a$, and consider the definition

```
f w = g (\x. x+w)
```

Since the call to `g` is instantiated at some unknown type α, local type inference will not push the partially-known type $\texttt{Int} \to \alpha$ into the argument of `g`, and the program will be rejected. Odersky, Zenger and Zenger developed a more sophisticated scheme, called *coloured local type inference*, that is capable of propagating partial, as well as total, type information down the tree (Odersky *et al.*, 2001). Their system is rather complex.

We have no such difficulties with this example, simply because we are not concerned with sub-typing. For us it is simple to pass partial information downwards, by passing `(Check t)` as the `Expected Rho` parameter to the inference engine, where `t` is a type with open internal type variables. On the other hand, being able to pass in partial type information could still be useful. Notably, in Section 4.8 we discussed the information-loss of rule APP2 in Figure 7. The work on coloured type inference was, like local type inference, carried out in the context of a sub-typing system. It is possible that it could be adapted for the higher-rank setting, but we have not yet attempted to do so, because we have not found motivating examples that are untypeable without it.

9.4 Improving error messages

One of the first published expositions of a practical, unification-based approach to type inference was Cardelli's paper (1987). This paper, along with many production compilers, takes the "top-down, left-right" approach, called Algorithm W (Damas & Milner, 1982). The algorithm takes a context Γ and a term t and infers a type σ for which $\Gamma \vdash^{gen} t : \sigma$. It works left-to-right in the following sense: given a term $(e_1\, e_2)$, say, it first infers a type for e_1 and then for e_2. As the algorithm proceeds, it incrementally solves the constraints by unification, carrying an ever-growing substitution (usually implemented by imperative updates of internal type variables).

The trouble is that a programming error in e_1 can sometimes lead the algorithm to report an error in e_2. For example:

```
f ys = head ys && ys
```

The uses of `ys` cannot both be correct — because the first implies that `ys` is a list, while the second implies that it is a boolean — but which is wrong? The left-to-right algorithm arbitrarily reports the second as an error, because when processing `head ys` it refines the type of `ys` to $[\alpha]$, for some unknown, internal type α. Indeed, early type-inference engines produce truly appalling error messages.

One big improvement is to use the "pushing types inwards" trick that we have

used extensively in this paper. While it does not solve the left-to-right problem completely, it dramatically improves error messages in practice, and rewards the programmer for supplying type signatures. For a long time this idea was folk lore, but its properties are studied by Lee and Yi (1998), where it is called Algorithm M. This approach is a "cheap and cheerful" approach to improving error messages: it is simple to implement, and gives a big improvement in most cases, but it does not *guarantee* an improvement.

A more principled alternative is to remove the arbitrary left-to-right order. Instead of incrementally solving the typing constraints by unification, get the inference algorithm to return a set of constraints, and solve them all together. Each constraint can carry a location to say which source location gave rise to it, so the error message can say "these two uses of `ys` are incompatible", rather than "the second use is wrong" or "the first use is wrong". Apart from generating better error messages, this approach scales better to richer type systems where the constraints are more complicated than simple equalities – for example, subtype constraints, or Haskell's class constraints. See Pottier and Rémy (2004) for a rather detailed treatment of this idea, which is also the basis for Helium's type checker (Heeren *et al.*, 2003).

Incidentally, it should be fairly easy to adapt the type inference engine in this paper to use the constraint-gathering approach. The `Tc` monad could carry an updatable bag of constraints; calls to `unify` would simply add a constraint to the bag, rather than solving the constraint immediately; and the constraint solver would be triggered by a call to `rhoTvs` or `sigmaTvs`. In short, almost all of the necessary changes could be hidden in the implementation of the monadic primitives of Figure 9.

This constraint-gathering approach eliminates the left-to-right bias but, at least in some formulations, it still suffers from the top-down bias. A yet more general idea, first suggested by Damas as Algorithm T in his thesis (1985), is to change the structure of the algorithm altogether. An Algorithm-T-style inference engine takes the term t as input and produces as output *both* a context Γ *and* type σ such that $\Gamma \vdash^{gen} t : \sigma$. This (Γ, σ) pair is called a *typing* for t. If there is a "best" or "most general" typing for every term t, then the type system has the principal typing property. What is particularly nice about this approach is that it is more compositional: to type a term `let` x `=` e_1 `in` e_2, compute the principal typing for e_1, and the principal typing for e_2, and combine the results to give the principal typing for the whole `let` term. In contrast, the more common, top-down approach first finds the type of e_1, extends the context Γ, and only then looks at e_2. This ordering means that programming errors in e_1 can be reported as type errors in e_2. Here is an example very similar to the earlier one:

```
f ys = let x = head ys
       in (x+1, ys+1)
```

These issues are discussed in detail by (Wells, 2002; Haack & Wells, 2003; Chitil, 2001). Sadly, neither the Damas-Milner type system, nor System F, enjoy the principal typing property. To achive that, one must move to *intersection types* (Kfoury & Wells, 1999).

10 Summary

This is a long paper, but it has a simple conclusion: higher-rank types are definitely useful, occasionally indispensable, and much easier to implement than one might guess. Every language should have them!

Acknowlegements

We would like to thank a number of people have given us extremely helpful feedback about earlier drafts of this paper: Umut Acar, Arthur Baars, Pal-Kristian Engstad, Matt Hellige, Dean Herrington, Greg Morrisett, Amr Sabry, Yanling Wang, Joe Wells, and Carl Witty. We are particularly indebted to Andres Löh and François Pottier for their detailed and insightful comments.

References

Baars, AL, & Swierstra, SD. (2002). Typing dynamic typing. *Pages 157–166 of: ACM SIGPLAN International Conference on Functional Programming (ICFP'02).* Pittsburgh: ACM.

Bird, Richard, & Paterson, Ross. (1999). De Bruijn notation as a nested datatype. *Journal of Functional Programming*, **9**(1), 77–91.

Cardelli, L. (1987). Basic polymorphic typechecking. *Science of Computer Programming*, **8**(2), 147–172.

Chitil, Olaf. (2001). Compositional explanation of types and algorithmic debugging of type errors. *ACM SIGPLAN International Conference on Functional Programming (ICFP'01).* ACM SIGPLAN notices, vol. 36, 10. Florence: ACM.

Clement, D, Despeyroux, J, Despeyroux, T, & Kahn, G. (1986). A simple applicative language: Mini-ML. *Pages 13–27 of: ACM Symposium on Lisp and Functional Programming.* ACM.

Damas, LMM. (1985). *Type assignment in programming languages (PhD thesis).* CST-33-85. Department of Computer Science, Edinburgh University.

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.

Gill, A, Launchbury, J, & Peyton Jones, SL. (1993). A short cut to deforestation. *Pages 223–232 of: ACM Conference on Functional Programming and Computer Architecture (FPCA'93).* Cophenhagen: ACM. ISBN 0-89791-595-X.

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

Haack, C, & Wells, JB. (2003). Type error slicing in implicitly typed, higher-order languages. *Programming Languages and Systems, 12th European Symposium on Programming (ESOP'03).* Lecture Notes in Computer Science. Springer Verlag.

Heeren, B, Hage, J, & Swierstra, SD. (2003). Scripting the type inference process. *In:* (ICFP03, 2003).

Hinze, Ralf. (2000). A new approach to generic functional programming. *Pages 119–132 of: 27th ACM Symposium on Principles of Programming Languages (POPL'00).* Boston: ACM.

Hinze, Ralf. (2001). Manufacturing datatypes. *Journal of Functional Programming*, **1**.

ICFP03. (2003). *ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*. Uppsala, Sweden: ACM.

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

Kfoury, AJ, & Wells, JB. (1994). A direct algorithm for type inference in the rank-2 fragment of the second-order lambda calculus. *Pages 196–207 of: ACM Symposium on Lisp and Functional Programming*. Orlando, Florida: ACM.

Kfoury, AJ, & Wells, JB. (1999). Principality and decidable type inference for finite-rank intersection types. *26th ACM Symposium on Principles of Programming Languages (POPL'99)*. San Antonio: ACM.

Lämmel, 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.

Le Botlan, D, & Rémy, D. (2003). MLF: raising ML to the power of System F. *In:* (ICFP03, 2003).

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.

Odersky, Martin, Zenger, Matthias, & Zenger, Christoph. (2001). Colored local type inference. *28th ACM Symposium on Principles of Programming Languages (POPL'01)*. London: ACM.

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

Peyton Jones, SL, & Santos, A. (1998). A transformation-based optimiser for Haskell. *Science of Computer Programming*, **32**(1-3), 3–47.

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.

Pottier, F, & Rémy, D. (2004). ML. Pierce, BC (ed), *Advanced topics in types and programming languages*. MIT Press.

Shao, Zhong. 1997 (June). An overview of the FLINT/ML compiler. *Proc. 1997 ACM SIGPLAN workshop on types in compilation (TIC'97)*.

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

Tarditi, D, Morrisett, G, Cheng, P, Stone, C, Harper, R, & Lee, P. (1996). TIL: A type-directed optimizing compiler for ML. *Pages 181–192 of: ACM Conference on Programming Languages Design and Implementation (PLDI'96)*. Philadelphia: ACM.

Wells, JB. (1999). Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, **98**, 111–156.

Wells, JB. (2002). The essence of principal typings. *Pages 913–925 of: Proc. 29th international colloquium on automata, languages, and programming.* Lecture Notes in Computer Science, vol. 2380. Springer Verlag.

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
  = instRho intType exp_ty

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

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

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

```
  = do { (var_ty, body_ty) <- unifyFun exp_ty
       ; extendVarEnv var var_ty (checkRho body body_ty) }

tcRho (Lam var body) (Infer ref)
  = do { var_ty <- newSigmaTyVar
       ; body_ty <- extendVarEnv var var_ty (inferRho body)
       ; writeTcRef ref (var_ty --> body_ty) }

tcRho (ALam var var_ty body) (Check exp_ty)
  = do { (arg_ty, body_ty) <- unifyFun exp_ty
       ; subsCheck arg_ty var_ty
       ; extendVarEnv var var_ty (checkRho body body_ty) }

tcRho (ALam var var_ty body) (Infer ref)
  = do { body_ty <- extendVarEnv var var_ty (inferRho body)
       ; writeTcRef ref (var_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
        ; instSigma ann_ty exp_ty }


-----------------------------------------
--      inferSigma and checkSigma
-----------------------------------------

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) <- skolemise 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
       ; check (null bad_tvs)
               (text "Type not polymorphic enough") }


-----------------------------------------
--      Subsumption checking           --
-----------------------------------------
```

```
subsCheck :: Sigma -> Sigma -> Tc ()
-- (subsCheck off exp) checks that
--     'off' is at least as polymorphic as 'exp'

subsCheck off_ty exp_ty         -- Rule SKOL
   = do { (skol_tvs, exp_ty') <- skolemise exp_ty
        ; subsCheckSR 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
        ; check (null bad_tvs)
                (vcat [text "Subsumption check failed:",
                       nest 2 (ppr off_ty),
                       text "is not as polymorphic as",
                       nest 2 (ppr exp_ty)])
     }

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

subsCheckSR off_ty exp_ty         -- Rule INST
   = do { off_ty' <- instantiate off_ty
        ; subsCheckRR off_ty' exp_ty }

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

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

subsCheckRR 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 ; subsCheckRR r1 r2 }

instSigma :: Sigma -> Expected Rho -> Tc ()
instSigma s1 e2 = do { t1 <- instantiate s1; instRho t1 e2 }

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

A.2 The monad and its operations

```
module TcMonad(
        Tc,     -- The monad type constructor
```

```
        runTc, ErrMsg, lift, check,

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

        -- Types and unification
        newTyVar, newSigmaTyVar,
        instantiate, skolemise, 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
    }

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)

check :: Bool -> Doc -> Tc ()
check True  d = return ()
check False d = failTc 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 } }
       ; 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 var ty (Tc m)
  = Tc (\env -> m (extend env))
  where
    extend env = env { var_env = addToFM (var_env env) var ty }

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)) }


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

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

newTyVar :: Tc Rho
newTyVar = do { tv <- newFlexiTyVar "t"; return (TcTv tv) }

newFlexiTyVar :: Name -> Tc TcTv
```

```
newFlexiTyVar n = do { uniq <- newUnique
                     ; tref <- newTcRef Nothing
                     ; return (Tv uniq (Flexi tref)) }

newRigidTyVar :: Name -> Tc TcTv
newRigidTyVar n = do { uniq <- newUnique
                     ; return (Tv uniq (Rigid n)) }


readTv  :: TcTv -> Tc (Maybe Tau)
readTv  (Tv _ (Flexi ref)) = readTcRef ref
readTv  (Tv _ (Rigid n))  = return Nothing

writeTv :: TcTv -> Tau -> Tc ()
-- When writing, the type variable must be flexible
writeTv (Tv _ (Flexi ref)) ty = writeTcRef ref (Just ty)

newUnique :: Tc Uniq
newUnique = Tc (\ (TcEnv {uniqs = ref}) ->
            do { uniq <- readIORef ref ;
               ; writeIORef ref (uniq + 1)
               ; return (Right uniq) })


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

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

skolemise :: Sigma -> Tc ([TcTv], Rho)
-- Instantiate the topmost for-alls of the argument type
-- with rigid type variables, and return those type variables
skolemise (ForAll [] ty) = return ([], ty)
skolemise (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 (all flexible)
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 (Arrow arg res) = do { tvs1 <- get_sigma_tvs arg
                                 ; tvs2 <- get_rho_tvs res
                                 ; return (tvs1 ++ tvs2) }
get_rho_tvs (TcTv tv)       = do { mb_ty <- readTv tv
                                 ; case mb_ty of
                                     Just ty -> get_rho_tvs ty
                                     Nothing -> return [tv] }
get_rho_tvs (TyVar _)       = return []
get_rho_tvs (TcTy tc)       = return []


-----------------------------------------
--      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 (TcTy tc)       = return (TcTy tc)
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 tv1) (TcTv tv2) | tv1 == tv2 = return ()
unify (TcTv tv) ty | isFlexi tv = unifyVar tv ty
unify ty (TcTv tv) | isFlexi tv = unifyVar tv ty

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

unify (TcTy tc1) (TcTy tc2)
  | tc1 == tc2
  = return ()

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

----------------------------------------
unifyVar :: TcTv -> Tau -> Tc ()
-- Invariant: tv1 is a flexible type variable
unifyVar tv1 ty2        -- Check whether tv1 is bound
  = do { mb_ty1 <- readTv tv1
       ; case mb_ty1 of
           Just ty1 -> unify ty1 ty2
           Nothing  -> unifyUnboundVar tv1 ty2 }

unifyUnboundVar :: TcTv -> Tau -> Tc ()
-- Invariant: the flexible type variable tv1 is not bound
unifyUnboundVar tv1 ty2@(TcTv tv2)
  = do { -- We know that tv1 /= tv2 (else the
         -- top case in unify would catch it)
```

```
         mb_ty2 <- readTv tv2
       ; case mb_ty2 of
           Just ty2 -> unify (TcTv tv1) ty2
           Nothing  -> writeTv tv1 ty2 }

unifyUnboundVar tv1 ty2
  = do { tvs2 <- get_rho_tvs ty2
       ; if tv1 `elem` tvs2 then
            occursCheckErr tv1 ty2
         else
            writeTv tv1 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 Name Term         -- \ x -> x
          | ALam Name Sigma Term  -- \ x -> x
          | Let Name Term Term    -- let x = f y in x+1
          | Ann Term Sigma        -- (f 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
         | TcTv  TcTv         -- A temporary, typechecker type var
         | TcTy TyCon         -- Type constants

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

data TcTv = Tv Uniq TvFlavour

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

data TvFlavour = Flexi TyRef    -- Can unify with any tau-type

               | Rigid Name     -- Must not unify with anything
                                -- except a type variable
                                -- The Name just improves error messages

isFlexi :: TcTv -> Bool
isFlexi (Tv _ (Flexi _)) = True
```

```
isFlexi other            = False

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

type Uniq = Int

data TyCon = IntT | BoolT
           deriving( Eq )


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

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

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

intType, boolType :: Rho
intType  = TcTy IntT
boolType = TcTy 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 (TyVar n)       = fromMaybe (TyVar n) (lookup n env)
subst env (TcTv tv)       = TcTv tv
subst env (TcTy tc)       = TcTy tc

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 (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 v e)     = sep [char '\\' <> pprName v <> text ".", ppr e]
   ppr (ALam v t e)  = sep [char '\\' <> brackets (pprName v <> dcolon <> ppr t) <> 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)
```

```
-------------- 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 (Flexi  _)) = 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 types are 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" <+>
                                hsep (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 (TcTy tc)  = ppr_tc tc
ppr_type (TyVar n)  = pprName n
ppr_type (TcTv tv)  = ppr tv

ppr_tc IntT  = text "Int"
ppr_tc BoolT = text "Bool"
```