# Arbitrary-rank polymorphism in (GHC) Haskell

CAS 743

Stephen Forrest

20 March 2006

## Damas-Milner Type System

A *Damas-Milner* type system (also called *Hindley-Milner*) is a traditional type system for functional languages, and arose from an effort to do type reconstruction on the untyped $\lambda$-calculus.

An appealing property of this type system is the guarantee of a *most general unifier* for a set of type expressions, and the fact that the algorithm can deduce types without any type annotations supplied by the programmer. It is the basis of the type system for Haskell98.

However, the system does have limitations. We will see what these are, and how certain extensions to the system defined in GHC address these problems.

## Example 1

Consider the following snippet of Haskell code:

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

We apply `reverse` to both a list of booleans and a list of characters. We would like this to "just work", since `reverse` is quite capable of handling each, and return `([False, True], ['b', 'a'])`.

However, the Damas-Milner type system requires that the $\lambda$-bound variable `x` have a type that is *monomorphic*, or free of type variables.

We will generalize this to the idea of the *rank* of a type.

## Rank of a Type

The **rank** of a type describes the depth at which universal quantifiers appear in a *contravariant* position, i.e. to the left of a function arrow.

A rank-0 type has no universal quantifiers at all (it is a *monotype*). A function type has rank $n+1$ when its argument has rank $n$. Formally:

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

Following are some examples of the ranks of types:

| | |
|---|---|
| $\mathsf{Int} \to \mathsf{Int}$ | has rank 0 |
| $\forall a\ .\ a \to a$ | has rank 1 |
| $\mathsf{Int} \to (\forall a\ .\ a \to a)$ | has rank 1 |
| $(\forall a\ .\ a \to a) \to \mathsf{Int}$ | has rank 2 |
| $\forall a\ .(\ a \to a \to \mathsf{Int})$ | has rank 1 |

## Exploiting type annotations

Every term in a Damas-Milner type system is rank 1: there are no quantifiers on the left of a function arrow. As we've seen, this has limitations. However, it is known that pure type inference becomes difficult or intractable for rank $\geq 2$.

A system described by Odersky and Läufer (1996) addresses this by adding type annotations on terms to guide type inference. Peyton Jones, Shields, et al. describe the amount of annotation required by this system as "quite heavy", but suggest that many of the annotations could be inferred.

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

In the above, the signature of `f` serves to identify the type of `x`, removing the need for an explicit type annotation.

## The Subsumption Relation

Before we go too far in discussing polymorphism and instantiation of type variables, we must have a way to compare two type expressions.

In the usual Damas-Milner type system, we have the expected partial order defined by substitution into type variables, e.g. we have `[a] -> [a]` $<$ `[Int] -> [Int]`.

This generalizes to arbitrary ranks: $\sigma_1 < \sigma_2$ if $\sigma_1$ is more polymorphic than $\sigma_2$.

For higher-order ranks, we must watch out for the fact that function types are contravariant in the first argument. Hence $\sigma_1 < \sigma_2$ implies $\sigma_1 \rightarrow \text{Int} > \sigma_2 \rightarrow \text{Int}$.

## Predicativity

After we choose to permit polytypes inside function types, we soon face a question: do we allow type variables to be instantiated at polymorphic types?

**Example**: In GHC, we define the following

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

This is all fine, and `poly (\x -> x)` returns `(3, True)` as expected. However, suppose we also define

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

Consider the application `revapp (\x->x) poly`. For this to be legal, we would need to instantiate the type variable `a` in `revapp` with the polytype $\forall v \,.\, v \to v$.

## Predicativity

A type system which only allows a polymorphic function to be instantiated at a monotype is called **predicative**. A type systems which permits a polymorphic function to be instantiated at a polytype is called **impredicative**.

Understandably, predicative systems are much easier to deal with. The classical Damas-Milner system used in Haskell98 is predicative. So is the Odersky-Läufer system which is the basis of the Haskell implementation in this talk.

An example of an impredicative system is System F, which extends $\lambda$-calculus with type abstractions and type applications.

## Overview of Damas-Milner type inference rules

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

**Int:**

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}$$

**Var:**

$$\frac{}{\Gamma, (x : \sigma) \vdash x : \sigma}$$

**Abs:**

$$\frac{\Gamma, (x : \tau) \vdash t : \rho}{\Gamma \vdash (\backslash x.t) : (\tau \rightarrow \rho)}$$

**App:**

$$\Gamma \vdash t : \tau \rightarrow \rho$$
$$\frac{\Gamma \vdash u : \tau}{\Gamma \vdash tu : \rho}$$

**Let:**

$$\Gamma \vdash u : \sigma$$
$$\frac{\Gamma, x : \sigma \vdash t : \rho}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : \rho}$$

**Annot:**

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash (t :: \sigma) : \sigma}$$

**Gen:**

$$\overline{a} \notin ftv(\Gamma)$$
$$\frac{\Gamma \vdash t : \rho}{\Gamma \vdash t : \forall \overline{a}.\rho}$$

**Inst:**

$$\frac{\Gamma \vdash t : \forall \overline{a}.\rho}{\Gamma \vdash t : [\overline{a \mapsto \tau}]\rho}$$

## Syntax-directed form of Damas-Milner

An issue with the previous slide lies in the fact that the Gen and Inst rules may be applied at any time. We want a set of rules which we can transform into an algorithm. In fact, we can rewrite the rules (somewhat more verbosely) as a *syntax-directed* rule set, which transforms naturally into the Damas-Milner algorithm.

Among the tools necessary for this is an inference rule $\vdash^{subs}$ which is related to the subsumption relation defined earlier and is important for our ultimate generalization of the Damas-Milner method. There are three rules for $\vdash^{subs}$:

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

## Syntax-directed form of Damas-Milner

The task that these rules accomplish is the following: given $\sigma_1 = \overline{a} \,.\, \rho_1$ and $\sigma_2 = \overline{a} \,.\, \rho_2$, we are asked to prove that

$$\forall \bar{b}\; \exists \overline{a} \;\text{ such that } \rho_1 \leq \rho_2$$

The rule SKOL serves the purpose of instantiating the outermost type variables of $\sigma_2$ to arbitrary, completely fresh type constants, called skolem constants. If after this operation, it is still possible to match $\sigma_1$ against $\sigma_2$, then $\sigma_1$ is at least as polynorphic as $\sigma_2$.

## Odersky-Läufer type inference

We now direct our attention to the Odersky-Läufer type system (1996). The critical difference between the two is that in this new system, a polytype may appear *in both the argument and the result* of a function type, and therefore polytypes may be of *arbitrary rank*.

The new system differs from Damas-Milner in a number of ways:

In addition to the usual lambda-abstractions, we add a new sort of beast, an *annotated abstraction*, `\(`$x$`::`$\sigma$`) .` $t$, where the bound variable is annotated with a polytype. Along with this new structure we have a new rule, AABS.

We have replaced rule INST (instantiation) with subsumption (SUBS) to reflect the new generality of the operation.

## Overview of Odersky-Läufer type inference rules (1/2)

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

**Int:**

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}$$

**Var:**

$$\frac{}{\Gamma, (x : \sigma) \vdash x : \sigma}$$

**Abs:**

$$\frac{\Gamma, (x : \tau) \vdash t : \sigma}{\Gamma \vdash (\backslash x.t) : (\tau \to \sigma)}$$

**AAbs:**

$$\frac{\Gamma, (x : \sigma) \vdash t : \sigma'}{\Gamma \vdash (\backslash(x :: \sigma)\ .\ t) : (\sigma \to \sigma')}$$

**App:**

$$\Gamma \vdash t : (\sigma \to \sigma')$$
$$\frac{\Gamma \vdash u : \sigma}{\Gamma \vdash tu : \sigma'}$$

**Let:**

$$\Gamma \vdash u : \sigma$$
$$\frac{\Gamma, x : \sigma \vdash t : \rho}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : \rho}$$

## Overview of Odersky-Läufer type inference rules (2/2)

$\Gamma \vdash t : \sigma$

**Annot:**

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash (t :: \sigma) : \sigma}$$

**Fun:**

$$\frac{\vdash^{subs} \sigma_3 \leq \sigma_1 \quad \vdash^{subs} \sigma_2 \leq \sigma_4}{\vdash^{subs} (\sigma_1 \to \sigma_2) \leq (\sigma_3 \to \sigma_4)}$$

**Gen:**

$$\frac{\overline{a} \notin ftv(\Gamma) \quad \Gamma \vdash t : \rho}{\Gamma \vdash t : \forall \overline{a} . \rho}$$

**Subs:**

$$\frac{\Gamma \vdash t : \sigma' \quad \vdash^{subs} \sigma' \leq \sigma}{\Gamma \vdash t : \sigma}$$

**Skol:**

$$\frac{\overline{a} \notin ftv(\sigma) \quad \vdash^{subs} \sigma \leq \rho}{\vdash^{subs} \sigma \leq \forall \overline{a} \, . \, \rho}$$

**Spec:**

$$\frac{\vdash^{subs} [\overline{a \mapsto \tau}] \rho_1 \leq \rho_2}{\vdash^{subs} \forall \overline{a} \, . \, \rho_1 \leq \rho_2}$$

**Mono:**

$$\frac{}{\vdash^{subs} \tau \leq \tau}$$

## Problem with Generalizing Damas-Milner

We have the same problem with Odersky-Läufer that we did with the first presentation of Damas-Milner: the inference rules do not in themselves lead directly to an algorithm. The rule Gen allows generalization anywhere, and Subs allows specialization anywhere.

We might attempt to extend the Damas-Milner idea to this problem. The idea there is that we specialize every time a variable occurrence is encountered, and generalize in `let` expression. Peyton Jones and Shields give the following example. Suppose `f` has the type

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

Then consider the application `f (\x . \y . (x, y))` . We infer

$$(\backslash x.\backslash y.(x,y)) :: a \to b \to (a,b)$$

for some types $a$, $b$. Trying the obvious generalization to polymorphic arguments, we get

$$(\backslash x.\backslash y.(x,y)) :: \forall ab\ .\ a \to b \to (a,b)$$

The problem is that this is a rank 1 type, while our original type was rank 2, and is not true that

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

So we cannot allow ourselves to infer this type. Odersky and Läufer solve this problem by "early generalization" at every node in the syntax tree and then use the rule SUBS at every function application. This becomes quite expensive at runtime.

## Prenex-form Polytypes

Peyton Jones and Shields amend this flaw with the approach of Odersky and Läufer by a simplifying assumption. Consider the two functions matching these signatures:

$$\begin{array}{lll} h1 & :: & \forall a\ .\ a \rightarrow (\forall b\ .\ b \rightarrow (a,b)) \\ h2 & :: & \forall ab\ .\ a \rightarrow b \rightarrow (a,b) \end{array}$$

Without type annotations, these two functions are indistinguishable by any program context. In general, $(\sigma_1 \rightarrow (\forall a\ .\ \sigma_2))$ and $\forall a\ .\ (\sigma_1 \rightarrow \sigma_2)$ cannot be distinguished.

For this reason, one can simply choose to forbid the latter case. This is done by GHC, which automatically converts any instances of the latter into the former (which is in *prenex* form). This is called “for-all hoisting” (see Section 7.4.3.2 of the user manual).

## Propogating Types Inwards

A serious limitation to this design so far is the manner in which we introduced the higher-rank terms: a function can only have a higher rank if its argument has a polymorphic type, which is true (so far) only if it is explicitly annotated with a type.

For example, consider the following:

$$foo = (\backslash \mathsf{i}\ .\ (\mathsf{i}\ 3, \mathsf{i}\ \mathsf{True})) :: (\forall a\ .\ \to a) \to (\mathsf{Int}, \mathsf{Bool})$$

Peyton Jones and Shields argue it is “plain as a pikestaff” that `i` should have the type $\forall a\ .\ a \to a$. They employ a system of *partial type inference* introduced by Pierce and Turner, and construct a bidirectional version of the Odersky-Läufer method. This system is able to work from annotations present somewhere else in the expression to infer the type of both the $\lambda$-argument and the entire expression.

## Example 2

With this power, we can define Monads directly without recourse to classes:

```
data MonadT m = MkMonad \{ return :: forall a. a -> m a,
                           bind   :: forall a b. m a -> (a -> m b) -> m b
                         \}
```

The constructor MkMonad has the following type:

```
MkMonad :: forall m. (forall a. a -> m a)
                  -> (forall a b. m a -> (a -> m b) -> m b)
                  -> MonadT m
```

(Example from the Haskell User's Guide.)

## References

- Peyton Jones S, Weirich S, Vytiniotis D, Shields M, *Practical type inference for arbitrary-rank types*, to appear in the Journal of Functional Programming.

- GHC User Manual, section 7.4.9. (Arbitrary-rank polymorphism), accessed online 19 March 2006.

- Pierce BC, *Types and Programming Languages*, The MIT Press, Cambridge (Massachusetts), London (England), 2002.