Practical type inference for arbitrary-rank types

14 September 2011

Simon Peyton Jones
Microsoft Research

Dimitrios Vytiniotis
University of Pennsylvania

Stephanie Weirich
University of Pennsylvania

Mark Shields
Microsoft

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 (impredicative) 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 requires many annotations. We show how to use local type inference (invented by Pierce and Turner) to greatly reduce the annotation burden, to the point where higher-rank types become eminently usable.

Higher-rank types have a very modest impact on type inference. We substantiate this claim in a very concrete way, by presenting a complete type-inference engine, written in Haskell, for a traditional Damas-Milner type system, and then showing how to extend it for higher-rank types. We write the type-inference engine using a monadic framework: it turns out to be a particularly compelling example of monads in action.

The paper is long, but is strongly tutorial in style. Although we use Haskell as our example source language, and our implementation language, much of our work is directly

applicable to any ML-like functional language.

The online version

The paper is published in the Journal of Functional Programming 17(1), 2007.

This online version embodies minor corrections or clarifications compared to the print version. It is available at

`http://research.microsoft.com/~simonpj/papers/higher-rank`

At the same URL you can find all the code described in the paper, and a Technical Appendix giving the proofs.

Here is a brief summary of the changes compared to the published version. Page references are to the print version (so they stay stable), which you can find at the above URL.

Nov 06 Minor wording changes and clarifications. Thanks to Norman Ramsey.

Feb 07 Modifications related to multi-branch constructs, Section 7.1. Thanks to Chuan-kai Lin.

Jan 10 Three spelling errors. Thanks to Gabor Greif.

Mar 11 Wording improvement in 4.8; thanks to David Sankel.

Sept 11 Section 4.8: a superscript should be a subscript. Thanks to Gabor Greif. Add a missing write-ref in Section 7.2. Simplify Section 8.1, removing some bogus code.

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 ever 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, the Glasgow Haskell Compiler (GHC), which implements the type system of this paper, 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 programmer-supplied type signature for `f`, which expresses the polymorphic type of `f`'s argument. (The explicit "`forall`" is GHC's concrete syntax for universal quantification "$\forall$".)

Our work draws together and applies Odersky & Läufer's type system for arbitrary-rank types (Odersky & Lufer, 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 only a 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 should not invest 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).

Although we use Haskell as our example source language, and as our implementation language, almost all our work is directly applicable to any functional language. In a language that has side effects, extra care would be required at one or two points.

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 $\forall a.a \rightarrow \texttt{m}\ a$.

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

[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.


```
mapM :: Monad m -> (a -> m b) -> [a] -> m [b]
mapM m@(Mon { 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 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 *etal.*, 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 & PeytonJones, 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 $\forall a\ b.(a \to b) \to \texttt{Term}\ a \to \texttt{Term}\ b$ 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

The *rank* of a type describes the depth at which universal quantifiers appear contravariantly (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:

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

Throughout this paper we will use the term "*monotype*", and the symbol τ, for a rank-zero type; monotypes have no universal quantifiers whatsoever. We use the term "*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 & Lufer, 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. In this case, annotating `x` directly would not be too bad:

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

But one would not want to annotate `x` *and* provide a separate type signature; and if `f` had multiple clauses one would tiresomely have to repeat the annotation. 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 4.7.

3.3 Subsumption

Suppose that we have variables bound with the following types:

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

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, $\sigma_1 \rightarrow \texttt{Int}$ is more polymorphic than $\sigma_2 \rightarrow \texttt{Int}$ if σ_1 is *less* polymorphic than σ_2. For example, consider

$$\begin{array}{lcl} \texttt{g} & :: & ((\forall b.[b] \rightarrow [b]) \rightarrow \texttt{Int}) \rightarrow \texttt{Int} \\ \texttt{k1} & :: & (\forall a.a \rightarrow a) \rightarrow \texttt{Int} \\ \texttt{k2} & :: & ([\texttt{Int}] \rightarrow [\texttt{Int}]) \rightarrow \texttt{Int} \end{array}$$

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

$$((\forall a.a \rightarrow a) \rightarrow \texttt{Int})$$

is *less* polymorphic than

$$((\forall b.[b] \rightarrow [b]) \rightarrow \texttt{Int})$$

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

Is it legal to have the type (`Tree` $(\forall a.a \rightarrow a)$); 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 allows 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 ML^F shows (Le Botlan & Rémy, 2003), but doing so adds significant new complications to both the type system and the implementation—see 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 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 in-formally 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 re-cast 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 (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

	Rank 1 $\rho ::= \tau$	Arbitrary rank $\rho ::= \tau \mid \sigma \to \sigma$
Not syntax-directed Does not lead to an algorithm	Damas-Milner Section 4.2, page 13 Figure 3	Odersky-Läufer Section 4.5, page 19 Figure 5
Syntax-directed Algorithm can be read off	Damas-Milner Section 4.3, page 14 Figure 4	This paper Section 4.6, page 20 Figures 6, 7
Bidirectional Algorithm can be read off		This paper Section 4.7, page 24 Figure 8

Type contexts	Γ	$::=$	$\Gamma, x : \sigma \mid \epsilon$
Polytypes	σ	$::=$	$\forall \overline{a} \,.\, \rho$
Rho-types	ρ	$::=$	*See table above*
Monotypes	τ	$::=$	$\texttt{Int} \mid \tau_1 \to \tau_2 \mid a$
Type variables	a, b		

Fig. 1: Road map

we derive a syntax-directed system, and then further develop that into a so-called bidirectional system, 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, written using "::" on both terms and abstractions, are the

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

Fig. 2: Syntax of the source language

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. In this paper we will assume that the type annotations are *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 here because it opens up a whole new design space that distracts from our main theme.

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 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 $\forall a.(\forall b.(a, b) \rightarrow (b, a)) \rightarrow [a] \rightarrow [a]$. (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 $\mathit{ftv}(\sigma)$ to be the free type variables of σ, and extend the function to type contexts in the obvious way: $\mathit{ftv}(\Gamma) = \bigcup\{\mathit{ftv}(\sigma) \mid (x : \sigma) \in \Gamma\}$. 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:

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

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

$$\boxed{\Gamma \vdash t : \sigma}$$

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

$$\frac{\Gamma, (x : \tau) \vdash t : \rho}{\Gamma \vdash (\backslash x\,.\,t) : (\tau \to \rho)}\ \text{ABS} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \tau \to \rho \\ \Gamma \vdash u : \tau\end{array}}{\Gamma \vdash t\ u : \rho}\ \text{APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \sigma \\ \Gamma, x : \sigma \vdash t : \rho\end{array}}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \text{LET} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash (t\texttt{::}\sigma) : \sigma}\ \text{ANNOT}$$

$$\frac{\begin{array}{c}\overline{a} \notin \mathit{ftv}(\Gamma) \\ \Gamma \vdash t : \rho\end{array}}{\Gamma \vdash t : \forall \overline{a}\,.\,\rho}\ \text{GEN} \qquad \frac{\Gamma \vdash t : \forall \overline{a}\,.\,\rho}{\Gamma \vdash t : [\overline{a \mapsto \tau}]\ \rho}\ \text{INST}$$

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

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. In the Damas-Milner system we omit the type-annotated lambda ($\backslash(x::\sigma).t$), because a Damas-Milner lambda can only abstract over a monotype, and that is adequately dealt with by the un-annotated lambda.

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, one 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

$$\text{Rho-types} \quad \rho \;::=\; \tau$$

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

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \textsc{int} \qquad \frac{\vdash^{inst} \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash x : \rho}\ \textsc{var}$$

$$\frac{\Gamma, (x : \tau) \vdash t : \rho}{\Gamma \vdash (\backslash x \,.\, t) : (\tau \to \rho)}\ \textsc{abs} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \tau \to \rho \\ \Gamma \vdash u : \tau\end{array}}{\Gamma \vdash t\ u : \rho}\ \textsc{app}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{poly} u : \sigma \\ \Gamma, x : \sigma \vdash t : \rho\end{array}}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \textsc{let} \qquad \frac{\begin{array}{c}\Gamma \vdash^{poly} t : \sigma' \\ \vdash^{sh} \sigma' \leq \sigma \qquad \vdash^{inst} \sigma \leq \rho\end{array}}{\Gamma \vdash (t\texttt{::}\sigma) : \rho}\ \textsc{annot}$$

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

$$\frac{\overline{a} = ftv(\rho) - ftv(\Gamma) \quad \Gamma \vdash t : \rho}{\Gamma \vdash^{poly} t : \forall \overline{a}\,.\,\rho}\ \text{GEN} \qquad \frac{}{\vdash^{inst} \forall \overline{a}\,.\,\rho \ \leq\ [\overline{a \mapsto \tau}]\ \rho}\ \text{INST}$$

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

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

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

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 $\vdash^{poly}$ which infers a polytype for a term. It is used in rule LET to type the right-hand side of the `let`. 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.

There is one further judgement, $\vdash^{sh}$, which we discuss very shortly, in Section 4.4.

We can very nearly regard these new 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:

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

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 to choose the types $\overline{\tau}$ to use when instantiating σ.

The distinction between syntax-directed and non-syntax-directed formulations of typing judgements is well known. The latter is more simple, elegant, and abstract. The former is more bulky, using auxiliary judgements to avoid duplication and, precisely because it is closer to an algorithm, is more concrete. However, although the trade-off 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.

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

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:

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

This term is well typed, because the most general type of `(\x.x)` is $\forall a.a \rightarrow a$, and that is certainly more general than $\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]}$. The annotation is a type *restriction*, because the annotated term must only be used at the specified type. For example, this term is illegal:

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

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. 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 $\vdash^{poly}$;
- 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 $\vdash^{sh}$, shown in Figure 4;

- Finally, instantiate σ, using $\vdash^{inst}$.

The new judgement form

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

means "the offered type σ_{off} is at least as polymorphic as the required type σ_{req}". In the rest of the paper we will often say "more polymorphic than" instead of the more precise but clumsier "at least as polymorphic as". The judgement embodies a simplified form of the subsumption relationship of Section 3.3—simplified in that it only deals with rank-1 polytypes. The superscript "*sh*" is used to indicate *shallow subsumption*; will encounter richer versions of subsumption shortly.

Unlike $\vdash^{inst}$, the subsumption judgement compares two *polytypes*. For example:

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

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 $\vdash^{sh}$, in Figure 4, because they play a central role in this paper. We reproduce them here for convenience:

$$\frac{\overline{a} \notin \mathit{ftv}(\sigma) \quad \vdash^{sh} \sigma \leq \rho}{\vdash^{sh} \sigma \leq \forall \overline{a}\,.\,\rho}\ \textsc{skol} \qquad \frac{\vdash^{sh} [\overline{a \mapsto \tau}]\,\rho_1 \leq \rho_2}{\vdash^{sh} \forall \overline{a}\,.\,\rho_1 \leq \rho_2}\ \textsc{spec} \qquad \frac{}{\vdash^{sh} \tau \leq \tau}\ \textsc{mono}$$

Rule MONO deals with the trivial case of two monotypes. When quantifiers are involved, to prove that $\sigma_{\mathit{off}} \leq \sigma_{req}$, for *any* given instantiation of σ_{req} we must be able to find an instantiation of σ_{off} that makes the two types match. In formal notation, to prove that $\forall \overline{a}.\rho_{\mathit{off}} \leq \forall \overline{b}.\rho_{req}$ we must prove that

$$\forall \overline{\tau_b}\ \exists \overline{\tau_a} \text{ such that } [\overline{a \mapsto \tau_a}]\rho_{\mathit{off}} \leq [\overline{b \mapsto \tau_b}]\rho_{req}$$

To this end, rule SPEC is straightforward: it allows us to instantiate the outermost type variables of σ_{off} arbitrarily to match ρ_{req}. But how can we check that σ_{off} can be instantiated by SPEC to match *any* instantiation of σ_{req}? Suppose we were to instantiate the outermost type variables of σ_{req} to arbitrary, completely fresh type constants, called *skolem constants*. If, having done this we can *still* make σ_{off} match, then we will have shown that indeed σ_{off} is at least as polymorphic as σ_{req}. Cunningly, rule SKOL does not actually instantiate σ_{req} with fresh constants; instead, it simply checks that the type variables of σ_{req} are fresh with respect to σ_{off} (perhaps by alpha-renaming σ_{req}); then these type variables will themselves serve very nicely as skolem constants, so we can vacuously instantiate $\forall\overline{a}.\rho$ with the types $\overline{a}$ to get ρ. That is the reason for the side condition in SKOL, $\overline{a} \notin \mathit{ftv}(\sigma)$.

Notice that one has to apply SKOL before SPEC, because the latter assumes a ρ type to the right of the $\leq$. That is, we *first* instantiate σ_{req} with skolem constants, and *then* choose how to instantiate σ_{off} to make it match. Let us take a particular example. To prove that

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

first use SKOL to skolemise b and c, checking that b and c are not free in $\forall a.a \rightarrow a$, and then use SPEC to instantiate a with the type (b, c). The derivation looks like

Rho-types $\rho ::= \tau \mid \sigma \to \sigma'$

$$\boxed{\Gamma \vdash t : \sigma}$$

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

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

$$\frac{\Gamma \vdash t : (\sigma \to \sigma') \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \sigma'}\ \text{APP} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \texttt{(}t\texttt{::}\sigma\texttt{)} : \sigma}\ \text{ANNOT}$$

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \sigma'}\ \text{LET} \qquad \frac{\overline{a} \notin ftv(\Gamma) \quad \Gamma \vdash t : \rho}{\Gamma \vdash t : \forall \overline{a}.\rho}\ \text{GEN} \qquad \frac{\Gamma \vdash t : \sigma \quad \vdash^{ol} \sigma \leq \sigma'}{\Gamma \vdash t : \sigma'}\ \text{SUBS}$$

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

$\overline{a} \notin ftv(\sigma)$

$$\frac{\vdash^{ol} \sigma \leq \rho}{\vdash^{ol} \sigma \leq \forall \overline{a}\,.\,\rho}\ \text{SKOL} \qquad \frac{\vdash^{ol} [\overline{a \mapsto \tau}]\ \rho_1 \leq \rho_2}{\vdash^{ol} \forall \overline{a}\,.\,\rho_1 \leq \rho_2}\ \text{SPEC}$$

$$\frac{\begin{array}{c}\vdash^{ol} \sigma_3 \leq \sigma_1 \\ \vdash^{ol} \sigma_2 \leq \sigma_4\end{array}}{\vdash^{ol} (\sigma_1 \to \sigma_2) \leq (\sigma_3 \to \sigma_4)}\ \text{FUN} \qquad \frac{}{\vdash^{ol} \tau \leq \tau}\ \text{MONO}$$

Fig. 5: The Odersky-Läufer type system

this:

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

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.

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

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

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

  Crucially, a polytype may appear in both the argument and result positions of a function type, and hence polytypes may be of *arbitrary rank*. Providing this freedom is the whole point of this paper.
- The 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 "at least as polymorphic as" condition is done by the type-subsumption judgement, $\vdash^{ol}$,

shown in Figure 5.

The definition of subsumption $\vdash^{ol}$ in Figure 5 is just like that of $\vdash^{sh}$ 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:

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

None of these types would have been syntactically legal in the Damas-Milner system. However, as we shall see in the next subsection, $\vdash^{ol}$ is a little too small; that is, it does not relate enough types.

4.6 A syntax-directed higher-rank system

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 `lets` (Figure 4). The obvious thing to do is simply to use the same idea at higher rank, which is done in Figure 6. Notice that the specialisation judgement, $\vdash^{inst} \sigma \leq \rho$, instantiates only the *outermost* quantified type variables of σ; and similarly

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

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

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

$$\frac{\Gamma, x : \tau \vdash t : \rho}{\Gamma \vdash (\backslash x\,.\,t) : (\tau \to \rho)}\ \text{ABS} \qquad \frac{\Gamma, x : \sigma \vdash t : \rho}{\Gamma \vdash (\backslash(x::\sigma)\,.\,t) : (\sigma \to \rho)}\ \text{AABS}$$

$$\frac{\Gamma \vdash t : (\sigma_1 \to \sigma_2) \qquad \Gamma \vdash^{poly} u : \sigma' \qquad \vdash^{dsk} \sigma' \leq \sigma_1 \qquad \vdash^{inst} \sigma_2 \leq \rho}{\Gamma \vdash t\ u : \rho}\ \text{APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{poly} t : \sigma' \\ \vdash^{dsk} \sigma' \leq \sigma \qquad \vdash^{inst} \sigma \leq \rho\end{array}}{\Gamma \vdash \texttt{(}t\ \texttt{::}\ \sigma\texttt{)} : \rho}\ \text{ANNOT} \qquad \frac{\begin{array}{c}\Gamma \vdash^{poly} u : \sigma \\ \Gamma, x : \sigma \vdash t : \rho\end{array}}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \text{LET}$$

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

$\overline{a} = ftv(\rho) - ftv(\Gamma)$

$$\frac{\Gamma \vdash t : \rho}{\Gamma \vdash^{poly} t : \forall \overline{a} . \rho}\ \textsc{gen} \qquad \frac{}{\vdash^{inst} \forall \overline{a} . \rho \ \leq \ [\overline{a \mapsto \tau}]\, \rho}\ \textsc{inst}$$

Fig. 6: Syntax-directed higher-rank type system

the generalisation judgement, $\Gamma \vdash^{poly} t : \sigma$, generalises only the *outermost* type variables of σ. Any polytypes hidden under arrows are unaffected.

Just as in the syntax-directed Damas-Milner system of Figure 4, we must invoke subsumption in rule ANNOT of Figure 6, but we use yet another form of subsumption, $\vdash^{dsk}$, for reasons we discuss next. The other new feature of the rules is that in rule APP we must use $\vdash^{poly}$ to infer a polytype σ' for the argument, because the function may require the argument to have a polytype σ_1. These two types may not be identical, because the argument may be more polymorphic than required, so again $\vdash^{dsk}$ is used to marry up the two.

4.6.1 A problem with subsumption

In the new syntax-directed rules we have used a new form of subsumption (not yet defined), which we write $\vdash^{dsk}$. If we instead used the Odersky/Läufer subsumption, $\vdash^{ol}$, the type system would be perfectly sound, but it it would type fewer programs

than the non-syntax-directed system of Figure 5. To see why, consider this program:

```
let f = \x.\y.y in (f :: ∀a.a → (∀b.b → b))
```

A Haskell programmer would expect to infer the type $\forall ab.a \rightarrow b \rightarrow b$ for the `let`-binding for `f`, and that is what the rules of Figure 6 would do. The type-annotated occurence of `f` then requires that `f`'s type be more polymorphic than the supplied signature, but alas, under the subsumption rules of Figure 5, it is simply not the case that

$$\vdash^{ol} \forall ab.a \rightarrow b \rightarrow b \quad \leq \quad \forall a.a \rightarrow (\forall b.b \rightarrow b)$$

although the converse is true. On the other hand the typing rules of Figure 5 *could* give the let-binding the type $\forall a.a \rightarrow (\forall b.b \rightarrow b)$ and then $\vdash^{ol}$ would succeed. In short the syntax-directed rules do not find the most general type for `f`, under the ordering induced by $\vdash^{ol}$.

One obvious solution is to fix Figure 6 to infer the type $\forall a.a \rightarrow (\forall b.b \rightarrow b)$ for the `let`-binding for `f`. Odersky and Läufer's syntax-directed version of their language does this simply by generalising every lambda body in rules ABS and AABS, so that the ∀'s in the result type occur as far to the right as possible. Here is the modified rule ABS

$$\frac{\Gamma, x : \tau \vdash^{poly} t : \sigma}{\Gamma \vdash (\backslash x \,.\, t) : (\tau \rightarrow \sigma)} \text{EAGER-ABS}$$

We call this approach *eager generalisation*; but we prefer to avoid it. A superficial but practically-important difficulty is that it yields inferred types that programmers will find unfamiliar. Furthermore, if the programmer adds a type signature, such as `f::`$\forall ab.a \rightarrow b \rightarrow b$, he may make the function less general without realising it. Finally, there is a problem related to conditionals. Consider the term

```
if ... then (\x.\y.y) else (\x.\y.x)
```

This term will type fine in Haskell, but eager generalisation would yield $\forall a.a \rightarrow (\forall b.b \rightarrow b)$ for the `then` branch, and $\forall a.a \rightarrow (\forall b.b \rightarrow a)$ for the `else` branch—and it is now un-clear how to unify these two types. Conditionals are not part of the syntax we treat formally, thus far, but we return to this question in Section 7.1.

4.6.2 The solution: deep skolemisation

Fortunately, another solution is available. The difficulty arises because it is not the case that

$$\vdash^{ol} \forall ab.a \to b \to b \quad \leq \quad \forall a.a \to (\forall b.b \to b)$$

But that is strange, because the two types are isomorphic[4]. So, from a semantic point of view, the two types should be equivalent; that is, we would like *both* of the

[4] More concretely, if $f : \forall ab.a \to b \to b$, then we can construct a System-F term of type $\forall a.a \to (\forall b.b \to b)$, namely:

$$(\Lambda a.\lambda(x:a).\Lambda b.\lambda(y:b).\, f\ a\ b\ x\ y) : \forall a.a \to (\forall b.b \to b)$$

$$\boxed{pr(\sigma) = \forall\overline{a}.\rho}$$

$$\frac{pr(\rho_1) = \forall\overline{b}.\rho_2 \quad \overline{a} \cap \overline{b} = \emptyset}{pr(\forall\overline{a}.\rho_1) = \forall\overline{a}\overline{b}.\rho_2}\ \text{PRPOLY} \qquad \frac{pr(\sigma_2) = \forall\overline{a}.\rho_2 \quad \overline{a} \cap ftv(\sigma_1) = \emptyset}{pr(\sigma_1 \to \sigma_2) = \forall\overline{a}.\sigma_1 \to \rho_2}\ \text{PRFUN}$$

$$\frac{}{pr(\tau) = \tau}\ \text{PRMONO}$$

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

$$\frac{pr(\sigma_2) = \forall\overline{a}\,.\,\rho \qquad \overline{a} \notin ftv(\sigma_1) \qquad \vdash^{dsk*} \sigma_1 \leq \rho}{\vdash^{dsk} \sigma_1 \leq \sigma_2}\ \text{DEEP-SKOL}$$

$$\boxed{\vdash^{dsk*} \sigma \leq \rho}$$

$$\frac{\vdash^{dsk*} [\overline{a \mapsto \tau}]\rho_1 \leq \rho_2}{\vdash^{dsk*} \forall\overline{a}\,.\,\rho_1 \leq \rho_2}\ \text{SPEC} \qquad \frac{\vdash^{dsk} \sigma_3 \leq \sigma_1 \qquad \vdash^{dsk*} \sigma_2 \leq \rho_4}{\vdash^{dsk*} (\sigma_1 \to \sigma_2) \leq (\sigma_3 \to \rho_4)}\ \text{FUN}$$

MONO

$$\frac{}{\vdash^{dsk*} \tau \leq \tau}\ \text{MONO}$$

Fig. 7: Subsumption with deep skolemisation

following to hold:

$$\begin{array}{rcl} \forall ab.a \to b \to b & \leq & \forall a.a \to (\forall b.b \to b) \\ \forall a.a \to (\forall b.b \to b) & \leq & \forall ab.a \to b \to b \end{array}$$

Hence, perhaps we can solve the problem by *enriching the definition of subsumption*, so that the type systems of Figure 5 and 6 admit the same programs. That is the reason for the new subsumption judgement $\vdash^{dsk}$, defined in Figure 7. This relation subsumes $\vdash^{ol}$; it relates strictly more types.

The key idea is that in DEEP-SKOL (Figure 7), we begin by pre-processing σ_2 to float out all its $\forall$s that appear to the right of a top level arrow, so that they can be skolemised immediately. We call this rule "DEEP-SKOL" because it skolemises quantified variables even if they are nested inside the result type of σ_2. The floating process is done by an auxiliary function $pr(\sigma)$, called *weak prenex conversion*, also

Likewise, if $g : \forall a. \to (\forall b.b \to b)$ then we can also construct:

$$(\Lambda a.\Lambda b.\lambda(x : a).\lambda(y : b).\ g\ a\ x\ b\ y) : \forall ab.a \to b \to b$$

Section 4.8 discusses System F in more detail.

defined in Figure 7. For example,

$$pr(\forall a.a \to (\forall b.b \to b)) = \forall ab.a \to b \to b$$

In general, $pr(\sigma)$ takes an arbitrary polytype σ and returns a polytype of the form

$$pr(\sigma) = \forall \overline{a}.\, \sigma_1 \to \ldots \to \sigma_n \to \tau$$

There can be ∀s in the σ_i, but there are no ∀s in the result types of the top-level arrows. Of course, when floating out the ∀s, we must be careful to avoid accidental capture, which is the reason for the side condition in the second rule for $pr()$. (We can always alpha-convert the type to satisfy this condition.) We call it "weak" prenex conversion because it leaves the argument types σ_i unaffected.

To keep the system syntax-directed, we have split the subsumption judgement into two. The main one, $\vdash^{dsk}$, has a single rule that performs deep skolemisation and invokes the auxiliary judgement, $\vdash^{dsk*}$. The latter has the remaining rules for subsumption, unchanged from $\vdash^{ol}$, except that FUN invokes $\vdash^{dsk}$ on the argument types but $\vdash^{dsk*}$ on the result types. To see why the split is necessary, consider trying to check $\vdash^{dsk} \forall a.\mathtt{Int} \rightarrow a \rightarrow a \leq \mathtt{Int} \rightarrow \forall a.a \rightarrow a$. Even though the $\forall a$ on the right is hidden under the arrow, we must still use DEEP-SKOL before SPEC.

The function $pr(\sigma)$ converts σ to weak-prenex form "on the fly". Another workable alternative – indeed one we used in an earlier version of this paper – is to ensure that *all* types are syntactically constrained to be in prenex form, using the following syntax:

$$\begin{array}{rcl} \sigma & ::= & \forall\overline{a}.\rho \\ \rho & ::= & \tau \mid \sigma \rightarrow \rho \end{array}$$

This seems a little less elegant in theory, and is a little less convenient in practice because it is sometimes convenient for the programmer to write non-prenex-form types — the curious reader may examine the type of `everywhere` in Section 6.1 of (Lämmel & Peyton Jones, 2003) for an example. The syntactically-constrained system also seems more fragile if we wanted to move to an impredicative system, because instantiation could yield a syntactically-illegal type.

The deep-skolemisation approach would not work for ML, because in ML the types

$\forall ab.a \to b \to b$ and $\forall a.a \to (\forall b.b \to b)$ are *not* isomorphic: one cannot push foralls around freely because of the value restriction. There are alternative approaches, as discussed by Rémy (Rémy, 2005), but we do not discuss this issue further here.

4.7 Bidirectional type inference

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 following definition[5]:

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

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: it was invented by Pierce and Turner, who called it *local type inference* (Pierce & Turner, 1998). We use the Pierce/Turner formalism in what follows, and return to a discussion of their work in Section 9.4.

4.7.1 Bidirectional inference judgements

Figure 8 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 $\vdash^{poly}$ and $\vdash^{inst}$ are generalised in the same way.

The main idea of the bidirectional typing rules is that a term might be typeable in checking mode when it is not typeable in inference mode; for example the term `(\x -> (x True, x 'a'))` can be checked with type $(\forall a.a \to a) \to (\texttt{Bool}, \texttt{Char})$, but is not typeable in inference mode. However, if we infer the type for a term, we can always check that the term has that type. That is:

$$\text{If} \quad \Gamma \vdash_{\Uparrow} t : \rho \quad \text{then} \quad \Gamma \vdash_{\Downarrow} t : \rho$$

Furthermore, checking mode allows us to impress on a term any type that is more specific than its most general type. In contrast, inference mode may only produce a type that is some substitution of the most general type. For example, if a variable has type $b \to (\forall a.a \to a)$ we can check that it has this type and also that it has types $\texttt{Int} \to (\forall a.a \to a)$ and $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$. On the other hand, of these types, we will only be able to infer $b \to (\forall a.a \to a)$ and $\texttt{Int} \to (\forall a.a \to a)$.

[5] 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.

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

$$\boxed{\Gamma \vdash_\delta t : \rho} \qquad \delta ::= \Uparrow \mid \Downarrow$$

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

$$\frac{\Gamma, (x : \tau) \vdash_\Uparrow t : \rho}{\Gamma \vdash_\Uparrow (\backslash x\,.\,t) : (\tau \to \rho)}\ \text{ABS1} \qquad \frac{\Gamma, (x : \sigma_a) \vdash^{poly}_\Downarrow t : \sigma_r}{\Gamma \vdash_\Downarrow (\backslash x\,.\,t) : (\sigma_a \to \sigma_r)}\ \text{ABS2}$$

$$\frac{\Gamma, (x : \sigma) \vdash_\Uparrow t : \rho}{\Gamma \vdash_\Uparrow (\backslash (x\texttt{::}\sigma)\,.\,t) : (\sigma \to \rho)}\ \text{AABS1} \qquad \frac{\begin{array}{c}\vdash^{dsk} \sigma_a \leq \sigma_x \\ \Gamma, (x : \sigma_x) \vdash^{poly}_\Downarrow t : \sigma_r\end{array}}{\Gamma \vdash_\Downarrow (\backslash (x\texttt{::}\sigma_x)\,.\,t) : (\sigma_a \to \sigma_r)}\ \text{AABS2}$$

$$\frac{\Gamma \vdash_\Uparrow t : (\sigma \to \sigma') \quad \Gamma \vdash^{poly}_\Downarrow u : \sigma \quad \vdash^{inst}_\delta \sigma' \leq \rho}{\Gamma \vdash_\delta t\ u : \rho}\ \text{APP}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{poly}_\Downarrow t : \sigma \\ \vdash^{inst}_\delta \sigma \leq \rho\end{array}}{\Gamma \vdash_\delta \texttt{(}t\texttt{::}\sigma\texttt{)} : \rho}\ \text{ANNOT} \qquad \frac{\begin{array}{c}\Gamma \vdash^{poly}_\Uparrow u : \sigma \\ \Gamma, x : \sigma \vdash_\delta t : \rho\end{array}}{\Gamma \vdash_\delta \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho}\ \text{LET}$$

$$\boxed{\Gamma \vdash_{\delta}^{poly} t : \sigma}$$

$$\frac{\begin{array}{c}\overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma) \\ \Gamma \vdash_{\Uparrow} t : \rho\end{array}}{\Gamma \vdash_{\Uparrow}^{poly} t : \forall \overline{a} . \rho}\ \textsc{gen1} \qquad \frac{\begin{array}{c}\overline{a} \notin \mathit{ftv}(\Gamma) \\ \Gamma \vdash_{\Downarrow} t : \rho \\ pr(\sigma) = \forall \overline{a} . \rho\end{array}}{\Gamma \vdash_{\Downarrow}^{poly} t : \sigma}\ \textsc{gen2}$$

$$\boxed{\vdash_{\delta}^{inst} \sigma \leq \rho}$$

$$\frac{}{\vdash_{\Uparrow}^{inst} \forall \overline{a} . \rho \ \leq\ [\overline{a \mapsto \tau}]\ \rho}\ \textsc{inst1} \qquad \frac{\vdash^{dsk} \sigma \ \leq\ \rho}{\vdash_{\Downarrow}^{inst} \sigma \ \leq\ \rho}\ \textsc{inst2}$$

Fig. 8: Bidirectional version of Odersky-Läufer

Finally, our intention is that any term typable by the uni-directional rules of Figure 6 is also typable in inference mode by Figure 8. That is:

$$\text{If} \quad \Gamma \vdash t : \rho \quad \text{then} \quad \Gamma \vdash_{\Uparrow} t : \rho$$

The reverse is of course false. That is the whole point: we expect that the definition of `foo` above will be typable with the new rules, whereas it is not with the old ones.

4.7.2 Bidirectional inference rules

Many of the rules in Figure 8 are "polymorphic" in the direction δ. For example, the rules INT, VAR, APP, 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 worthwhile.

The rule APP, which deals with function application $(t\ u)$, is of particular interest. Regardless of the direction δ, we first *infer* the type $\sigma \to \sigma'$ for t, and then *check* that u has type σ. 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. We then use $\vdash^{inst}$ to check that σ' and ρ are compatible. Notice that, even in the checking case $\Downarrow$, we ignore the required type ρ when inferring the type for the function t. There is clearly some information loss here: we know the result type, ρ, for t, but we do not know its argument type. The rules provide no way to express this *partial* information about t's type—but see the discussion in Section 9.4.

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 $\sigma_a \rightarrow \sigma_r$, we bind x to the polytype σ_a, *even though x is not explicitly annotated*, before checking that the body has type σ_r. 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:

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

4.7.3 Instantiation and generalisation

The $\vdash_{\delta}^{inst}$ judgement also has separate rules for inference and checking. Rule INST1 deals with the inference case: just as in the old INST rule of Figure 6, we simply instantiate the outer $\forall$'s. The checking case, INST2, is more interesting. Here, we are pushing inward a type ρ, and it meets a variable of known polytype σ. The right thing to do is simply to check that ρ is more polymorphic than σ, using our subsumption judgement $\vdash^{dsk}$. Rule ANNOT and VAR both make use of the $\vdash_{\delta}^{inst}$, just as they did in Figure 6, but ANNOT becomes slightly simpler. In the syntax-directed rules of Figure 6, 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.

The reader may wonder why we do not need deep *instantiation* as well as deep *skolemisation*. In particular, here is an alternative version of rule INST1:

$$\frac{pr(\sigma) = \forall\overline{a}\,.\,\rho}{\vdash_{\Uparrow}^{inst} \sigma \;\leq\; [\overline{a \mapsto \tau}]\,\rho}\ \text{DEEP-INST1}$$

(The prenex-conversion function $pr(\sigma)$, was introduced in Section 4.6.2.) This rule instantiates *all* the top-level $\forall$'s of a type, even if they are hidden under the right-hand end of an arrow. For example, under DEEP-INST1:

$$\vdash_{\Uparrow}^{inst} \forall a.a \to \forall b.b \to b \;\leq\; [\overline{a \mapsto \tau_a, b \mapsto \tau_b}]\; a \to b \to b$$

Adopting this rule would give an interesting invariant, namely that

$$\Gamma \vdash_{\Uparrow} t : \rho \qquad \Rightarrow \qquad \rho \text{ is in weak-prenex form}$$

However, there seems to be no other reason to complicate INST1, so we use the simpler version.

The generalisation judgement $\Gamma \vdash^{poly}_{\delta} t : \sigma$ also has two cases. When we are inferring a polytype (rule GEN1) we need to quantify over all free variables of the inferred ρ type that do not appear in the context Γ, just as before.

On the other hand, when we check that a polytype can be assigned to a term (rule GEN2), we simply skolemise the quantified variables, checking they do not appear free in the environment Γ. The situation is very similar to that of DEEP-SKOL in Figure 7, so GEN2 must perform weak prenex conversion on the expected type σ, to bring all its quantifiers to the top. If it fails to do so, the following program would not typecheck:

$$\texttt{f} : (\forall ab.\texttt{Int} \to a \to b \to b) \vdash^{poly}_{\Downarrow} \texttt{f 3} : \texttt{Bool} \to \forall c.c \to c$$

The problem is that `f`'s type is instantiated by VAR before rule APP invokes $\vdash^{inst}_{\Downarrow}$ to marry up the result type with the type of `(f 3)`, and hence before the $\forall c.c \to c$ is skolemised.

Once we use GEN2, however, the reader may verify that $\Gamma \vdash_{\Downarrow} t : \rho$ is invoked only when ρ is in weak-prenex form. However, for generality we prefer to define $\vdash_{\Downarrow}$ over arbitrary ρ-types. For example, this generality allows us to state, without side conditions, that if $\Gamma \vdash_{\Uparrow} t : \rho$ then $\Gamma \vdash_{\Downarrow} t : \rho$.

$$
\begin{array}{lcll}
e, f & ::= & i & \text{Literal} \\
 & | & x & \text{Variable} \\
 & | & \Lambda\alpha.t & \text{Type abstraction} \\
 & | & \lambda(x : \sigma).t & \text{Value abstraction} \\
 & | & e\ \sigma & \text{Type application} \\
 & | & f\ e & \text{Value application} \\
 & | & \texttt{let}\ x : \sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \text{Local binding}
\end{array}
$$

Fig. 9: Syntax of System F

4.7.4 Summary

In summary, the bidirectional 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.

4.8 Type-directed translation

A type system tells whether a term is well-typed. In some compilers, the type inference engine also performs a closely-related task, that of performing a *type-directed translation* from the *implicitly-typed source language* into an *explicitly-typed target language*. The target language is "explicitly typed" because the term is decorated with enough type information to make type-checking very simple. The source language is "implicitly typed" because as much type clutter as possible is omitted. The business of the type inference engine is to fill in the missing type information.

One very popular target language is System F (Girard, 1990), an extremely expressive, strongly-typed lambda calculus. Figure 9 gives the syntax of the variant of System F that we will use here. It differs from the source language in the following ways:

- The binding occurrence of every variable is annotated with its type.
- An explicit *type application* $(e\,\sigma)$ specifies the types that instantiate a polymorphic function f.
- An explicit *type abstraction* $(\Lambda a.e)$ specifies where and how generalisation takes place.

For example, consider:

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

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

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

With explicit type abstractions and applications, `concat` would look like this:

$$\texttt{concat}:\forall a.\texttt{[[}a\texttt{]]} \rightarrow \texttt{[}a\texttt{]} = \Lambda a.\lambda(xs:\texttt{[[}a\texttt{]]}).\texttt{foldr}\,\texttt{[}a\texttt{]}\,\texttt{[}a\texttt{]}\,((\texttt{++})\,a)\,(\texttt{Nil}\,a)\,xs$$

The "Λa" 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.

We cannot give a full introduction to System F here, and readers unfamiliar with System F may safely skip this section. However, the System F translation is an extremely useful tool. On the theory side, we use it to prove that our type system is sound, in Section 4.9.3. In practical terms, the explicit type information produced by the type-directed translation is useful to guide subsequent transformations and optimisations. Furthermore, type-checking the System F program at a later stage (which is very easy to do) gives a very strong consistency check that the intermediate stages have not performed an invalid transformation (Morrisett, 1995; Tarditi *etal.*, 1996; Shao, 1997; PeytonJones & Santos, 1998). In the rest of this section we show how to specify the translation into System F.

4.8.1 Translating terms

The term "type-directed" translation comes from the fact that the translation is specified in the type rules themselves. For example, the main judgement for our bidirectional system becomes

$$\Gamma \vdash_{\delta} t : \rho \mapsto e$$

meaning that t has type ρ, and translates to the System-F term e. Furthermore the term e will have type ρ in System F's type system; we write $\Gamma \vdash^{F} e : \rho$. (We do not give the type system for System F here because it is so standard (Pierce, 2002). The interested reader can find it in the Technical Appendix (Vytiniotis *etal.*, 2005).)

The translated, System F terms have explicit type annotations on binders. For example, rule ABS1 from Figure 8 becomes

$$\frac{\Gamma, (x : \tau) \vdash_{\Uparrow} t : \rho \mapsto e}{\Gamma \vdash_{\Uparrow} (\backslash x \,.\, t) : (\tau \to \rho) \mapsto (\lambda(x : \tau).e)} \textsc{abs}1$$

The source program did not have an annotation on x, but the translated System F program does have one.

Many of the other rules in Figure 8 can be modified in a similar routine way and, for completeness, Figure 10 shows the result. In effect, the translated program encodes the exact shape of the derivation tree, and therefore amounts to a proof that the original program is indeed well typed.

$$\boxed{\Gamma \vdash_{\delta} t : \rho \mapsto e}$$

$$\frac{}{\Gamma \vdash_{\delta} i : \mathtt{Int} \mapsto i}\ \textsc{Int} \qquad \frac{\vdash^{inst}_{\delta} \sigma \leq \rho \mapsto f}{\Gamma, (x : \sigma) \vdash_{\delta} x : \rho \mapsto f\ x}\ \textsc{Var}$$

$$\frac{\Gamma, (x : \tau) \vdash_{\Uparrow} t : \rho \mapsto e}{\Gamma \vdash_{\Uparrow} (\backslash x\,.\,t) : (\tau \to \rho) \mapsto \lambda(x{:}\tau).e}\ \textsc{Abs1}$$

$$\frac{\Gamma, (x : \sigma_a) \vdash^{poly}_{\Downarrow} t : \sigma_r \mapsto e}{\Gamma \vdash_{\Downarrow} (\backslash x\,.\,t) : (\sigma_a \to \sigma_r) \mapsto (\lambda x{:}\sigma_a).e}\ \textsc{Abs2}$$

$$\frac{\Gamma, (x : \sigma) \vdash_{\Uparrow} t : \rho \mapsto e}{\Gamma \vdash_{\Uparrow} (\backslash (x{::}\sigma)\,.\,t) : (\sigma \to \rho) \mapsto \lambda(x{:}\sigma).e}\ \textsc{AAbs1}$$

$$\frac{\begin{array}{c}\vdash^{dsk} \sigma_a \leq \sigma_x \mapsto f \\ \Gamma, (x : \sigma_x) \vdash^{poly}_{\Downarrow} t : \sigma_r \mapsto e\end{array}}{\Gamma \vdash_{\Downarrow} (\backslash (x{::}\sigma_x)\,.\,t) : (\sigma_a \to \sigma_r) \mapsto \lambda(x{:}\sigma_a).[x \mapsto (f\ x)]e}\ \textsc{AAbs2}$$

$$\frac{\Gamma \vdash_{\Uparrow} t : (\sigma \to \sigma') \mapsto e_1 \quad \Gamma \vdash^{poly}_{\Downarrow} u : \sigma \mapsto e_2 \quad \vdash^{inst}_{\delta} \sigma' \leq \rho \mapsto f}{\Gamma \vdash_{\delta} t\ u : \rho \mapsto f\ (e_1\ e_2)}\ \textsc{App}$$

$$\frac{\Gamma \vdash^{poly}_{\Downarrow} t : \sigma \mapsto e \qquad \vdash^{inst}_{\delta} \sigma \leq \rho \mapsto f}{\Gamma \vdash_{\delta} (t :: \sigma) : \rho \mapsto f\ e}\ \textsc{annot}$$

$$\frac{\Gamma \vdash^{poly}_{\Uparrow} u : \sigma \mapsto e_1 \qquad \Gamma, x : \sigma \vdash_{\delta} t : \rho \mapsto e_2}{\Gamma \vdash_{\delta} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho \mapsto \texttt{let}\ x : \sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2}\ \textsc{let}$$

$$\boxed{\Gamma \vdash^{poly}_{\delta} t : \sigma \mapsto e}$$

$$\frac{\overline{a} = ftv(\rho) - ftv(\Gamma) \qquad \Gamma \vdash_{\Uparrow} t : \rho \mapsto e}{\Gamma \vdash^{poly}_{\Uparrow} t : \forall \overline{a}.\rho \mapsto \Lambda \overline{a}.e}\ \textsc{gen1}$$

$$\frac{pr(\sigma) = \forall \overline{a}.\rho \mapsto f \qquad \overline{a} \notin ftv(\Gamma) \quad \Gamma \vdash_{\Downarrow} t : \rho \mapsto e}{\Gamma \vdash^{poly}_{\Downarrow} t : \sigma \mapsto f\ (\Lambda \overline{a}.e)}\ \textsc{gen2}$$

$$\boxed{\vdash^{inst}_{\delta} \sigma \leq \rho \mapsto f}$$

$$\frac{}{\vdash^{inst}_{\Uparrow} \forall \overline{a}.\rho\ \leq\ [\overline{a \mapsto \tau}]\ \rho \mapsto \lambda(x{:}\forall \overline{a}.\rho).x\ \overline{\tau}}\ \textsc{inst1}$$

$$\frac{\vdash^{dsk} \sigma\ \leq\ \rho \mapsto f}{\vdash^{inst}_{\Downarrow} \sigma\ \leq\ \rho \mapsto f}\ \textsc{inst2}$$

Fig. 10: Bidirectional higher-rank type system with translation

$$\boxed{pr(\sigma) = \forall\overline{a}.\rho \mapsto f}$$

$$\frac{pr(\rho_1) = \forall\overline{b}.\rho_2 \mapsto f \quad \overline{a} \notin \overline{b}}{pr(\forall\overline{a}.\rho_1) = \forall\overline{a}\overline{b}.\rho_2 \mapsto \lambda(x{:}\forall\overline{a}\overline{b}.\rho_2).\Lambda\overline{a}.f\ (x\ \overline{a})}\ \text{PRPOLY}$$

$$\frac{pr(\sigma_2) = \forall\overline{a}.\rho_2 \mapsto f \quad \overline{a} \notin ftv(\sigma_1)}{pr(\sigma_1 \to \sigma_2) = \forall\overline{a}.\sigma_1 \to \rho_2 \mapsto \lambda(x{:}\forall\overline{a}.\sigma_1 \to \rho_2).\lambda(y{:}\sigma_1).f\ (\Lambda\overline{a}.x\ \overline{a}\ y)}\ \text{PRFUN}$$

$$\frac{}{pr(\tau) = \tau \mapsto \lambda(x{:}\tau).x}\ \text{PRMONO}$$

$$\boxed{\vdash^{dsk} \sigma \leq \sigma' \mapsto f}$$

$$\frac{\begin{array}{c} pr(\sigma_2) = \forall\overline{a}.\rho \mapsto f_1 \\ \overline{a} \notin ftv(\sigma_1) \qquad \vdash^{dsk*} \sigma_1 \leq \rho \mapsto f_2 \end{array}}{\vdash^{dsk} \sigma_1 \leq \sigma_2 \mapsto (\lambda x{:}\sigma_1).f_1\ (\Lambda\overline{a}.f_2\ x)}\ \text{DEEP-SKOL}$$

$$\boxed{\vdash^{dsk*} \sigma \leq \rho \mapsto f}$$

$$\frac{\vdash^{dsk*} [\overline{a \mapsto \tau}]\rho_1 \leq \rho_2 \mapsto f}{\vdash^{dsk*} \forall\overline{a}.\rho_1 \leq \rho_2 \mapsto \lambda(x\!:\!\forall\overline{a}.\rho).f\ (x\ \overline{\tau})}\ \text{SPEC}$$

$$\frac{\vdash^{dsk} \sigma_3 \leq \sigma_1 \mapsto f_1 \qquad \vdash^{dsk*} \sigma_2 \leq \sigma_4 \mapsto f_2}{\vdash^{dsk*} (\sigma_1 \to \sigma_2) \leq (\sigma_3 \to \sigma_4) \mapsto \lambda(x\!:\!\sigma_1 \to \sigma_2).\lambda(y\!:\!\sigma_3).f_2\ (x\ (f_1\ y))}\ \text{FUN}$$

$$\frac{}{\vdash^{dsk*} \tau \leq \tau \mapsto \lambda(x\!:\!\tau).x}\ \text{MONO}$$

Fig. 11: Creating coercion terms

4.8.2 Instantiation, generalisation, and subsumption

The translation of terms is entirely standard, but matters become more interesting when we consider instantiation and generalisation. Consider rule VAR from Figure 8:

$$\frac{\vdash_{\delta}^{inst} \sigma \leq \rho}{\Gamma, (x : \sigma) \vdash_{\delta} x : \rho}\ \text{VAR}$$

What should x translate to? It cannot translate to simply x, because x has type σ, not ρ! After a little thought we see that the $\vdash^{inst}$ judgement should return a coercion *function* of type $\sigma \to \rho$, which can be thought of as concrete—indeed, executable—evidence for the claim that $\sigma \leq \rho$. Then we can add translation to the

VAR rule as follows:

$$\frac{\vdash^{inst}_{\delta} \sigma \leq \rho \mapsto f}{\Gamma, (x : \sigma) \vdash_{\delta} x : \rho \mapsto f\ x}\ \textsc{var}$$

Figure 10 shows the rules for the $\vdash^{inst}$ judgement:

$$\frac{}{\vdash^{inst}_{\Uparrow} \forall \overline{a} \,.\, \rho \ \leq\ [\overline{a \mapsto \tau}]\, \rho \mapsto \lambda(x : \forall \overline{a} \,.\, \rho).x\ \overline{\tau}}\ \textsc{inst1} \qquad \frac{\vdash^{dsk} \sigma \ \leq\ \rho \mapsto e}{\vdash^{inst}_{\Downarrow} \sigma \ \leq\ \rho \mapsto e}\ \textsc{inst2}$$

The inference case, rule INST1, uses a System F type application $(x\ \overline{\tau})$ to record the types at which x is instantiated. For the checking case, rule INST2 defers to $\vdash^{dsk}$, which also returns a coercion function.

So much for instantiation. Dually, generalisation is expressed by System-F type abstraction, as we can see in the rules for $\vdash^{poly}$ in Figure 10:

$$\frac{\begin{array}{c}\overline{a} = ftv(\rho) - ftv(\Gamma) \\ \Gamma \vdash_{\Uparrow} t : \rho \mapsto e\end{array}}{\Gamma \vdash^{poly}_{\Uparrow} t : \forall \overline{a} \,.\, \rho \mapsto \Lambda \overline{a}.e}\ \textsc{gen1} \qquad \frac{\begin{array}{c}\overline{a} \notin ftv(\Gamma) \qquad pr(\sigma) = \forall \overline{a} \,.\, \rho \mapsto f \\ \Gamma \vdash_{\Downarrow} t : \rho \mapsto e\end{array}}{\Gamma \vdash^{poly}_{\Downarrow} t : \sigma \mapsto f\ (\Lambda \overline{a}.e)}\ \textsc{gen2}$$

Rule GEN1 directly introduces a type abstraction, while but GEN2 needs a coercion function, just like VAR, to account for the prenex-form conversion. The rules for prenex-form conversion, and for $\vdash^{dsk}$, are given in in Figure 11.

When reading the rules for type-directed translation, the key invariants to bear in mind are these:

If this holds	then so does this
$\Gamma \vdash_\delta t : \rho \mapsto e$	$\Gamma \vdash^F e : \rho$
$\vdash_\delta^{inst} \sigma \leq \rho \mapsto e$	$\vdash^F e : \sigma \to \rho$
$\vdash^{dsk} \sigma_1 \leq \sigma_2 \mapsto e$	$\vdash^F e : \sigma_1 \to \sigma_2$
$pr(\sigma_1) = \sigma_2 \mapsto e$	$\vdash^F e : \sigma_2 \to \sigma_1$

This type-directed translation also provides a semantics for our language. To determine the meaning of a term, translate it to System F and evaluate the result. Although this semantics is defined by translation, it is fairly simple and what we might expect. If we erase types in the source and target languages it is easy to verify that, except for the insertion of coercions, the translation is the identity translation. Furthermore, the coercions themselves only produce terms that, after type erasure, are eta-expansions of the identity function.

4.9 Metatheory of higher-rank type systems

In this section we give formal statements of the most important properties of the type systems and subsumption relations presented so far. Again, the Technical Appendix (Vytiniotis *etal.*, 2005) contains the proofs of the theorems in this section.

We begin with properties of the various subsumption judgements in Section 4.9.1.

In Section 4.9.2 we describe the precise connection between the type systems of this paper: the original Damas-Milner system, the non syntax-directed, the syntax-directed, and the bidirectional higher-rank type system. Section 4.9.3 gives the most important properties of the bidirectional system.

4.9.1 Properties of the subsumption judgements

We have now defined three different subsumption relations:

- $\vdash^{sh} \sigma_1 \leq \sigma_2$ is the Damas-Milner shallow-subsumption relation (Figure 4), which we now extend to higher-rank types. The only difference is that the rule MONO is replaced with

$$\frac{}{\vdash^{sh} \rho \leq \rho}$$

This way shallow-subsumption naturally applies to the type syntax defined in Figure 5.

- $\vdash^{ol} \sigma_1 \leq \sigma_2$ is the Odersky-Läufer subsumption, defined in Figure 5.
- $\vdash^{dsk} \sigma_1 \leq \sigma_2$ refers to subsumption with deep skolemisation, defined in Figure 7.

These three relations are connected in the following way: Deep skolemisation subsumption relates strictly more types than the Odersky-Läufer relation, which in turn relates strictly more types than the Damas-Milner relation.

Theorem 4.1 *If* $\vdash^{sh} \sigma_1 \leq \sigma_2$ *then* $\vdash^{ol} \sigma_1 \leq \sigma_2$. *If* $\vdash^{ol} \sigma_1 \leq \sigma_2$ *then* $\vdash^{dsk} \sigma_1 \leq \sigma_2$.

The following theorem captures the essence of $\vdash^{dsk}$; any type is equivalent to its prenex form.

Theorem 4.2 $\vdash^{dsk} \sigma \leq pr(\sigma)$ *and* $\vdash^{dsk} pr(\sigma) \leq \sigma$.

In contrast notice that only $\vdash^{ol} \sigma \leq pr(\sigma)$.

All three relations are reflexive and transitive. However, only deep skolemisation subsumption enjoys a distributivity property, that lets us distribute type quantification among the components of an arrow type:

Theorem 4.3 (Distributivity) $\vdash^{dsk} \forall a\,.\,\sigma_1 \rightarrow \sigma_2 \leq (\forall a\,.\,\sigma_1) \rightarrow \forall a\,.\,\sigma_2$.

This theorem is essential for showing that the coercion functions generated by our $\vdash^{dsk}$ derivations correspond exactly to the System F functions that, after erasure of types, are $\beta\eta$-convertible to the identity. We defer further discussion for Section 9.5.

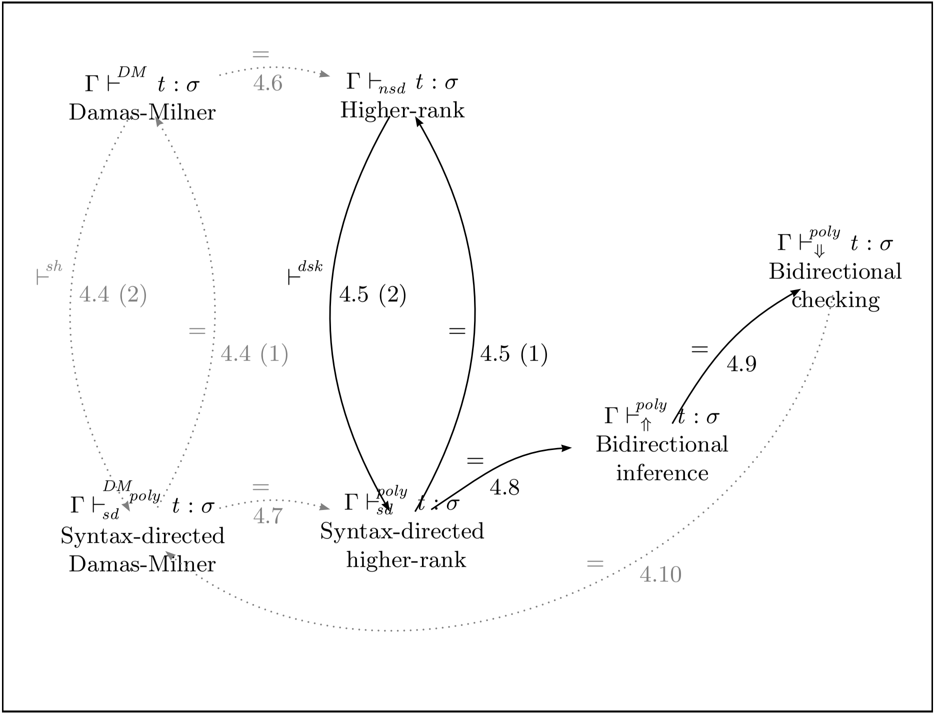

Fig. 12: Relations between type systems in this paper

4.9.2 Connections between the type systems

At this point we have discussed the five type systems that appeared in the road map in Figure 1. We started with the Damas-Milner type system, and described its declarative and syntax-directed forms. We then presented an extension that supports higher-rank types, the Odersky-Läufer type system, and developed a syntax directed version. Finally, we introduced the bidirectional type system, an extension of our syntax-directed version of the Odersky-Läufer system.

This section states the formal connections between all of these systems. The results of this section are summarised in Figure 12. In some of these results, it matters whether we are talking about Damas-Milner types and terms, or higher-rank types and terms. In the figure, dashed lines correspond to connections where we assume that the types appearing in the judgements are only Damas-Milner types and that the terms contain no type annotations.

Some of the connections in this figure have already been shown. In particular the relation between the syntax-directed and the non syntax-directed Damas-Milner type system is captured by the theorem below.

Theorem 4.4 ((Milner, 1978), (Damas & Milner, 1982)) *Suppose that t contains no type annotations and the context Γ contains only Damas-Milner types.*

1. *If* $\Gamma \vdash_{sd}^{DM_{poly}} t : \sigma$ *then* $\Gamma \vdash^{DM} t : \sigma$.
2. *If* $\Gamma \vdash^{DM} t : \sigma$ *then there is a* σ' *such that* $\Gamma \vdash_{sd}^{DM_{poly}} t : \sigma'$ *and* $\vdash^{sh} \sigma' \leq \sigma$.

We can show an analogous result for the higher-rank systems. We began our discussion of higher-rank polymorphism with the Odersky-Laufer type system (Section 4.5), and developed a syntax-directed version of it (Section 4.6). Recall that with the Odersky-Läufer definition of subsumption, but without eager generalisation, the two type systems did not agree. There were some programs that type-checked in the original version, but did not typecheck in the syntax-directed version (Section 4.6.1). By changing the subsumption relation in the syntax-directed version to deep skolemisation, we can make it accept all of the programs accepted by the original type system.

However, it turns out that the two systems are still not equivalent: the syntax-directed system, using deep skolemisation, accepts some programs that are rejected by the original typing rules! For example, the derivation

$$x : \forall b.\mathtt{Int} \to b \vdash (x :: \mathtt{Int} \to \forall b.b) : \mathtt{Int} \to \forall b.b$$

is valid in the syntax-directed version. But, because it uses deep skolemisation in checking the type annotation, there is no analogue in the original system. Fortunately, if we replace subsumption in the orginal system with deep skolemisation, the two type systems do agree.

In what follows, let $\vdash_{nsd}$ refer to the typing rules of Figure 5 where the $\vdash^{ol}$ relation has been replaced by the $\vdash^{dsk}$ relation. Also let $\Gamma \vdash_{sd} t : \rho$ refer to the syntax-directed rules in Figure 6.

Theorem 4.5 (Agreement of $\vdash_{nsd}$ and $\vdash_{sd}$)

1. *If* $\Gamma \vdash^{poly}_{sd} t : \sigma$ *then* $\Gamma \vdash_{nsd} t : \sigma$.
2. *If* $\Gamma \vdash_{nsd} t : \sigma$ *then there is a* σ' *such that* $\Gamma \vdash^{poly}_{sd} t : \sigma'$ *and* $\vdash^{dsk} \sigma' \leq \sigma$.

The first two clauses of this theorem say that if a term can be typed by the syntax-directed system, then the non-syntax-directed system can also type it, and with the same type. The exact converse is not true; for example, in the non-syntax-directed system we have $\vdash_{nsd}$ `\x.\y.y` $: \forall a.a \rightarrow \forall b.b \rightarrow b$, but this type is not derivable in the syntax-directed system. Instead we have $\vdash^{poly}_{sd}$ `\x.\y.y` $: \forall ab.a \rightarrow b \rightarrow b$. In general, as clause (3) says, if a term in typeable in the non-syntax-directed system, then it is also typeable in the syntax-directed system, but perhaps with a different type σ' that is at least as polymorphic as the original one.

Next, we show that the Odersky-Läufer system is an extension of the Damas-Milner system. Any term that type checks using the Damas-Milner rules, type checks with the same type using the Odersky-Läufer rules. Let $\Gamma \vdash^{DM} t : \sigma$ refer to the Damas-Milner judgement, defined in Figure 3.

Theorem 4.6 (Odersky-Läufer extends Damas-Milner) *Suppose t contains no type annotations and the context Γ only contains Damas-Milner types. If $\Gamma \vdash^{DM} t : \sigma$ then $\Gamma \vdash_{nsd} t : \sigma$.*

Likewise, our version of the Odersky-Läufer syntax-directed system extends the Damas-Milner syntax-directed system.

Theorem 4.7 (Syntax-directed extension) *Suppose t contains no type annotations and the context Γ only contains Damas-Milner types. If $\Gamma \vdash_{sd}^{DM\,poly} t : \sigma$ then $\Gamma \vdash_{sd}^{poly} t : \sigma$.*

Furthermore, the bidirectional system extends the syntax-directed system. Anything that can be inferred by Figure 6 can be inferred in the bidirectional system. (The converse is not true, of course. The point of the bidirectional system is to typecheck more terms.)

Theorem 4.8 (Bidirectional inference extends syntax-directed system)

1. *If* $\Gamma \vdash_{sd} t : \rho$ *then* $\Gamma \vdash_{\Uparrow} t : \rho$.
2. *If* $\Gamma \vdash_{sd}^{poly} t : \sigma$ *then* $\Gamma \vdash_{\Uparrow}^{poly} t : \sigma$.

Checking mode extends inference mode for the bidirectional system. If we can infer a type for a term, we should be able to check that this type can be assigned to the term.

Theorem 4.9 (Bidirectional checking extends inference)

1. *If* $\Gamma \vdash_{\Uparrow} t : \rho$ *then* $\Gamma \vdash_{\Downarrow} t : \rho$.
2. *If* $\Gamma \vdash_{\Uparrow}^{poly} t : \sigma$ *then* $\Gamma \vdash_{\Downarrow}^{poly} t : \sigma$.

Finally, the bidirectional system is *conservative* over the Damas-Milner type system. If a term typechecks in the bidirectional system without any higher-rank annotations, and with a monotype, then the term type checks in the syntax-directed Damas-Milner system, with the same type. Let $\Gamma \vdash_{sd}^{DM} t : \tau$ refer to the judgement defined in Figure 4.

Theorem 4.10 (Bidirectional conservative over Damas-Milner) *Suppose* t *contains no type annotations, and* Γ *contains only Damas-Milner types. If* $\Gamma \vdash_{\delta} t : \tau$

then $\Gamma \vdash^{DM}_{sd} t : \tau$.

4.9.3 Properties of the bidirectional type system

The bidirectional type system, in Figure 8, is a novel contribution of this paper. Any type system must enjoy the self-consistency properties of *type safety* and *principal types*. In this section we describe these properties in more detail.

The *type safety* theorem asserts that the type system rules out ill-behaved programs. In other words, the evaluation of any well-typed program will produce a value (or possibly diverge, if the language contains diverging terms). This theorem is proven with respect to a semantics—rules that describe how programs produce values. In Section 4.8 we defined the semantics of the bidirectional type system by translation to System F. To evaluate an expression, we translate it to System F and evaluate the System F term.

System F is already known to be type safe. Therefore, to show type safety for the bidirectional type system, all we must do is show that the translation to System F produces well-typed terms. That way we know that all terms accepted by the bidirectional system will evaluate without error.

In other words:

Theorem 4.11 (Soundness of bidirectional system)

1. *If* $\Gamma \vdash_{\delta} t : \rho \mapsto e$ *then* $\Gamma \vdash^{F} e : \rho$.
2. *If* $\Gamma \vdash_{\delta}^{poly} t : \sigma \mapsto e$ *then* $\Gamma \vdash^{F} e : \sigma$.

The proof of this theorem relies on a number of theorems that say that the coercions produced by the subsumption judgment are well typed. The proof of these theorems is by a straightforward induction on the appropriate judgment.

Theorem 4.12 (Coercion typing)

1. *If* $pr(\sigma) = \forall\overline{a}.\rho \mapsto e$ *then* $\vdash^{F} e : (\forall\overline{a}.\rho) \rightarrow \sigma$.
2. *If* $\vdash^{dsk} \sigma \leq \sigma' \mapsto e$ *then* $\vdash^{F} e : \sigma \rightarrow \sigma'$.
3. *If* $\vdash^{dsk*} \sigma \leq \sigma' \mapsto e$ *then* $\vdash^{F} e : \sigma \rightarrow \sigma'$.
4. *If* $\vdash_{\delta}^{inst} \sigma \leq \rho \mapsto e$ *then* $\vdash^{F} e : \sigma \rightarrow \rho$.

The bidirectional type system also has the *principal types* property. In other words, for all terms typable in a particular context, there is some "best" type for that term:

Theorem 4.13 (Principal Types for bidirectional system) *If there exists some*

σ' such that $\Gamma \vdash^{poly}_{\Uparrow} t : \sigma'$, then there exists σ (the principal type of t in context Γ) such that

1. $\Gamma \vdash^{poly}_{\Uparrow} t : \sigma$
2. *For all σ'', if $\Gamma \vdash^{poly}_{\Uparrow} t : \sigma''$, then $\vdash^{sh} \sigma \leq \sigma''$.*

The principal types theorem is very important in practice. It means that an implementation can infer a single, principal type for each `let`-bound variable, that will "work" regardless of the contexts in which the variable is subsequently used.

Notice that, in the second clause of the theorem, all types that are inferred for a

given term are related by the Damas-Milner definition of subsumption, $\vdash^{sh}$. The theorem holds *a fortiori* if $\vdash^{sh}$ is replaced by $\vdash^{dsk}$.

We prove this theorem in the same way that Damas and Milner showed that their type system has principal types: by developing and algorithm that unambiguously assigns types to terms and showing that this algorithm is sound and complete with respect to the rules. The formalisation of the algorithm can be found in the Technical Appendix (Vytiniotis *etal.*, 2005).

The principal-types theorem above only deals with *inference* mode. An analogous version is not needed for *checking* mode because we know exactly what type the term should have—there is no ambiguity. And in fact, such a theorem is not true. For example, the term `(\g.(g 3, g True))` typechecks in the empty context with types $(\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})$ and $(\forall a\,.\,a \to a) \to (\texttt{Int}, \texttt{Bool})$, but there is no type that we can assign to the term that is more general than both of these types.

Even though there are no "most general" types that terms may be assigned in checking mode, checking mode still statisfies properties that make type checking predictable for programmers. For example, it is the case that if we can check a term, then we can always check it at a more specific type. The following theorem formalises this, and other, claims:

Theorem 4.14

1. *If* $\Gamma \vdash^{poly}_{\Downarrow} t : \sigma$ *and* $\vdash^{dsk} \sigma \leq \sigma'$ *then* $\Gamma \vdash^{poly}_{\Downarrow} t : \sigma'$.
2. *If* $\Gamma \vdash_{\Downarrow} t : \rho_1$ *and* $\vdash^{dsk} \rho_1 \leq \rho_2$ *and* ρ_1 *and* ρ_2 *are in weak-prenex form, then* $\Gamma \vdash_{\Downarrow} t : \rho_2$.
3. *If* $\Gamma' \vdash^{poly}_{\Downarrow} t : \sigma$ *and* $\vdash^{dsk} \Gamma \leq \Gamma'$ *then* $\Gamma \vdash^{poly}_{\Downarrow} t : \sigma$.
4. *If* $\Gamma' \vdash_{\Downarrow} t : \rho$ *and* $\vdash^{dsk} \Gamma \leq \Gamma'$ *and* ρ *is in weak-prenex form then* $\Gamma \vdash_{\Downarrow} t : \rho$.

The first clause is self explanatory, but the second might seem a little surprising:

why must ρ_1 and ρ_2 be in weak-prenex form? Here is a counter-example when they are not. Suppose $\sigma_1 = \forall a\,.\,a \to \forall b\,.\,b \to \forall c\,.\,b \to c$, $\sigma_2 = Int \to \forall c\,.\,Int \to c$, and $\sigma_3 = \forall abc\,.\,a \to b \to b \to c$. Then it is derivable that $\vdash_{\Downarrow} (\backslash x\,.\,x\ 3) : (\sigma_1 \to \sigma_2)$ but it is not derivable that $\vdash_{\Downarrow} (\backslash x\,.\,x\ 3) : (\sigma_3 \to \sigma_2)$, although $\vdash^{dsk} \sigma_1 \to \sigma_2 \leq \sigma_3 \to \sigma_2$. However, because GEN2 converts the checked type into that form before continuing, any pair of related types may be used for the $\vdash^{poly}_{\Downarrow}$ judgement, so the first clause needs no side condition.

Just as the first two clauses say that we can make the result type *less* polymorphic; dually, the third and fourth clauses allow us to make the context *more* polymorphic. The notation $\vdash^{dsk} \Gamma \leq \Gamma'$ means that the context Γ is point-wise more general (using the relation $\vdash^{dsk}$) than the context Γ'.

We conclude our discussion of the properties of the bidirectional type system by observing that it *lacks* some properties of the traditional Damas-Milner system.

In particular, in Damas-Milner one can always name a sub-expression using `let`, without affecting typeability:

$$\Gamma \vdash t_1[t_2] : \tau \qquad \textit{implies} \qquad \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ t_2\ \texttt{in}\ t_1[x]$$

(where x does not appear in $t_1[]$. In the bidirectional system, however, the context of t_2 may provide type information that makes it typeable, so the `let` form might fail. To make it succeed, one would need to add a type signature for x.

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 Damas-Milner 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. Admittedly, the Damas-Milner inference engine is deliberately crafted so that it can readily be modified for higher-rank types—but no aspect of the former is there solely to prepare for the latter.

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 13 is the representation for terms, whose syntax was given in Figure 2. The data type of types, also given in Figure 13, deserves a little more explanation. We use a single data type `Type` to represent σ-types, ρ-types, and τ-types, and declare type synonyms `Sigma`, `Rho`, and `Tau` as unchecked documentation about which particular flavour of type is expected at any particular place in the code.[6] The data type `Type` has constructors for quantification (`ForAll`), functions (`Fun`), constants (`TyCon`). We maintain the invariant that the `Type` immediately inside a `ForAll` is not itself a `ForAll`; i.e. that it is a `Rho`.

More interestingly, it has two different constructors for type variables, because the implementation distinguishes two kinds of type variable. Consider the syntax of Damas-Milner types:

$$\begin{array}{rcl} \sigma & ::= & \forall \overline{a}.\tau \\ \tau & ::= & \texttt{Int} \mid \tau_1 \rightarrow \tau_2 \mid a \end{array}$$

[6] This tension between static and dynamic checks is a common one when writing software. The reader is invited to try stratifying the implementation, and compare the result with the version we present here.

```haskell
---------------- 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

type Name = String

---------------- Types -------------------
type Sigma = Type
type Rho   = Type        -- No top-level ForAll
type Tau   = Type        -- No ForAlls anywhere

data Type = ForAll [TyVar] Rho    -- Forall type
          | Fun    Type Type      -- Function type
          | TyCon  TyCon          -- Type constants
          | TyVar  TyVar          -- Always bound by a ForAll
          | MetaTv MetaTv         -- A meta type variable

data TyVar
  = BoundTv String                  -- A type variable bound by a ForAll

  | SkolemTv String Uniq            -- A skolem constant; the String is
                                    -- just to improve error messages
```

```
data TyCon = IntT | BoolT

(-->) :: Type -> Type -> Type    -- Build a function type
arg --> res = Fun arg res

intType :: Tau
intType = TyCon IntT
```

Fig. 13: The `Term` and `Type` data types

The type variable "a" is part of the concrete syntax of types: $a \to$ `Int` and $\forall a.a \to a$ are both legal types. On the other hand, "τ" and "σ" are *meta-variables*, part of the language that we use to discuss types, but not part of the language of syntax of types themselves. For example, $\tau \to \tau$ is not a legal type.

The typing judgements for a type system (Figure 3, for example) uses both kinds of variables. It uses "a" to mean "a type variable", and "τ" to mean "some type obeying the syntax of τ-types". This distinction is reflected in two distinct data types of the implementation:

A concrete type variable, written a, b etc., has type `TyVar` and occurs with constructor `TyVar` in a `Type`.

```
data TyVar = BoundTv String | SkolemTv String Uniq
```

```
type Uniq  = Int
```


There are two kinds of concrete type variables, corresponding to the two constructors of `TyVar`.

- A *bound type variable*, whose constructor is `BoundTv`, is always bound by an enclosing `ForAll`; it may appear in (the type annotations of) a source program; and it is represented by a simple `String`. No well-formed `Type` ever has a free `BoundTv`.
- A *skolem constant*, whose constructor is `SkolemTv`, stands for a constant, but unknown type. It is never bound by a `ForAll`, and it can be free in a `Type`. It is represented by a `Uniq`, a unique integer that distinguishes it from others; the `String` is just for documentation.

A meta type variable, written τ_1, τ_2 etc.[7], is simply a temporary place-holder for an as-yet-unknown monotype. It has type `MetaTv`, and occurs with constructor `MetaTv` in a `Type`.

```
data MetaTv = Meta Uniq TyRef
```

It is never quantified by a `ForAll` ($\forall\tau.\tau$ would not make sense!); and it is created only by the type inference engine itself. Again we use a `Uniq` to give its identity; we will discuss the `TyRef` part later, in Section 5.7.

Although we give the representation of types here, for the sake of concreteness, much of the type inference engine is independent of the details of the representation. The infix function `(-->)` helps to maintain this abstraction, by allowing the inference engine to construct a function type without knowing how it is represented internally. Similarly `intType` is the `Type` 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 14. The monad serves the following roles:

- It supports exceptions, when type inference fails (`check`).
- It carries the environment Γ (`lookupVar` and `extendVarEnv`).
- It allocates fresh meta type variables (`newMetaTv`).
- It maintains a global, ever-growing substitution that supports unification (`unify`).

The function `check` (Figure 14) is typically used in a context like this

```
do { ...
```

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

[7] It turns out that the implementation does not require a representation for the meta-variable σ.

```
        -- 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
getEnvTypes  :: Tc [Sigma]          -- Get all types in the envt

    -- Instantiation, skolemisation, quantification
instantiate :: Sigma -> Tc Rho
skolemise   :: Sigma -> Tc ([TyVar], Rho)
quantify    :: [MetaTv] -> Rho -> Tc Sigma

    -- Unification and fresh type variables
newMetaTyVar   :: Tc Tau   -- Make (MetaTv tv), where tv is fresh
newSkolemTyVar :: Tc TyVar -- Make a fresh skolem TyVar
unify          :: Tau -> Tau -> Tc ()  -- Unification (may fail)

    -- Free type variables
getMetaTyVars :: [Type] -> Tc [MetaTv]
getFreeTyVars :: [Type] -> Tc [TyVar]
```

Fig. 14: The `TcMonad` module

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 meta type variables (`MetaTvs`) to monotypes—it does not affect concrete 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 15, 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; for example:

$\vdash$	`inferRho`	`::`	`Term -> Tc Rho`
$\vdash^{poly}$	`inferSigma`	`::`	`Term -> Tc Sigma`
$\vdash^{sh}$	`subsCheck`	`::`	`Type  -> Type -> 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 <- newMetaTv
       ; 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 meta type variable (using `newMetaTv`), relying on unification to fill out its value later. *Remember that these meta type variables each stand for a monotype*; as inference proceeds, unification extends an ever-growing substitution, which maps `MetaTv`s to monotypes.

This algorithm is called "Algorithm W" (Milner, 1978). It traverses the term from left to right (e.g. in the `App` case above, we infer the type for `fun` before `arg`), using unification to solve type constraints as it goes. Rather than develop it in full detail, we instead discuss an important variation of the algorithm.

5.4 Propagating types inward

A type inference engine written using Algorithm W 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 <- newMetaTv
                   ; 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, or a result-pointer argument in C: 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 4.7.

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

```
checkRho (Lit i) exp_ty = unify intType 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[8]!

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

[8] 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.

function type. It is needed to implement the matching against the function type $\tau \to \rho$ that is implicit in rule APP.

```
unifyFun :: Rho -> Tc (Rho, Rho)
unifyFun (Fun arg_ty res_ty) = return (arg_ty,res_ty)
unifyFun fun_ty = do { arg_ty <- newMetaTv
                     ; res_ty <- newMetaTv
                     ; 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[9].

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 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 meta 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:

```
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) }
```

[9] Exercise: add another case to optimise the situation where `fun_ty` is a `MetaTv` that is already bound by the substitution.

We use `inferSigma` to infer the (polymorphic) type of `rhs`. Here is its implementation, which can be read directly from the $\vdash^{poly}$ judgement in Figure 4:

```
inferSigma :: Term -> Tc Sigma
inferSigma e = do { res_ty  <- inferRho e
                  ; env_tys <- getEnvTypes
                  ; env_tvs <- getMetaTyVars env_tys
```

```
; res_tvs <- getMetaTyVars [exp_ty]
; let forall_tvs = res_tvs \\ env_tvs
; quantify forall_tvs res_ty }
```

The function `getEnvTypes` returns a list of all the types in the (monad-carried) environment Γ (Figure 14). The function `getMetaTyVars` finds the free meta type variables of a list of types, returning a set of `MetaTv`s. It takes account of the current substitution, which is why it has a monadic type (Figure 14). We quantify over `forall_tvs`, the difference of these two sets, computed using the list-difference operator `(\\)`:

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

When we quantify, we can turn an meta type variable into a concrete type variable, because no further constraints on its value can possibly arise. For example, consider the `Rho`

```
Fun (MetaTv t) (MetaTv t)
```

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

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

where the name `"t"` is chosen arbitrarily[10].

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.

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
```

10 Well, almost arbitrarily: it must not conflict with any concrete 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.


```
       ; subsCheck body_sigma_ty ann_ty
       ; instSigma ann_ty exp_ty }
```

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

```
subsCheck :: Sigma -> Sigma -> Tc ()
subsCheck sigma1 sigma2@(ForAll _ _)      -- Rule SKOL
  = do { (skol_tvs, rho2') <- skolemise sigma2
       ; subsCheck sigma1 rho2'
       ; esc_tvs <- getFreeTyVars [sigma1]
       ; let bad_tvs = filter (`elem` esc_tvs) skol_tvs
       ; check (null bad_tvs)
               "Type not polymorphic enough" }

subsCheck sigma1@(ForAll _ _) rho2        -- Rule INST
  = do { rho1' <- instantiate sigma1
       ; subsCheck rho1' rho2 }

subsCheck rho1 rho2                       -- Rule MONO
  = unify rho1 rho2
```

The second and third equations (corresponding to rules INST and MONO of Figure 4) are quite straightforward, but the first (rule SKOL) requires more care. Here is is again, for reference

$$\frac{\vdash^{sh} \sigma \leq \rho \qquad \overline{a} \notin \mathit{ftv}(\sigma)}{\vdash^{sh} \sigma \leq \forall \overline{a} \,.\, \rho} \text{ SKOL}$$

The function `skolemise` does the alpha-renaming of `sigma2`, to avoid unfortunate name clashes as explained in Section 4.4, returning the fresh (concrete) type variables, or *skolem constants*, as well as the instantiated type:

```
skolemise :: Sigma -> Tc ([TyVar], Rho)
skolemise (ForAll tvs ty)
  = do { sks <- mapM newSkolemTyVar tvs
       ; return (sks, substTy tvs (map TyVar sks) ty) }
skolemise ty
  = return ([], ty)
```

These skolem constants, allocated with `newSkolemTyVar`, still have type `TyVar`, *and they will not unify with anything except themselves and meta type variables.*

After recursively calling `subsCheck`, we must check the side condition $\overline{a} \notin \mathit{ftv}(\sigma)$ 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 `subsCheck` might have bound a meta type variable in `sigma1` to one of the skolems. That is why we wait until *after* the call to `subsCheck` 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 $\forall a.a \rightarrow a$. By this time, `x` will be in the environment with type τ, a meta type variable, so we end up checking this judgement

$$\tau \quad \leq \quad \forall a.a \rightarrow a$$

We skolemise $\forall a.a \rightarrow a$ to get $b \rightarrow b$ (where b is the fresh skolem constant), and then unify, which binds τ to the type $b \rightarrow b$. But rule SKOL requires that the skolem constant b 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 `getFreeTyVars` finds the free `TyVars` of its argument, which are precisely the skolem constants. Like `getMetaTyVars`, `getFreeTyVars` 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 6.5.

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

5.7 Meta type variables and the `Tc` monad

So far we have said little about how meta type variables are represented, or how the `Tc` monad works. In this section we briefly describe them; the full code is in the Appendix.

A meta type variable, of type `MetaTv` is represented like this:

```
data MetaTv = Meta Uniq TyRef

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 `MetaTv` has a unique identity, which is just an `Int`, and a mutable reference cell of type `TyRef`. This mutable cell 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[11]. We need to "lift" the standard operations over `IORef`s (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 `getFreeTyVars`—are in the `Tc` monad.

The `Tc` monad, then, is the `IO` monad augmented with an environment, and a way to report failure[12]:

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

Throughout, we maintain the following invariant:

A meta type variable can only be substituted by a τ-type.

This invariant is absolutely crucial. For example, suppose `f` $:: \tau$, where τ is a meta type variable. If we see the expression `(f 'c', f True)`, we will first unify τ with `Char` $\rightarrow \tau_1$ and then with `Bool` $\rightarrow \tau_2$, and will fail with a type error. But if τ were allowed to be unifiable with $\forall b.b \rightarrow b$—that is, if the meta type variables were really σ variables—this failure would have been premature. (Le Botlan *et al.* deal with this issue by using a constraint system to collect the required instantiations of the type variables; see Section 9.2.)

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 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 a meta 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.

[11] See Peyton Jones (2001) for a tutorial on the `IO` monad. We could also have used the `ST` state transformer monad, since we are not performing any input/output. However, in real life the type checker does perform some limited I/O, mainly to consult interface files of imported modules, so we have used the `IO` monad here.

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

6.1 Changes to the basic structure

In moving to higher rank, we first add a new constructor to the `Term` data type, `ALam` for an annotated lambda:

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

The data type of types remains unchanged. 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 8 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 `MetaTv`, which can be instantiated only to a τ-type, the reference cell in an `Expected Rho` can (indeed must) be filled in by a ρ-type; and we will later encounter `tcPat` which takes an `Expected Sigma` argument, which must be filled in by a σ-type. There is no difficulty here, because these `Expected` locations are always written exactly once—there is no question of unification. On the other hand, we continue to maintain the previous invariant, that a meta type variable can only be bound to a τ-type, for the reasons discussed in Section 5.7.

As in the Damas-Milner case, we will write a Haskell function for each judgement form:

$\vdash_{\delta}$	`tcRho`	`::`	`Term` $\rightarrow$ `Expected Rho -> Tc ()`
$\vdash^{poly}_{\Uparrow}$	`inferSigma`	`::`	`Term -> Tc Sigma`
$\vdash^{poly}_{\Downarrow}$	`checkSigma`	`::`	`Term -> Sigma -> Tc ()`
$\vdash^{inst}_{\delta}$	`instSigma`	`::`	`Sigma -> Expected Rho -> Tc ()`
$\vdash^{dsk}$	`subsCheck`	`::`	`Sigma -> Sigma -> Tc ()`
$\vdash^{dsk*}$	`subsCheckRho`	`::`	`Sigma -> Rho -> Tc ()`

We can write immediately `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.

As we noted in Section 4.7.3, in checking mode we can guarantee that the result type is in weak-prenex form, so we establish the following invariants:

- For `tcRho` and `instSigma`, if the `Expected` argument is `(Check t)`, then `t` is in weak-prenex form.
- For `checkRho` and `subsCheckRho`, the second argument is in weak-prenex form.

These invariants can readily be checked by inspection of the code that follows.

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" version of the judgement $\vdash_{\delta}^{inst}$. Here is its implementation:

```
instSigma :: Sigma -> Expected Rho -> Tc ()
instSigma t1 (Infer r)  = do { t1' <- instantiate t1
                             ; writeTcRef r t1' }
instSigma t1 (Check t2) = subsCheckRho t1 t2
```

In the inference case, following rule INST1, we instantiate the first argument to obtain the result type, which we write into the reference cell.

In the checking case, we simply invoke `subsCheckRho` (rule INST2). In the typing rules, INST2 invokes $\vdash^{dsk}$ (which corresponds to `subsCheck`), but here in the implementation we call `subsCheckRho` (corresponding to $\vdash^{dsk*}$), an improvement relies on `tcRho`'s invariant. We discuss `subsCheckRho` in Section 6.5.

Because `instSigma` deals with the `Expected` argument, it is convenient to re-use it for literals.


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

In our Damas-Milner inference engine, we called `unify` for literals, but we cannot do that here, because `exp_ty` has type `Expected Rho`. Happily, `instSigma` does the job very nicely. Indeed, to a first approximation, to move to higher rank, we simply replace calls to `unify` with calls to `instSigma`!

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

We infer the type of the function, and split its type into its argument and result parts, using `unifyFun` from Section 5.4.

Returning to the `App` case of `tcRho`, 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 `instSigma` 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_ty <- newTyVar
       ; 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 meta 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. Notice that we can call `checkRho`, rather than `checkSigma`, because `body_ty` is guaranteed to be a ρ-type by the invariant for `tcRho` (Section 6.1).

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 Generalisation

The judgement $\vdash_{\delta}^{poly}$ in Figure 8 infers or checks that a term has a polytype. All its invocations have a known direction ($\Uparrow$ or $\Downarrow$), as the reader may verify from Figure 8, so we implement it with two functions, `inferSigma` and `checkSigma`. The former implements rule GEN1, and its code is unchanged from the Damas-Milner version given in Section 5.5.

However, we also need `checkSigma`, which implements rule GEN2. Here is the code, which is mostly a straight transliteration of the rule:

```
checkSigma :: Term -> Sigma -> Tc ()
checkSigma expr sigma
  = do { (skol_tvs, rho) <- skolemise sigma
       ; checkRho expr rho
       ; env_tys <- getEnvTypes
       ; esc_tvs <- getFreeTyVars (sigma : env_tys)
       ; let bad_tvs = filter (`elem` esc_tvs) skol_tvs
       ; check (null bad_tvs)
               (text "Type not polymorphic enough") }
```

We met the function `skolemise` in Section 5.6, but we must modify it to perform *deep* skolemisation, as we discussed in Section 4.6.2. This is easily done, just by altering its definition so that it looks under `Fun` arrows:

```
skolemise :: Sigma -> Tc ([TyVar], Rho)
skolemise (ForAll tvs ty)        -- Rule PRPOLY
  = do { sks1 <- mapM newSkolemTyVar tvs
       ; (sks2, ty') <- skolemise (substTy tvs (map TyVar sks1) ty)
```

```
      ; return (sks1 ++ sks2, ty') }
skolemise (Fun arg_ty res_ty)        -- Rule PRFUN
  = do { (sks, res_ty') <- skolemise res_ty
       ; return (sks, Fun arg_ty res_ty') }
skolemise ty                         -- Rule PRMONO
  = return ([], ty)
```

The three equations correspond directly to the three rules of the function $pr(\sigma)$ in Figure 7.

Returning to `checkSigma`, once we have obtained the skolemised type $\forall\overline{a}.\rho$, we check that the term indeed has type ρ, using `checkRho`. Lastly, we must check that none of the skolem constants $\overline{a}$ have escaped into the environment. And therein lies a tricky point. Rule GEN2 merely says $\overline{a} \notin \Gamma$, but our code calls `getFreeTyVars` on `sigma` as well as `env_tys`. The reason is this: although the skolem constants `skol_tvs` cannot, by construction, appear free in `sigma` *before* the call to `checkRho`, they may do so *afterwards*, because a meta type variable in `sigma` might be unified with one of them.

Here is a real example. Consider the types of `runST` and `newRef`:

$$\begin{aligned} \texttt{runST} &:: \forall a.(\forall s.\texttt{ST}\ s\ a) \to a \\ \texttt{newRef} &:: \forall s\ a.a \to \texttt{ST}\ s\ (\texttt{Ref}\ s\ a) \end{aligned}$$

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 (see Section 2.5 for an explanation of why not).

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

$$(\forall s.\texttt{ST}\ s\ \tau) \to \tau$$

Next, we will call `checkSigma` on the expression `(newRef 'c')`, with expected type $\forall s.\texttt{ST}\ s\ \tau$. In turn, `checkSigma` will skolemise s to s', say, and call `checkRho` to check that `(newRef 'c')` has type $\texttt{ST}\ s'\ \tau_1$. This will succeed, but in doing so *it will bind the meta type variable* τ to $\texttt{Ref}\ s'\ \texttt{Char}$.

Notice what has happened here. The meta type variable τ in `sigma` has become bound to a type involving the skolem constant s'. That is why we must include `sigma` in the call to `getFreeTyVars`. This point is rather subtle and easily overlooked, which 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.

6.5 Subsumption

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 6.5.1).
- We must refine the implementation of skolemisation (Section 6.5.2).

6.5.1 Subsumption for function types

In this section we will define `subsCheckRho`, which implements the auxiliary judgement $\vdash^{dsk*}$ in Figure 7. At first it seems simple to read off the implementation from the rules:

```
subsCheckRho :: Sigma -> Rho -> Tc ()
-- Invariant: the second argument is in weak-prenex form

subsCheckRho sigma1@(ForAll _ _) rho2                -- Rule SPEC
  = do { rho1 <- instantiate sigma1
       ; subsCheckRho rho1 rho2 }

subsCheckRho (Fun arg1 res1) (Fun arg2 res2)  -- Rule FUN
  = do { subsCheck arg2 arg1
       ; subsCheckRho res1 res2 }

subsCheckRho tau1 tau2                          -- Rule MONO
  = unify tau1 tau2      -- Revert to ordinary unification
```

Notice the invariant: $\vdash^{dsk*} \sigma \leq \rho$ is invoked only when ρ is in weak-prenex form. Hence `subsCheckRho` needs no `ForAll` case for its second argument.

This implementation is not quite right, however, because either argument might be a meta type variable. In that case, if the other argument is a `Fun`, we should use `unifyFun` to persuade the meta type variable to look like a `Fun` too. To do this, we must replace the `Fun`/`Fun` equation with two equations, thus:

```
subsCheckRho t1 (Fun a2 r2)
  = do { (a1,r1) <- unifyFun t1; subsCheckFun a1 r1 a2 r2 }
subsCheckRho (Fun a1 r1) t2
  = do { (a2,r2) <- unifyFun t2; subsCheckFun a1 r1 a2 r2 }

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


6.5.2 Skolemisation revisited

Next, we turn our attention to the $\vdash^{dsk}$ judgement, implemented by `subsCheck`. Its implementation follows closely that of `checkSigma` (Section 6.4), just as rule DEEP-SKOL is similar to GEN2.

```
subsCheck sigma1 sigma2         -- Rule DEEP-SKOL
  = do { (skol_tvs, rho2) <- skolemise sigma2
       ; subsCheckRho sigma1 rho2
       ; esc_tvs <- getFreeTyVars [sigma1,sigma2]
              -- The line above has changed!
       ; let bad_tvs = filter (`elem` esc_tvs) skol_tvs
```

```
    ; check (null bad_tvs)
            (vcat [text "Subsumption check failed:",
                   nest 2 (ppr sigma1),
                   text "is not as polymorphic as",
                   nest 2 (ppr sigma2)])
  }
```

Just as in `checkSigma`, notice that we had to call `getFreeTyVars` on `sigma2` as well as `sigma1`, whereas only the latter is obvious from the rule. In fact, this change (compared to Section 5.6) is not fundamentally related to higher-rank types: it arises whenever `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 `sigma2` can have no free meta type variables. 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, with exactly the same solution, would arise in the Damas-Milner system too.

6.6 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`	252	252
`TcMonad`	292	292
`TcTerm`	106	151
Total	650	695

The only significant changes are around 35 lines of code required to implement subsumption checking in `TcTerm`, plus about another 10 to handle the `ALam` cases in `tcRho`.

Proportionally, 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 many more term forms, but only a few more type forms) but the same 45 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 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:

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

In checking mode, everything is easy; we simply push the result type into the branches of the conditional, thus:

$$\frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\Downarrow} e_2 : \rho \quad \Gamma \vdash_{\Downarrow} e_3 : \rho}{\Gamma \vdash_{\Downarrow} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho}\ \textsc{if2}$$

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. Thus far, however, we have only unified *monotypes*, but the inferred types of the branches will be ρ-types. At this point, there are three possible design choices:

1. *Insist that the branches are monotyped.* This is exactly what will happen if we expressed conditionals using a function, instead of syntactic form:

   ```
   cond :: Bool -> a -> a -> a
   ```

   Since the type variable `a` can only be instantiated with a monotype, the branches will be monotyped. It is easy to express this condition directly in the typing judgement for `if`:

   $$\frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\Uparrow} e_2 : \tau \quad \Gamma \vdash_{\Uparrow} e_3 : \tau}{\Gamma \vdash_{\Uparrow} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \text{IF1A}$$

   Note the monotype τ in the two premises and conclusion.

2. *Elaborate unification to handle polytypes.* It is possible to modify the unifier so that it can unify polytypes: when it encounters a $\forall$ quantifier in one type,

it insists on a $\forall$ in the other. This is called "*unification under a mixed prefix*" and has been well studied (Miller, 1992). The typing rule is now the same for both inference and checking, so we can use use a direction-polymorphic rule:

$$\frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\delta} e_2 : \rho \quad \Gamma \vdash_{\delta} e_3 : \rho}{\Gamma \vdash_{\delta} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho}\ \textsc{if}$$

3. *Allow polytyped branches by performing two-way subsumption* In this case we simply check that in inference mode, the two types of the branches are equivalent in our subsumption relation, and return one of them.

$$\frac{\Gamma \vdash_{\Downarrow} e_1 : \texttt{Bool} \quad \Gamma \vdash_{\Uparrow} e_2 : \rho_1 \quad \Gamma \vdash_{\Uparrow} e_3 : \rho_2 \quad \vdash^{dsk} \rho_1 \leq \rho_2 \quad \vdash^{dsk} \rho_2 \leq \rho_1}{\Gamma \vdash_{\Uparrow} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \rho_1}\ \textsc{if}$$

Choices (2) and (3) are more satisfactory than (1), because they ensure that a conditional (or a `case` expression, or pattern-matching in a function definition) does not accidentally kill higher-rank polymorphism.

It is worth noting that although choice (2) types more programs than (1) (but fewer than (3)), it does lose one property, namely clause (3) of Theorem 4.14. The theorem says that if a term typechecks in an environment Γ, and we make one of the bindings in Γ more polymorphic with respect to the deep-skolemisation relation, then the term should still typecheck. But consider `(if x then f1 else f2)`, where `f1` and `f2` have identical, higher-rank types. The program will typecheck under IF. But if we make `f1` more polymorphic, and its type has a different "shape" from that of `f2`, the program will be rejected. We are not unduly worried about this: it is easy to make the program work again using a type signature, but the loss of the theorem is worth noting.

Implementing choice (1) is easy. How can the implementation guarantee to infer only a monotype for e_2 and e_3? By passing in a fresh meta type variable, just as would happen if we used the polymorphic `cond` function, thus:

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

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

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

Implementing choice (2) is more involved, because we must modify the unification algorithm to handle polytypes. In particular, when unifying two polymorphic types, we have to skolemise both using the same skolem constants (which results in the unsatisfactory situation where the order of bound variables is significant for unification), subsequently recursively unify the resulting ρ-types, and finally ensure that no skolem variable escaped by getting unified with a unification variable in the bodies of types. To make this algorithm independent from the order in which skolemisation/quantification happens we would have to maintain a separate bijection between the skolem variables of the two types.

Choice (3) looks more sophisticated than (2). Nevertheless, it is much simpler to implement, because the subsumption check already implements all the tricky points we mentioned for choice (2)! Here is the code:

```
tcRho (If e1 e2 e3) (Infer ref)
  = do { checkRho e1 boolType
       ; rho1 <- inferRho e2
       ; rho2 <- inferRho e3
       ; subsCheck rho1 rho2
       ; subsCheck rho2 rho1
       ; writeTcRef ref rho1 }
```

The only unsatisfactory point is that the type rule in (3) arbitrarily chooses to give the expression type ρ_1, rather ρ_2. Although these types are equivalent, they may look different; for example $\rho_1 = Int \rightarrow \forall a.Int \rightarrow a \rightarrow a$, $\rho_2 = Int \rightarrow Int \rightarrow \forall a.a \rightarrow a$. This infelicity could be circumvented by skolemising the return type and re-generalising at the top-level all of its quantified variables.

7.2 *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:

$$
\begin{array}{llcll}
\text{Terms} & t, u & ::= & \ldots & \\
 & & | & \backslash p.t & \text{Pattern abstraction} \\
 & & & & \\
\text{Patterns} & p & ::= & x & \text{Variable} \\
 & & | & \_ & \text{Wild card} \\
 & & | & (p::\sigma) & \text{Type annotated pattern} \\
 & & | & (p_1, p_2) & \text{Pair} \\
 & & | & \ldots &
\end{array}
$$

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

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

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:

$$\frac{\begin{array}{c}\vdash_\delta^{pat} p : \sigma, \Gamma' \\ \Gamma, \Gamma' \vdash_\delta t : \rho\end{array}}{\Gamma \vdash_\delta (\backslash p \,.\, t) : (\sigma \to \rho)} \text{ABS}$$

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
                                ; writeTcRef ref ty
                                ; return [(v,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.

7.3 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. 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. Correspondingly we extend `tcPat` as follows[13]:

[13] We use standard Haskell functions

```
zip :: [a] -> [b] -> [(a,b)]
concat :: [[a]] -> [a]
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```



```
tcPat (PCon con ps) exp_ty
  = do { (arg_tys, res_ty) <- instDataCon con
       ; envs <- mapM check_arg (ps `zip` arg_tys)
       ; instPatSigma 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 are pushed into the argument patterns.

7.4 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` $(\forall a.a \rightarrow \texttt{Int})$? 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 $(\forall a.a \rightarrow \texttt{Tree}\ a)$, and our restriction to predicativity therefore requires that we instantiate `Leaf` only at a τ-type (Section 3.4). So the simplest solution is to require that type constructors are parameterised only by monotypes. Then, just as `instSigma` instantiates a polymorphic function with fresh meta (mono-)type variables, so `instDataCon` instantiates the data constructor's type with fresh meta (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 $[\forall a.a \rightarrow a]$ is more polymorphic than $[\texttt{Int} \rightarrow \texttt{Int}]$. 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` $(\texttt{Int} \rightarrow \texttt{Int})$ would be more, rather than less, polymorphic than `Contra` $(\forall a.a \rightarrow a)$. The situation gets 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 4.8), one would actually need to traverse the entire list at runtime, coercing each function in the list from type $(\forall a.a \rightarrow a)$ to $(\texttt{Int} \rightarrow \texttt{Int})$. 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, and allow them to have polymorphic components. 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:

$$
\begin{array}{llcl}
\text{Polytypes} & \sigma & ::= & \forall \overline{a} \,.\, \rho \\
\text{Rho-types} & \rho & ::= & \sigma_1 \to \sigma_2 \mid (\sigma_1, \ldots, \sigma_n) \mid \tau \\
\text{Monotypes} & \tau & ::= & \tau_1 \to \tau_2 \mid (\tau_1, \ldots, \tau_n) \mid K\,\overline{\tau} \mid a
\end{array}
$$

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

$$
\frac{\Gamma \vdash_{\Uparrow} t_i : \rho_i \quad (1 \leq i \leq n)}{\Gamma \vdash_{\Uparrow} (t_1, \ldots, t_n) : (\rho_1, \ldots, \rho_n)}\ \textsc{tup1}
\qquad
\frac{\Gamma \vdash_{\Downarrow} t_i : \sigma_i \quad (1 \leq i \leq n)}{\Gamma \vdash_{\Downarrow} (t_1, \ldots, t_n) : (\sigma_1, \ldots, \sigma_n)}\ \textsc{tup2}
$$

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

$$
\frac{\vdash^{dsk} \sigma_i \leq \sigma_i' \quad (1 \leq i \leq n)}{\vdash^{dsk} (\sigma_1, \ldots, \sigma_n) \leq (\sigma_1', \ldots, \sigma_n')}\ \textsc{tuple}
$$

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 polymorphic components. However, a function such as

$$\texttt{fst} :: \forall ab.(a, b) \to a$$

can still only be used predicatively, because it is an ordinary polymorphic function. For example, the application

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

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.

8 Type-directed translation

In Section 4.8 we showed how to incorporate a type-directed translation into the typing rules of the language. We now briefly discuss the following question: how can we adapt our type inference engine so that it performs type-directed translation at the same time as type inference?

Fortunately, the answer 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 function 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`.

8.1 Implementing subsCheck

The implementation of `subsCheck` is a straightforward extension of the code we developed in Sections 5.6 and 6.5. One interesting case is `subsCheckFun`, which 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 <- subsCheckRho 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`[14].

In a similar way, type abstractions are generated by `subsCheck`, and type applications by `subsCheckRho` (see Section 6.5), but we omit the details here.

8.2 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)) ::` $(\forall a.a \rightarrow a) \rightarrow$ `Int` $\rightarrow$ `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.

[14] 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 variable name, or by using an extra `let` binding.

8.3 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). We have found the mechanism required to support type classes in a non-higher-rank system (such as Haskell 98) requires virtually no change to support higher rank types; in that sense, the two features are almost entirely orthogonal.

8.4 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 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. $(g : \forall b.b \rightarrow a)$). 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, does not interact well with user-supplied type annotations, and has not, to our knowledge, been implemented in a production compiler. All of these results are for the standard, impredicative, System F. We do not know of analogous results for the predicative fragment of System F.

9.2 ML^F

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—building on previous work by Garrigue and Rémy on extending ML with semi-explicit first-class polymorphism (Garrigue & Rémy, 1999)—have described a new and ingenious type system, ML^F, which supports the impredicative polymorphism of System F while retaining principal types (Le Botlan & Rémy, 2003). They achieve this remarkable *rapprochement* using a form of constrained polymorphism, with a constraint domain very reminiscent of Huet's classic higher-order unification algorithm (Huet, 2002). Hence their system is actually *more* expressive than System F.

Like us, they do not attempt to infer higher-ranked polymorphism, and instead accept that the programmer will have to guide the type system using annotations.

Also like us, every program typeable by Damas-Milner can be typed in ML^F without any annotations at all. Though not described in their paper, they also suggest that annotations may be propagated as we have described here.

However, unlike us, they allow type variables to be instantiated to type schemes. Furthermore, their type system can discover an appropriate instantiation without the aid of any annotations, at least for arguments which are simply "passed through" functions. Additionally, ML^F only supports covariant instantiation of type schemes.

The price they pay for these remarkable results is a somewhat complicated type system. The constraints require that higher-ranked types be encoded in a form which makes manifest any potential sharing of type variables. The programmer

must perform this encoding, and be prepared to interpret the type schemes and constraints which come back from type inference and type errors. On the other hand, even more recent work (Leijen & Lh, 2005) indicates that this complexity eventually may not be daunting.

Overall, we can say ML^F supports impredicativity but with a somewhat more indirect approach to higher-ranked types and a more sophisticated inference algorithm, while our system supports higher-ranked types directly and has a simple inference algorithm, but without support for impredicativity. 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 potential cost/benefit trade-off to be made, with our system and ML^F occupying interestingly different points on the design spectrum.

9.3 Type inference in general

Considering how many papers there are on type systems, there is surprising little literature on type inference that is aimed unambiguously at implementors. Cardelli's paper was the first widely-read tutorial (Cardelli, 1987), with Hancock's tutorial shortly afterwards (Hancock, 1987). More recently Mark Jones's paper "Typing Haskell in Haskell" gave an executable implementation of Haskell's type system (Jones, 1999). Apart from the higher-rank aspect, the distinguishing feature of our presentation is the pervasive use of a monad to structure the type inference engine, and the use of the `Expected Rho` argument to represent the bidirectional nature of local type inference.

9.4 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*[15] 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

[15] "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.

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 8.

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

```
foo w = bar (\x. x+w)
```

Since the call to `bar` is instantiated at some unknown type α, local type inference will not push the partially-known type $\texttt{Int} \to \alpha$ into the anonymous function passed to `bar`, 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 *etal.*, 2001). In effect, local type inference uses $\Uparrow$ and $\Downarrow$ in *judgements*, whereas coloured local type inference goes further and pushes $\Uparrow$ and $\Downarrow$ into *types* as well. The system is, however, rather complex.

Coloured type inference was, like local type inference, originally developed 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. For example, our system has no difficulty with the function `foo` above, 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 unbound meta type variables. On the other hand, being able to pass in partial type information could still be useful: notably, in Section 4.7 we discussed the information-loss of rule APP in Figure 8.

An important point of our bidirectional system is that the types of terms may be determined with the help of user annotations that are not "on" the terms themselves, but maybe further away. A different approach to partial type inference, as suggested by Rémy (Rémy, 2005), is to introduce an *elaboration* phase prior to the actual type inference. During the elaboration phase, bidirectional propagation of user annotations determines the polymorphic *shapes* of terms. Shapes capture polymorphic information that cannot be inferred and originates in annotations. During elaboration monomorphic information is kept abstract. However at the end of the elaboration phase, each term need only be checked against its polymorphic shape—and the monomorphic type information can be inferred with a unification-based mechanism. Additionally, in his paper Remy discusses the predicative fragment of System F and System F closed under η-expansion, and describes the necessary changes if side-effects are to be added.

9.5 *Deep skolemisation subsumption*

It turns out that our deep skolemisation relation corresponds to the predicative restriction of a subsumption relation (denoted with $\vdash^{\eta} \sigma_1 \leq \sigma_2$) originally proposed by Mitchell (Mitchell, 1988).

Theorem 9.1 $\vdash^{dsk} \sigma_1 \leq \sigma_2$ *if and only if* $\vdash^{\eta} \sigma_1 \leq \sigma_2$.

Mitchell's original relation, also referred to as Mitchell's *containment* relation, is used in type inference for the System F language, closed under η-expansion. Mitchell showed that a derivation of $\vdash^{\eta} \sigma_1 \leq \sigma_2$ exists iff there exists a System F function of type $\sigma_1 \rightarrow \sigma_2$ that, after erasure of types, $\beta\eta$-reduces to the identity. Crucial to this is the distributivity property, which in our system is given by Theorem 4.3. The impredicative version of type containment was shown to be undecidable (Tiuryn & Urzyczyn, 1996). Mitchell's containment was originally presented in a declarative style; syntax-directed presentations of containment are also well known (Tiuryn, 2001; Longo *etal.*, 1995). In particular, Longo *et al.* (Longo *etal.*, 1995) employ an idea similar to our deep skolemisation. To the best of our knowledge, no one had previously considered whether the predicative variant of the containment relation was decidable, although it is not a hard problem; our algorithm in Figure 7 shows that it is decidable.

9.6 Improving error messages

Historically, the most common approach to inference for ML-style type systems, is the "top-down, left-right" approach, called Algorithm W (Damas & Milner, 1982), which we introduced in Section 5.3. One big improvement is to use the "pushing types inwards" trick that we have used extensively in this paper (Section 5.4). 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, gives a big improvement in most cases, and rewards the programmer for supplying type signatures, but it does not *guarantee* an improvement.

The trouble is that even Algorithm M has a left-to-right bias. 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 $[\tau]$, for some unknown, meta type τ.

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 (2005) for a rather detailed treatment of this idea, which is also the basis for Helium's type checker (Heeren *etal.*, 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 `getFreeTyVars` or `getMetaTyVars`. In short, almost all of the necessary changes could be hidden in the implementation of the monadic primitives of Figure 14.

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, Gabor Greif, Matt Hellige, Dean Herrington, Ian Lynagh, Greg Morrisett, Amr Sabry, David Sankel, Yanling Wang, Keith Wansbrough, Geoffrey Washburn, Joe Wells, and Carl Witty. We are particularly indebted to Andres Löh, François Pottier, Josef Svenningsson, Norman Ramsey and his students, and the JFP referees, for their detailed and insightful comments.

References

Baars, Arthur L, & Swierstra, S. Doaitse. (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.

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, 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.

Garrigue, Jacques, & Rémy, Didier. (1999). Semi-explicit first-class polymorphism for ML. *Journal of information and computation*, **155**, 134–169.

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 Press. 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.

Hancock, P. (1987). A type checker. *Pages 163–182 of:* Peyton Jones, SL (ed), *The implementation of functional programming languages*. Prentice Hall.

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**.

Huet, G. (2002). Higher order unification 30 years later. *15th international workshop on higher order logic theorem proving and its applications (IWHOLTP'02)*. LNCS.

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

ICFP05. (2005). *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*. Tallinn, Estonia: ACM.

Jones, Mark. (1999). Typing Haskell in Haskell. Meijer, Erik (ed), *Proceedings of the 1999 haskell workshop*. Technical Reports, nos. UU–CS–1999–28. Available at `ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf`.

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.

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 Press.

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.

Leijen, Daan, & Lh, Andres. (2005). Qualified types for MLF. *In:* (ICFP05, 2005).

Longo, Giuseppe, Milsted, Kathleen, & Soloviev, Sergei. (1995). A logic of subtyping (extended abstract). *Pages 292–299 of:* |*lics95*|.

Miller, Dale. (1992). Unification under a mixed prefix. *J. symb. comput.*, **14**(4), 321–358.

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

Mitchell, John C. (1988). Polymorphic type inference and containment. *Inf. comput.*, **76**(2-3), 211–249.

Morrisett, G. 1995 (Dec.). *Compiling with types*. Ph.D. thesis, Carnegie Mellon University.

Odersky, M, & Lufer, 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. (2002). *Types and programming languages*. MIT Press.

Pierce, Benjamin C., & Turner, David N. (1998). Local type inference. *Pages 252–265 of: 25th ACM Symposium on Principles of Programming Languages (POPL'98)*. San Diego: ACM.

Pottier, François, & Rémy, Didier. (2005). The essence of ML type inference. *Chap. 10, pages 389–489 of:* Pierce, Benjamin C. (ed), *Advanced topics in types and programming languages*. MIT Press.

Rémy, Didier. (2005). Simple, partial type inference for System F, based on type containment. *In:* (ICFP05, 2005).

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*. 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.

Tiuryn, J, & Urzyczyn, P. (1996). The subtyping problem for second order types is undecidable. *Pages 74–85 of: Proc ieee symposium on logic in computer science (lics'96)*.

Tiuryn, Jerzy. (2001). A sequent calculus for subtyping polymorphic types. *Inf. comput.*,

164(2), 345–369.

Vytiniotis, Dimitrios, Weirich, Stephanie, & Peyton Jones, Simon. 2005 (July). *Practical type inference for arbitrary-rank types, Technical Appendix*. Tech. rept. MS-CIS-05-14. University of Pennsylvania.

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

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 Data.IORef
import TcMonad
import List( (\\) )
import Text.PrettyPrint.HughesPJ
```

```haskell
------------------------------------------
--      The top-level wrapper           --
------------------------------------------

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

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

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


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

checkRho :: Term -> Rho -> Tc ()
-- Invariant: the Rho is always in weak-prenex form
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 ()
-- Invariant: if the second argument is (Check rho),
--               then rho is in weak-prenex form
tcRho (Lit _) exp_ty
  = instSigma 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
       ; instSigma 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  <- newTyVarTy
       ; 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
        ; env_tys <- getEnvTypes
         ; env_tvs <- getMetaTyVars env_tys
        ; res_tvs <- getMetaTyVars [exp_ty]
        ; let forall_tvs = res_tvs \\ env_tvs
        ; quantify forall_tvs exp_ty }

checkSigma :: Term -> Sigma -> Tc ()
checkSigma expr sigma
  = do { (skol_tvs, rho) <- skolemise sigma
       ; checkRho expr rho
```



```
; env_tys <- getEnvTypes
; esc_tvs <- getFreeTyVars (sigma : env_tys)
; let bad_tvs = filter (`elem` esc_tvs) skol_tvs
; check (null bad_tvs)
       (text "Type not polymorphic enough") }
```

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

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

subsCheck sigma1 sigma2        -- Rule DEEP-SKOL
  = do { (skol_tvs, rho2) <- skolemise sigma2
       ; subsCheckRho sigma1 rho2
       ; esc_tvs <- getFreeTyVars [sigma1,sigma2]
       ; let bad_tvs = filter (`elem` esc_tvs) skol_tvs
       ; check (null bad_tvs)
               (vcat [text "Subsumption check failed:",
                      nest 2 (ppr sigma1),
                      text "is not as polymorphic as",
                      nest 2 (ppr sigma2)])
    }

subsCheckRho :: Sigma -> Rho -> Tc ()
-- Invariant: the second argument is in weak-prenex form

subsCheckRho sigma1@(ForAll _ _) rho2          -- Rule SPEC
  = do { rho1 <- instantiate sigma1
       ; subsCheckRho rho1 rho2 }
```

```
subsCheckRho rho1 (Fun a2 r2)                -- Rule FUN
  = do { (a1,r1) <- unifyFun rho1; subsCheckFun a1 r1 a2 r2 }

subsCheckRho (Fun a1 r1) rho2                -- Rule FUN
  = do { (a2,r2) <- unifyFun rho2; subsCheckFun a1 r1 a2 r2 }

subsCheckRho tau1 tau2                       -- Rule MONO
  = unify tau1 tau2    -- Revert to ordinary unification

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

instSigma :: Sigma -> Expected Rho -> Tc ()
-- Invariant: if the second argument is (Check rho),
--                then rho is in weak-prenex form
instSigma t1 (Check t2) = subsCheckRho t1 t2
instSigma t1 (Infer r)  = do { t1' <- instantiate t1
                             ; 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,
          getEnvTypes, getFreeTyVars, getMetaTyVars,

        -- Types and unification
        newTyVarTy,
        instantiate, skolemise, zonkType, quantify,
        unify, unifyFun,

        -- Ref cells
        newTcRef, readTcRef, writeTcRef

    ) where

import BasicTypes
import qualified Data.Map as Map
import Text.PrettyPrint.HughesPJ
import Data.IORef
import List( nub, (\\) )

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

data TcEnv
```

```
  = TcEnv { uniqs   :: IORef Uniq,            -- Unique supply
            var_env :: Map.Map Name Sigma  -- Type environment for term variables
    }

newtype Tc a = Tc (TcEnv -> IO (Either ErrMsg a))
unTc :: Tc a ->   (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  _ = 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 = Map.fromList 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 = Map.insert var ty (var_env env) }

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

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


--------------------------------------------------
--      Creating, reading, writing MetaTvs       --
--------------------------------------------------

newTyVarTy :: Tc Tau
newTyVarTy = do { tv <- newMetaTyVar
```



```
                ; return (MetaTv tv) }

newMetaTyVar :: Tc MetaTv
```

```haskell
newMetaTyVar = do { uniq <- newUnique
                  ; tref <- newTcRef Nothing
                  ; return (Meta uniq tref) }

newSkolemTyVar :: TyVar -> Tc TyVar
newSkolemTyVar tv = do { uniq <- newUnique
                       ; return (SkolemTv (tyVarName tv) uniq) }


readTv  :: MetaTv -> Tc (Maybe Tau)
readTv  (Meta _ ref) = readTcRef ref

writeTv :: MetaTv -> Tau -> Tc ()
writeTv (Meta _ 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 tvs ty)
  = do { tvs' <- mapM (\_ -> newMetaTyVar) tvs
       ; return (substTy tvs (map MetaTv tvs') ty) }
instantiate ty
  = return ty

skolemise :: Sigma -> Tc ([TyVar], Rho)
-- Performs deep skolemisation, retuning the
-- skolem constants and the skolemised type
skolemise (ForAll tvs ty)         -- Rule PRPOLY
  = do { sks1 <- mapM newSkolemTyVar tvs
       ; (sks2, ty') <- skolemise (substTy tvs (map TyVar sks1) ty)
       ; return (sks1 ++ sks2, ty') }
skolemise (Fun arg_ty res_ty)         -- Rule PRFUN
  = do { (sks, res_ty') <- skolemise res_ty
       ; return (sks, Fun arg_ty res_ty') }
skolemise ty                             -- Rule PRMONO
  = return ([], ty)

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

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

quantify :: [MetaTv] -> 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' <- zonkType 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 :: [TyVar]    -- a,b,..z, a1, b1,... z1, a2, b2,...
allBinders = [ BoundTv [x]          | x <- ['a'..'z'] ] ++
             [ BoundTv (x : show i) | i <- [1 :: Integer ..], x <- ['a'..'z']]

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

getEnvTypes :: Tc [Type]
  -- Get the types mentioned in the environment
```

```
getEnvTypes = do { env <- getEnv;
                 ; return (Map.elems env) }

getMetaTyVars :: [Type] -> Tc [MetaTv]
-- This function takes account of zonking, and returns a set
-- (no duplicates) of unbound meta-type variables
getMetaTyVars tys = do { tys' <- mapM zonkType tys
                       ; return (metaTvs tys') }

getFreeTyVars :: [Type] -> Tc [TyVar]
-- This function takes account of zonking, and returns a set
-- (no duplicates) of free type variables
getFreeTyVars tys = do { tys' <- mapM zonkType tys
                          ; return (freeTyVars tys') }

------------------------------------------
--      Zonking                         --
-- Eliminate any substitutions in the type
------------------------------------------

zonkType :: Type -> Tc Type
zonkType (ForAll ns ty) = do { ty' <- zonkType ty
                             ; return (ForAll ns ty') }
zonkType (Fun arg res)  = do { arg' <- zonkType arg
                             ; res' <- zonkType res
                             ; return (Fun arg' res') }
```

```haskell
zonkType (TyCon tc)      = return (TyCon tc)
zonkType (TyVar n)       = return (TyVar n)
zonkType (MetaTv tv)     -- A mutable type variable
  = do { mb_ty <- readTv tv
```



```haskell
       ; case mb_ty of
           Nothing -> return (MetaTv tv)
           Just ty -> do { ty' <- zonkType ty
                         ; writeTv tv ty'       -- "Short out" multiple hops
                         ; return 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 (TyVar tv1)  (TyVar tv2)  | tv1 == tv2 = return ()
```

```haskell
unify (MetaTv tv1) (MetaTv tv2) | tv1 == tv2 = return ()
unify (MetaTv tv) ty = unifyVar tv ty
unify ty (MetaTv tv) = unifyVar tv ty

unify (Fun arg1 res1)
      (Fun arg2 res2)
  = do { unify arg1 arg2; unify res1 res2 }

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

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

-----------------------------------------
unifyVar :: MetaTv -> 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 :: MetaTv -> Tau -> Tc ()
-- Invariant: the flexible type variable tv1 is not bound
unifyUnboundVar tv1 ty2@(MetaTv 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 (MetaTv tv1) ty2'
          Nothing  -> writeTv tv1 ty2 }

unifyUnboundVar tv1 ty2
  = do { tvs2 <- getMetaTyVars [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 (Fun arg res) = return (arg,res)
unifyFun tau           = do { arg_ty <- newTyVarTy
                            ; res_ty <- newTyVarTy
                            ; unify tau (arg_ty --> res_ty)
                            ; return (arg_ty, res_ty) }
```

```
-----------------------------------------
occursCheckErr :: MetaTv -> 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 (BoundTv _)) = True
badType _                   = 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 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
```

```haskell
atomicTerm :: Term -> Bool
atomicTerm (Var _) = True
atomicTerm (Lit _) = True
atomicTerm _       = False


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

type Sigma = Type
type Rho   = Type          -- No top-level ForAll
type Tau   = Type          -- No ForAlls anywhere

data Type = ForAll [TyVar] Rho          -- Forall type
          | Fun    Type Type            -- Function type
          | TyCon  TyCon                 -- Type constants
          | TyVar  TyVar                 -- Always bound by a ForAll
          | MetaTv MetaTv                -- A meta type variable

data TyVar
  = BoundTv String                  -- A type variable bound by a ForAll

  | SkolemTv String Uniq         -- A skolem constant; the String is
                                    -- just to improve error messages
```

```haskell
data MetaTv = Meta Uniq TyRef  -- Can unify with any tau-type

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

instance Eq MetaTv where
  (Meta u1 _) == (Meta u2 _) = u1 == u2

instance Eq TyVar where
  (BoundTv s1)    == (BoundTv s2)    = s1 == s2
  (SkolemTv _ u1) == (SkolemTv _ u2) = u1 == u2

type Uniq = Int

data TyCon = IntT | BoolT
```



```haskell
         deriving( Eq )

---------------------------------
--      Constructors
```

```haskell
(-->) :: Sigma -> Sigma -> Sigma
arg --> res = Fun arg res

intType, boolType :: Tau
intType  = TyCon IntT
boolType = TyCon BoolT

---------------------------------
--          Free and bound variables

metaTvs :: [Type] -> [MetaTv]
-- Get the MetaTvs from a type; no duplicates in result
metaTvs tys = foldr go [] tys
  where
    go (MetaTv tv)   acc
         | tv `elem` acc  = acc
         | otherwise          = tv : acc
    go (TyVar _)     acc = acc
    go (TyCon _)     acc = acc
    go (Fun arg res) acc = go arg (go res acc)
    go (ForAll _ ty) acc = go ty acc         -- ForAll binds TyVars only

freeTyVars :: [Type] -> [TyVar]
-- Get the free TyVars from a type; no duplicates in result
freeTyVars tys = foldr (go []) [] tys
```

```
  where
    go :: [TyVar]          -- Ignore occurrences of bound type variables
       -> Type                -- Type to look at
       -> [TyVar]          -- Accumulates result
       -> [TyVar]
    go bound (TyVar tv)      acc
      | tv `elem` bound        = acc
      | tv `elem` acc                  = acc
      | otherwise                  = tv : acc
    go bound (MetaTv _)      acc = acc
    go bound (TyCon _)       acc = acc
    go bound (Fun arg res)   acc = go bound arg (go bound res acc)
    go bound (ForAll tvs ty) acc = go (tvs ++ bound) ty acc

tyVarBndrs :: Rho -> [TyVar]
-- 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 (ForAll tvs body) = tvs ++ bndrs body
    bndrs (Fun arg res)     = bndrs arg ++ bndrs res
    bndrs _                 = []
```



```
tyVarName :: TyVar -> String
```

```haskell
tyVarName (BoundTv n)    = n
tyVarName (SkolemTv n _) = n


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

type Env = [(TyVar, Tau)]

substTy :: [TyVar] -> [Type] -> Type -> Type
-- Replace the specified quantified type variables by
-- given meta type variables
-- No worries about capture, because the two kinds of type
-- variable are distinct
substTy tvs tys ty = subst_ty (tvs `zip` tys) ty

subst_ty :: Env -> Type -> Type
subst_ty env (Fun arg res)   = Fun (subst_ty env arg) (subst_ty env res)
subst_ty env (TyVar n)       = fromMaybe (TyVar n) (lookup n env)
subst_ty env (MetaTv tv)     = MetaTv tv
subst_ty env (TyCon tc)      = TyCon tc
subst_ty env (ForAll ns rho) = ForAll ns (subst_ty env' rho)
  where
    env' = [(n,ty') | (n,ty') <- env, not (n `elem` ns)]


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

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

class Outputable a where
  ppr :: a -> Doc

docToString :: Doc -> String
docToString = render

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

------------- 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 '\\' <> parens (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 <+> pprParendType ty

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

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

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

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


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

instance Outputable Type where
   ppr ty = pprType topPrec ty

instance Outputable MetaTv where
   ppr (Meta u _) = text "$" <> int u
```

```haskell
   --

instance Outputable TyVar where
   ppr (BoundTv n)    = text n
   ppr (SkolemTv n u) = text n <+> int u

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

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

precType :: Type -> Precedence
precType (ForAll _ _) = topPrec
precType (Fun _ _)    = arrPrec
precType _            = atomicPrec
        -- All the types are be atomic

pprParendType :: Type -> Doc
pprParendType ty = pprType tcPrec ty


pprType :: Precedence -> Type -> 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_type :: Type -> Doc         -- No parens
ppr_type (ForAll ns ty) = sep [text "forall" <+>
                                  hsep (map ppr ns) <> dot,
                               ppr ty]
ppr_type (Fun arg res)  = sep [pprType arrPrec arg <+> text "->",
                               pprType (arrPrec-1) res]
ppr_type (TyCon tc)     = ppr_tc tc
ppr_type (TyVar n)      = ppr n
ppr_type (MetaTv tv)    = ppr tv

ppr_tc :: TyCon -> Doc
ppr_tc IntT  = text "Int"
ppr_tc BoolT = text "Bool"
```