

1 Explicit Language

PrenexF is a variant of System F, limited to rank-1 polymorphism by the prenex restriction. The syntax of PrenexF is shown in Fig.1.

Terms		
Polyterms		
$p ::= x$		Variable
$\Lambda \bar{a}.m$		Type lambda
Monoterms		
$m ::= m m$		Function application
$\lambda x : \tau. m$		Lambda abstraction
$[p \bar{\tau}]$		Type application
$\mathbf{let} x : \sigma = p \mathbf{in} m$		Let binding
Types		
Polytypes		
$\sigma ::= \forall \bar{a}. \tau$		Forall / Universal quantification
Monotypes		
$\tau ::= T$		Primitive type
a		Type variable
$\tau \rightarrow \tau$		Function

Figure 1: PrenexF

The most important detail is the stratification of terms into monoterms and polyterms, which matches the more conventional stratification of types - monoterms have monotypes, polyterms have polytypes. This makes an important aspect of the typing process clearly visible at a term level, rather than hiding it behind the let construct. Indeed, the fact let binds polyterms highlights the nature of let polymorphism.

Universal quantification in PrenexF is n -ary, and n may be 0 – this gives a uniform treatment of variables. Individual type lambdas and applications thus act as conversions from monoterms to polyterms and back.

Typing rules are in Fig.2. Where the syntax is interesting and at least minorly novel, the typing rules are rather predictable; this is very much in line with the goals of simplicity and understandability.

Unfortunately, much as PrenexF does a nice job of guiding intuitions, it is not pleasant for a programmer to work with. Not only are type annotations required on lambdas, but now the distinction between polymorphic and monomorphic must be explicitly managed. This creates quite a burden, and one that is entirely unnecessary.

To remedy this, clearly we are looking to perform some kind of type inference. More than this, in order to remove the burden of type lambdas and type applications we must infer terms to some extent too - rather, we must *elaborate* a program from a more comfortable source language into PrenexF. While this could be done by modifying a classic Hindley-Milner algorithm, I have chosen to

Polyterms:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^P x : \sigma} Var \qquad \frac{\Gamma, \bar{a} \vdash^m m : \tau}{\Gamma \vdash^P \Lambda \bar{a}. m : \forall \bar{a}. \tau} TLam$$

Monoterms:

$$\frac{\Gamma \vdash^m m_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash^m m_2 : \tau_1}{\Gamma \vdash^m m_1 m_2 : \tau_2} App \qquad \frac{\Gamma, x : \tau_1 \vdash^m m : \tau_2}{\Gamma \vdash^m \lambda x : \tau_1. m : \tau_1 \rightarrow \tau_2} Lam$$

$$\frac{\Gamma \vdash^P p : \sigma \quad \Gamma, x : \sigma \vdash^m m : \tau}{\Gamma \vdash^m \text{let } x = p \text{ in } m : \tau} Let \qquad \frac{\Gamma \vdash^P p : \forall \bar{a}. \tau_p}{\Gamma \vdash^m p [\bar{\tau}] : \tau_p [\bar{a} \mapsto \bar{\tau}]} TApp$$

Figure 2: Checking

feign ignorance and instead develop a source language that contains PrenexF. As the next two sections show, by using constraint systems and treating the process as one of generating a proof this turns out not only to be easy but also to yield proofs about the system that demonstrate important intuitions about the way Hindley-Milner inference works.

2 Elaboration Prerequisites

Before an elaborator can be written, there are certain issues worth tackling to keep things simple. In order to elaborate effectively, the elaborator must be type-aware. As the typechecker is going to use constraints, it is helpful to have a constraint-based presentation of the type system on hand - this also helps in choosing the constraint system in question!

Fig.3 gives us this, using the HERBRAND constraint system. Most rules change little, simply propagating the new constraint set C . The exception is App, where the premise has been relaxed and the conclusion adds an equality constraint to the constraint set in order to compensate. The programs for which the constraint set is satisfiable are thus those which can be typed according to Fig.2.

It would also be desirable, however, for the elaborator to be able to ask ‘does this typecheck if I replace its subterms with elaborated, checkable subterms for it?’. We can do this in two steps: first, the checking relation becomes an elaboration relation that yields a new term as well as a type. This produces judgements such as $C, \Gamma \vdash^m m \mapsto m' : \tau$, pronounced ‘given constraints C and context Γ , m elaborates to (or maps to) m' with monotone τ ’.

But this doesn’t give us room to elaborate the sub-terms: to do that we must pass the elaborator into the checker! This sadly gives us an even more complicated judgement - where previously we had \vdash^m , we now have $\vdash_{(j)}^m$. A

Polyterms:

$$\frac{x : \sigma \in \Gamma}{C, \Gamma \vdash^P x : \sigma} Var \qquad \frac{C, \Gamma, \bar{a} \vdash^m m : \tau}{C, \Gamma \vdash^P \Lambda \bar{a}. m : \forall \bar{a}. \tau} TLam$$

Monoterms:

$$\frac{C, \Gamma \vdash^m m_1 : \tau_f \quad C, \Gamma \vdash^m m_2 : \tau_p}{C \wedge (\tau_f = \tau_p \rightarrow \tau_r), \Gamma \vdash^m m_1 m_2 : \tau_r} App \qquad \frac{C, \Gamma, x : \tau_1 \vdash^m m : \tau_2}{C, \Gamma \vdash^m \lambda x : \tau_1. m : \tau_1 \rightarrow \tau_2} Lam$$

$$\frac{C, \Gamma \vdash^P p : \sigma \quad C, \Gamma, x : \sigma \vdash^m m : \tau}{C, \Gamma \vdash^m \text{let } x = p \text{ in } m : \tau} Let \qquad \frac{C, \Gamma \vdash^P p : \forall \bar{a}. \tau_p}{C, \Gamma \vdash^m p [\bar{\tau}] : \tau_p[\bar{a} \mapsto \bar{\tau}]} TApp$$

Figure 3: Checking with Constraints

similar use of the parameterised judgement ¹ would be \vdash_j^m - note the absence of parentheses around the subscript j . This gives us the ‘single step’ parameterised checker in Fig.4.

From here, it would be a good idea to develop an ‘elaborator’ that merely recovers the old typechecker - this is shown in Fig.5. The Fix rules act as a fixpoint, turning our single-step judgement into a recursive one, and the Check rules merely convert this into the same form as the rules in Fig.3.

As one further example, we could extend our fixpoint elaborator to handle lambdas without type annotations with a rule such as:

$$\frac{C, \Gamma, x : \tau \vdash_{(\text{fix})}^m \lambda x : \tau. m \mapsto m' : \tau'}{C, \Gamma \vdash_{\text{fix}}^m \lambda x. m \mapsto m' : \tau'} Lam?$$

This rule simply annotates the parameter’s type with a fresh metavariable, leaving it to the constraint system to resolve as checking continues.

It’s not difficult to show that the recovered checker above is the same as the constraint checker by fixing the parameter and replacing references to the \vdash_{fix} judgement with the precedents for the applicable rules. Adding rules to \vdash_{fix} preserves typing so long as the right-hand of \mapsto has been through the checking rules (or equivalently, would check cleanly) - though there may not be a unique solution to the constraints.

3 Inference

Fig. 6 contains syntax extensions to PrenexF for a language with type inference. The existence of unannotated lambdas is no great surprise. However, on the face of it allowing polyterms and monoterms to be intermingled completely contradicts the point of having them in the explicit calculus in the first place!

¹Or perhaps judgements, one for monotypes and one for polytypes

Polyterms:

$$\frac{x : \sigma \in \Gamma}{C, \Gamma \vdash_{(j)}^P x \mapsto x : \sigma} Var \qquad \frac{C, \Gamma, \bar{a} \vdash_j^m m \mapsto m' : \tau}{C, \Gamma \vdash_{(j)}^P \Lambda \bar{a}. m \mapsto \Lambda \bar{a}. m' : \forall \bar{a}. \tau} TLam$$

Monoterms:

$$\frac{C, \Gamma \vdash_j^m m_1 \mapsto m'_1 : \tau_f \quad C, \Gamma \vdash_j^m m_2 \mapsto m'_2 : \tau_p}{C \wedge (\tau_f = \tau_p \rightarrow \tau_r), \Gamma \vdash_{(j)}^m m_1 m_2 \mapsto m'_1 m'_2 : \tau_r} App$$

$$\frac{C, \Gamma, x : \tau_1 \vdash_j^m m \mapsto m' : \tau_2}{C, \Gamma \vdash_{(j)}^m \lambda x : \tau_1. m \mapsto \lambda x : \tau_1. m' : \tau_1 \rightarrow \tau_2} Lam$$

$$\frac{C, \Gamma \vdash_j^P p \mapsto p' : \sigma \quad C, \Gamma, x : \sigma \vdash_j^m m \mapsto m' : \tau}{C, \Gamma \vdash_{(j)}^m \text{let } x = p \text{ in } m \mapsto \text{let } x = p' \text{ in } m' : \tau} Let$$

$$\frac{C, \Gamma \vdash_j^P p \mapsto p' : \forall \bar{a}. \tau_p}{C, \Gamma \vdash_{(j)}^m p [\bar{\tau}] \mapsto p' [\bar{\tau}] : \tau_p[\bar{a} \mapsto \bar{\tau}]} TApp$$

Figure 4: Elaboration-ready Checking

$$\frac{C, \Gamma \vdash_{(fix)}^m m \mapsto m' : \tau}{C, \Gamma \vdash_{fix}^m m \mapsto m' : \tau} FixM \quad \frac{C, \Gamma \vdash_{(fix)}^P p \mapsto p' : \sigma}{C, \Gamma \vdash_{fix}^P p \mapsto p' : \sigma} FixP$$

$$\frac{C, \Gamma \vdash_{fix}^m m \mapsto m : \tau}{C, \Gamma \vdash_{chk}^m m : \tau} CheckM \quad \frac{C, \Gamma \vdash_{fix}^P p \mapsto p : \sigma}{C, \Gamma \vdash_{chk}^P p : \sigma} CheckP$$

Figure 5: Recovering the Old Checker

Thankfully this is not the case: rather, type lambdas and applications can now be elided and it is clear from the parsing context where they need to be generated as no other constructs can turn a monoterms into a polyterm or vice versa². This can lead us straight to instantiation and generalisation per Hindley-Milner – or to a slightly more flexible alternative.

Handling unannotated lambdas and missing type applications has something in common - the need to find the appropriate monotypes to fill in. In fact, missing applications can be treated as a special case of applications with too few parameters - and this is precisely what the rule *Inst* in Fig 7 does. *InfLam* is much the same rule as the *Lam?* example in the previous section, and *MCheckable* takes the role of *FixM*. *InfTApp* is similar, replacing missing parameters

²One could try using *let* to generate a monoterms from a polyterm, but then another monoterms is needed

Terms		
Polyterms		
$p ::= \dots$		m Monoterm
Monoterms		
$m ::= \dots$		$\lambda x.m$ Lambda abstraction (unannotated)
		p Polyterm

Figure 6: Extended Syntax for Inference

with metavariables. Note that the premises for `InfTApp` entail that the resulting term is checkable. We could in fact make the monotype part of the inferrer syntax-directed by modifying `MCheckable` to ignore type applications.

Polytypes are slightly trickier. The obvious solution is to generalise over all unconstrained metavariables in a type, but this creates a variable capture situation! Metavariables that came from an outer scope may yet collect constraints, but we must generalise without that knowledge because we need to know the arity of the polytype we're creating. Thankfully, those metavariables must all be present in the context - this is the source of the complication in the `InfTLam` rule. `Gen` is the counterpart to `Inst`, and `PCheckable` once again brings the checker into play.

There are a number of important properties that we can observe about the resulting inference system:

- Monoterm inference always yields the most general solution - this arises from the nature of the constraint system and the fact all constraints originate from the checker rather than the inferrer
- Polyterm inference always yields the most general solution - including further generalising polymorphic terms where possible
- Given a context containing no metavariables, the polyterm elaborator yields a term in which all metavariables are solved for - a direct consequence of the `InfTLam` rule

There are also some advantages over the traditional Hindley-Milner system. Users can introduce scoped type variables themselves and refer to them in annotations. Perhaps more interestingly, even where the elaborator failed it is possible to produce a partial elaboration with the failed goals and/or incompatible constraints marked out - combined with the constraint set this captures much of the information from the typing in a readable manner, allowing the user to see into the inference process.

Polytypes:

$$\begin{array}{c}
\frac{p \neq \Lambda \bar{a}.m}{C, \Gamma \vdash_{(\text{inf})}^{\text{P}} p \mapsto p' : \sigma} \text{PCheckable} \qquad \frac{C, \Gamma \vdash_{\text{inf}}^{\text{P}} \Lambda.m \mapsto p : \sigma}{C, \Gamma \vdash_{\text{inf}}^{\text{P}} m \mapsto p : \sigma} \text{Gen} \\
\\
\frac{C, \Gamma \vdash_{\text{inf}}^{\text{m}} m \mapsto m' : \tau \quad \bar{\tau} = fv(\tau) \setminus \setminus fv(\Gamma) \quad len(\bar{a}') = len(\bar{\tau})}{(C \wedge (\bar{\tau} = \bar{a}'), \Gamma \vdash_{\text{inf}}^{\text{P}} \Lambda \bar{a}.m \mapsto \Lambda(\bar{a} + \bar{a}').m' : \forall(\bar{a} + \bar{a}').\tau)} \text{InfTLam}
\end{array}$$

Monotypes:

$$\begin{array}{c}
\frac{C, \Gamma \vdash_{(\text{inf})}^{\text{m}} m \mapsto m' : \tau}{C, \Gamma \vdash_{\text{inf}}^{\text{m}} m \mapsto m' : \tau} \text{MCheckable} \\
\\
\frac{C, \Gamma, x : \tau \vdash_{(\text{inf})}^{\text{m}} \lambda x : \tau.m \mapsto m' : \tau'}{C, \Gamma \vdash_{\text{inf}}^{\text{m}} \lambda x.m \mapsto m' : \tau'} \text{InfLam} \\
\\
\frac{C, \Gamma \vdash_{\text{inf}}^{\text{P}} p \mapsto p' : \forall \bar{a}.\tau \quad len(\bar{\tau} + \bar{\tau}') = len(\bar{a})}{\Gamma \vdash_{\text{inf}}^{\text{m}} p[\bar{\tau}] \mapsto p'[\bar{\tau} + \bar{\tau}'] : \tau[\bar{a} \mapsto \bar{\tau} + \bar{\tau}']} \text{InfTApp} \\
\\
\frac{C, \Gamma \vdash_{\text{inf}}^{\text{m}} p[] \mapsto p'[\bar{a}] : \tau}{C, \Gamma \vdash_{\text{inf}}^{\text{m}} p \mapsto p'[\bar{a}] : \tau} \text{Inst}
\end{array}$$

Figure 7: Inference