FreezeML

Complete and Easy Type Inference for First-Class Polymorphism

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

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

Jan Stolarek
The University of Edinburgh
Lodz University of Technology
jan.stolarek@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: $\textbf{let} f = \lambda x\, y.(x, y)\ \textbf{in}\ f\ 42\ \mathsf{True}$. 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 $\forall a\, b.a \to b \to a \times b$)
2. *Instantiation*, that is, saturating type application, which only happens on variables.
   (f is invoked with $[a \mapsto \mathsf{Int},\ b \mapsto \mathsf{Bool}]$)

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.

$$\begin{array}{ll} \mathsf{id} : \forall a.a \to a & \mathsf{single} : \forall a.a \to \mathsf{List}\ a \\ \mathsf{id}\ x = x & \mathsf{single}\ x = [x] \end{array}$$

In ML the term $\mathsf{single}\ \mathsf{id}$ can be assigned the type $\mathsf{List}\ (T \to T)$ for any monomorphic type T; once let-bound to a variable, we may then generalise to $\forall a.\mathsf{List}\ (a \to a)$. In a system with first-class polymorphism one might wish to suppress instantiation of id, instead yielding $\mathsf{List}\ (\forall a.a \to a)$. The quandary of type inference with first-class polymorphism is that both $\forall a.\mathsf{List}\ (a \to a)$ and $\mathsf{List}\ (\forall a.a \to a)$ 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 single id has type List $(a \to a)$ (as in ML). On the other hand, the programmer can instead elect to suppress instantiation. For instance, the term single $\lceil \mathsf{id} \rceil$ has type 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 annotations on λ- and let-bindings. These extensions suffice to express all of System F. There exist compositional type-preserving 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 W [1], that infers principal types.

5 λ-Bound Variables

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

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

To avoid the "swamp" [17] of undecidability and to keep type inference compositional, we insist that unannotated λ-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.

$$\begin{aligned} \mathsf{bad1} &= \lambda f.(\mathsf{poly}\ \lceil f \rceil, f\ 42) \\ \mathsf{bad2} &= \lambda f.(f\ 42, \mathsf{poly}\ \lceil f \rceil) \end{aligned}$$

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

6 Explicit Generalisation

The freeze operator supports named polymorphic arguments.

$$\mathbf{let}\ f = \lambda x.x\ \mathbf{in}\ \mathsf{poly}\ \lceil f \rceil$$

With an explicit generalisation operator \$ we can write:

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

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

$$\$V \equiv \mathbf{let}\ x = V\ \mathbf{in}\ \lceil x \rceil$$

We can also define a type-annotated variant:

$$\$^A V \equiv \mathbf{let}\ (x : A) = V\ \mathbf{in}\ \lceil x \rceil$$

We choose to restrict generalisation to values as FreezeML adopts the ML value restriction [22].

7 Explicit Instantiation

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

$$\mathbf{let}\ x = \mathsf{head}\ \mathsf{ids}\ \mathbf{in}\ x\ 42$$

With an explicit instantiation operator @ we can write:

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

Explicit instantiation is macro-expressible in FreezeML.

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

8 Freezing Let Generalisation

It is natural to ask whether, as well as suppressing instantiation of variables, it is also possible (or necessary) to suppress let generalisation; after all System F does neither. We write $\lceil \mathbf{let} \rceil$ to denote a "frozen" let binding that does not perform generalisation. We can macro-express frozen let in FreezeML.

$$\lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N \equiv \mathbf{let}\ x = \mathsf{id}\ M\ \mathbf{in}\ N$$

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 β-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.

$$\begin{aligned} \mathbf{let}\ x = V\ \mathbf{in}\ N &\simeq N[\$V\ /\ \lceil x \rceil,\ V@\ /\ x] \\ \mathbf{let}\ (x : A) = V\ \mathbf{in}\ N &\simeq N[\$^A V\ /\ \lceil x \rceil,\ V@\ /\ x] \\ (\lambda x.M)\ V &\simeq M[V\ /\ \lceil x \rceil,\ V\ /\ x] \\ (\lambda(x : A).M)\ V &\simeq M[V\ /\ \lceil x \rceil,\ V@\ /\ x] \end{aligned}$$

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:

$$\begin{aligned} &\mathbf{let}\ x = (\lambda x.x)\,(\lambda x.x)\ \ \mathbf{in}\ \lceil x \rceil : a \to a \\ &\mathbf{let}\ x = \lambda x.x\ \ \mathbf{in}\ \lceil x \rceil : \forall a.a \to a \end{aligned}$$

Thus:

$$M \simeq M' \not\Longrightarrow \mathbf{let}\ x = M\ \mathbf{in}\ N \simeq \mathbf{let}\ x = M'\ \mathbf{in}\ N$$

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

$$M \simeq M' \Longrightarrow \lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N \simeq \lceil \mathbf{let} \rceil\ x = M'\ \mathbf{in}\ N$$

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

$$\mathbf{let}\ x = M\ \mathbf{in}\ N \simeq \lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N$$

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 \$, @, $\lceil - \rceil$, $\lceil \mathbf{let} \rceil$. If we annotate these operators appropriately then we obtain exactly System F.

References

[1] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212.

[2] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. ACM, 429–442.

[3] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.

[4] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination de coupures dans l'arithméticque d'ordre supérieur*. Thèse de doctorat d'état. U. Paris VII.

[5] Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types*. Cambridge University Press. http://www.paultaylor.eu/stable/Proofs&Types.html

[6] J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc* 146 (1969), 29–60.

[7] Mark P. Jones. 1997. First-class Polymorphism with Type Inference. In *POPL*. ACM Press, 483–496.

[8] Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1411–1430.

[9] Didier Le Botlan and Didier Rémy. 2003. ML^{F}: raising ML to the power of system F. In *ICFP*. ACM, 27–38.

[10] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *ICFP*. ACM, 283–294.

[11] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *POPL*. ACM, 66–77.

[12] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375.

[13] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *POPL*. ACM Press, 54–67.

[14] Frank Pfenning. 1993. On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundam. Inform.* 19, 1/2 (1993), 185–199.

[15] Didier Rémy. 1994. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *TACS (Lecture Notes in Computer Science)*, Vol. 789. Springer, 321–346.

[16] Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-class Polymorphism for ML. In *ML*. ACM, 3–14.

[17] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *PLDI*. ACM, 783–796.

[18] Dimitrios Vytiniotis, Simon L. Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *TLDI*. ACM, 39–50.

[19] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*. ACM, 251–262.

[20] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *ICFP*. ACM, 295–306.

[21] J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *LICS*. IEEE Computer Society, 176–185.

[22] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.