FreezeML
Complete and Easy Type Inference for First-Class Polymorphism

Frank Emrich
The University of Edinburgh
frank.emrich@ed.ac.uk

Jan Stolarek
The University of Edinburgh
Lodz University of Technology
jan.stolarek@ed.ac.uk

Sam Lindley
The University of Edinburgh
Imperial College London
sam.lindley@ed.ac.uk

James Cheney
The University of Edinburgh
The Alan Turing Institute
jcheney@inf.ed.ac.uk

Abstract
ML is remarkable in providing statically typed polymorphism without the programmer ever having to write any type annotations. The cost of this parsimony is that the programmer is limited to a form of polymorphism in which quantifiers can occur only at the outermost level of a type and type variables can be instantiated only with monomorphic types.

The general problem of type inference for unrestricted System F-style polymorphism is undecidable in general. Nevertheless, the literature abounds with a range of proposals to bridge the gap between ML and System F by augmenting ML with type annotations or other features.

We present a new proposal, with different goals to much of the existing literature. Our aim is to design a minimal extension to ML to support first-class polymorphism. We err on the side of explicitness over parsimony, extending ML with two new features. First, λ- and let-bindings may be annotated with arbitrary System F types. Second, variable occurrences may be frozen, explicitly disabling instantiation.

The resulting language is not always as concise as more sophisticated systems, but in practice it does not appear to require a great deal more ink. FreezeML is a conservative extension of ML, equipped with type-preserving translations back and forth between System F. It admits a type inference algorithm, a mild extension of algorithm W, that is sound and complete and which yields principal types.

1 ML Magic
Consider the ML program: let \( f = \lambda x y. (x, y) \) in \( f \) 42 True. Hindley-Milner type inference [6, 12] relies on two pieces of implicit magic.

1. Generalisation, that is, saturating type abstraction, which only happens at let-bindings.
   \( f \) has type \( \forall a b. a \rightarrow b \rightarrow a \times b \)
2. Instantiation, that is, saturating type application, which only happens on variables.
   \( f \) is invoked with \( [a \mapsto \text{Int}, b \mapsto \text{Bool}] \)

These two features hide the boilerplate of languages with explicit first-class polymorphism like System F [4, 5].

2 The Perils of Instantiation
Whilst some argue that let-bound variables should not be generalised implicitly [18], instantiation is the bigger obstacle to type inference for first-class polymorphism because it throws away type information. In ML, because polymorphism may only occur at the top-level, variables must always be instantiated right away. Nothing is lost in instantiating eagerly, providing it happens at the correct types. As type variables can be instantiated only with monomorphic types, these can be inferred just by inspecting the program text.

Consider the following two functions.

\[ \text{id} : \forall a. a \rightarrow a \quad \text{single} : \forall a. a \rightarrow \text{List} a \]
\[ \text{id} x = x \quad \text{single} x = [x] \]

In ML the term single id can be assigned the type List \((T \rightarrow T)\) for any monomorphic type \(T\); once let-bound to a variable, we may then generalise to \(\forall a. \text{List} (a \rightarrow a)\). In a system with first-class polymorphism one might wish to suppress instantiation of \(\text{id}\), instead yielding List \((\forall a. a \rightarrow a)\). The quandary of type inference with first-class polymorphism is that both \(\forall a. \text{List} (a \rightarrow a)\) and List \((\forall a. a \rightarrow a)\) are fully general, and neither is an instance of the other. In fact, type inference, and indeed type checking, is undecidable for System F [21] without type annotations. Moreover, even in System F with type annotations, but no explicit instantiation, type inference remains undecidable [14]. As a consequence, the programmer must provide at least a modicum of explicit type information.

3 Prior Work
There is a plethora of work on bridging the gap between ML and System F: some systems stratify the type system, hiding polymorphism inside nominal types [7, 8, 13, 15]; others add features to the type system [9, 11, 16]; and others strive to stay within the System F type system whilst minimising the number of type annotations [2, 10, 17, 19, 20].

4 Freezing Variable Instantiation
Our proposal is modest. Having accepted that the programmer must provide explicit type annotations as a prerequisite,
we propose a system FreezeML in which the programmer can
furthermore explicitly choose whether or not to instantiate
a variable. For backwards compatibility with ML, the default
is to instantiate. For instance the term $\text{id}$ has type
$\text{List}(a \to a)$ (as in ML). On the other hand, the programmer
can instead elect to suppress instantiation. For instance, the
term single $\text{[id]}$ has type $\text{List}(\forall a.a \to a)$. The use of id has
been frozen. The freeze operator $\lceil-\rceil$ may only be applied to
variables. It has the effect of suppressing instantiation.

FreezeML extends ML with $\lceil-\rceil$ and explicit type annota-
tions on $\lambda$- and let-bindings. These extensions suffice to
express all of System F. There exist compositional type-
expressing translations back and forth between System F
and FreezeML. Moreover, there exists a sound and complete
type inference algorithm for FreezeML, a mild extension of
algorithm $\text{W}$ [1], that infers principal types.

5 $\lambda$-Bound Variables

Unlike in ML we can write lambda abstractions that use their
arguments polymorphically.

$$\text{poly} = \lambda(f : \forall a.a \to a).((f \ 42, f \text{ True})$$

To avoid the “swamp” [17] of undecidability and to keep
type inference compositional, we insist that unannotated
$\lambda$-bound variables be monomorphic. If we were to remove
the annotation from poly then in order to infer a type for $f$
we would have to inspect all uses together. One might hope
that even if we disallow such examples that rely on global
reasoning, it might still be safe to infer polymorphism when
it can be done locally. Consider the following two functions.

$$\text{bad1} = \lambda f. (\text{poly } [f], (f \ 42 + 1))$$
$$\text{bad2} = \lambda f. ((f \ 42 + 1, \text{poly } [f])$$

Assume type inference proceeds from left to right. In bad1 we
first infer that $f$ has type $\forall a.a \to a$ (as $[f]$ is the argument
to poly); then we may instantiate $a$ to Int when applying $f$
over 42. In bad2 we eagerly infer that $f$ has type $\text{Int} \to \text{Int}$;
now when we pass $[f]$ to poly, type inference fails. To rule
out this kind of sensitivity to the order of type inference, we
insist that unannotated $\lambda$-bound variables be monomorphic.

6 Explicit Generalisation

The freeze operator supports named polymorphic arguments.

$$\text{let } f = \lambda x.x \text{ in } \text{poly } [f]$$

With an explicit generalisation operator $\$ we can write:

$$\text{poly } (\lambda x.x)$$

Explicit generalisation is macro-expressive [3] in FreezeML.

$$\$ V \equiv \text{let } x = V \text{ in } [x]$$

We can also define a type-annotated variant:

$$\$^A V \equiv \text{let } (x : A) = V \text{ in } [x]$$

We elect to restrict generalisation to syntactic values; this
value restriction [22] is inessential.

7 Explicit Instantiation

Suppose head : $\forall a.\text{List} a \to a$ and ids : $\text{List}(\forall a.a \to a)$. We
can instantiate a term by binding it to a variable.

$$\text{let } x = \text{head ids in } x \ 42$$

With an explicit instantiation operator @ we can write:

$$\text{(head ids)} \@ 42$$

Explicit instantiation is macro-expressive in FreezeML.

$$M@ \equiv \text{let } x = M \text{ in } x$$

8 Freezing Let Generalisation

It is natural to ask whether, as well as suppressing instantia-
tion of variables, it is also possible (or necessary) to suppress
let generalisation; after all System F does neither. We write
$\text{[let]}$ to denote a “frozen” let binding that does not perform
generalisation. This is not directly definable as $(\lambda x.N) M$
in FreezeML, but we can macro-expressive frozen let as follows:

$$\text{[let]} x = M \text{ in } N \equiv \text{let } x = \text{id } M \text{ in } N$$

9 Is FreezeML Reasonable?

To be usable as a programming language FreezeML must
support reasoning principles. We write $M \simeq N$ to mean $M$
is observationally equivalent to $N$. At a minimum we expect
$\beta$-rules to hold, and indeed they do; the twist is that they
involve substituting a different value depending on whether
the variable being substituted for is frozen or not.

$$\text{let } x = V \text{ in } N \simeq N[\$ V / [x], \$ V @/x]$$
$$\text{let } (x : A) = V \text{ in } N \simeq N[\$^A V / [x], \$ V @/x]$$
$$\text{let } (x : A).M V \simeq M[V / [x], V @/x]$$
$$\text{let } (x : A).M V \simeq M[V / [x], V @/x]$$

In ML, due to the value restriction, reduction can change
the type of a subterm, but not the type of a program. FreezeML
is more subtle. For instance:

$$\text{let } x = (\lambda x.x) (\lambda x.x) \text{ in } [x] : a \to a$$
$$\text{let } x = \lambda x.x \text{ in } [x] : \forall a.a \to a$$

Thus:

$$M \simeq M' \iff \text{let } x = M \text{ in } N \simeq \text{let } x = M' \text{ in } N$$

However, this is what frozen let is good for, as:

$$M \simeq M' \iff \text{[let]} x = M \text{ in } N \simeq \text{[let]} x = M' \text{ in } N$$

Moreover, if $M$ is not a syntactic value, then:

$$\text{let } x = M \text{ in } N \simeq \text{[let]} x = M \text{ in } N$$

FreezeML is a convenient syntactic sugar for programming
System F. For reasoning (and defining a type-preserving
reduction semantics) it is preferable to think in terms of
$, @, [\lceil-\rceil], \text{[let]}. If we annotate these operators appropriately
then we obtain exactly System F.
References