Type inference with polymorphic recursion and semi-unification

Kristian Magnani dos Santos, João Rafael Moraes Nicola

30th June 2004

Abstract

This paper describes the Milner-Mycroft type system, which extends the traditional Damas-Milner type system by including polymorphic recursion. An introduction to type systems and the type inference problem is made, followed by the description of the Milner-Mycroft type system and the related semi-unification problem, along with a semi-decision algorithm for the semi-unification problem.

1 Introduction

Most programming languages provides the notion of *type* as a fundamental abstraction for programming. A typed language classifies some (or all) of its constructs (data, functions, procedures, classes, etc.) according to a *type system*. A type system is a tractable method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute [Pie02].

In the past, type systems were not studied *per se*, but were considered as just a small detail in the design of a language. Since the 1970s, type systems have been drawing increasing interest in the research community. Today, type systems are a central aspect of many programming languages, specially among declarative languages, such as functional and logic languages.

One of the most regarded improvements brought by the research in type systems is *type inference* (or *type reconstruction*). Instead of requiring the programmer to annotate the type of variables, parameters, function return values, etc., a programming language with type inference is able to reconstruct the type of all valid expressions (and identify type errors in invalid expressions) without any type annotation of the programmer.

This paper describes the Milner-Mycroft type system, which extends the traditional Damas-Milner type system by including polymorphic recursion. Type inference for this type systems was shown to be equivalent to the *finite semi-unification* problem in [Hen89]. This problem is also described in this paper, along with a semi-decision algorithm for it.

Some programming languages that make use of the Milner-Mycroft type system are: Haskell, Miranda, Clean, Mercury and a variety of the SML ([EL]). The first three languages require the programmer to annotate the type of a function defined with polymorphic recursion, while the later two use polymorphic recursion type inference.

Although type inference for the Milner-Mycroft type system is theoretically undecidable (as shown in [AJK93]), real usage suggests that it is very practical. Compiler implementors for languages with this type system have not yet reported an example of an expression for which the algorithm described in [Hen93] does not halt. The paper describes also a conjuncture proposed in [Hen93] to explain this apparent paradox.

2 Type systems

2.1 The Damas-Milner type system

The *Damas-Milner* is a polymorphic type inference system which was first studied by Damas and Milner, and it represents the type of polymorphism found in some functional languages like ML and Haskell. In this type system, a recursive polymorphic function can be used within its own body only monomorphically. The terms of the Damas-Milner Polymorphic λ-Calculus are defined by

$$\begin{array}{rcl} t & ::= & x \mid \lambda x:\tau.t \mid t\ t \\ & \mid & \textbf{let}\ x = t\ \textbf{in}\ t \\ & \mid & \textbf{fix}\ x.t \end{array}$$

and the type expressions are given by

$$\begin{array}{rcll} \tau & ::= & \alpha \mid \tau \to \tau & \textit{(simple types)} \\ \sigma & ::= & \tau \mid \forall\alpha.\sigma & \textit{(type schemes)} \end{array}$$

The reductions of the terms are

$$\begin{array}{rcl} (\lambda x.t_1)\, t_2 & \rightsquigarrow & [x \mapsto t_2]\, t_1 \\ \textbf{let}\ x = t_2\ \textbf{in}\ t_1 & \rightsquigarrow & [x \mapsto t_2]\, t_1 \\ \textbf{fix}\ x.t & \rightsquigarrow & [x \mapsto \textbf{fix}\ x.t]\, t \end{array}$$

Its rules of type inference can be viewed in Figure 1. Note that the rule for the **fix** construction, which defines recursive functions, has a *simple type* associated with, instead of a *type scheme*. This constraint assures that all occurrences of x in t have the same type τ. The implication of that rule is illustrated by the example given in Figure 2, taken from [HK04], where a trial program written in Haskell is presented, accompanied by the error message reported by the compiler when trying to infer the type of the constructions. The error is raised when the type checker tries to infer the type of the `compose` function. As it is part of a triple which is recursively defined, it can be used just monomorphically inside its definition. But from the first occurrence of `compose` it can be inferred that its type is `Int` $\to$ `Int`, while from its second occurrence the type inferred is `Double` $\to$ `Double`, leading to a type inference conflict, as reported by the type checker.

$$\frac{\Gamma \vdash t : \forall\alpha.\sigma}{\Gamma \vdash t : [\alpha \mapsto \tau]} \quad \textit{(INST)}$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\, t_2\ :\ \tau_1} \quad \textit{(APPL)}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \alpha \text{ not free in } \Gamma}{\Gamma \vdash t : \forall\alpha.\sigma} \quad \textit{(GEN)}$$

$$\frac{\Gamma, x : \tau_2 \vdash t : \tau_1}{\Gamma \vdash \lambda x.t : \tau_2 \mapsto \tau_1]} \quad (ABS)$$

$$\frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash t_2 : \sigma_2}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : \sigma_2} \quad \textit{(LET)}$$

$$\frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \textbf{fix}\ x.t : \tau} \quad (FIX - M)$$

Figure 1: Damas-Milner Type Inference System.

```
intFunc :: Int → Int
intFunc x = x + 1
doubleFunc :: Double → Double
doubleFunc x = sqrt x
myPair = let (compose, foo, goo) =
             (λf → λy → f (f y)
              λv → compose intFunc v,
              λw → compose doubleFunc w)
         in (foo 3, goo 16.0)
```

GHC Type Checker reports

```
Couldn't match 'Double' against 'Int'
    Expected type: Int → Int
    Inferred type: Double → Double
In the first argument of 'compose',
namely 'doubleFunc'
In a lambda abstraction:
    λw → compose doubleFunc w
```

Figure 2: An example in Haskell where the Damas-Milner Type Inference System fails in inferring a type for the constructions.

2.2 The Milner-Mycroft type system

The Milner-Mycroft Polymorphic λ-Calculus differs from the Damas-Milner by the substitution of the recursive function definition rule, *FIX-M*, by the rule *FIX-P*, which allows polymorphic recursion. The new rule is presented in Figure 3. The modified typing rules allow the program presented in Figure 2 to be typed. In this case, the type of the `compose` function would be $\forall a.\, a \rightarrow a$.

Polymorphic recursion usually appears when dealing with non-uniform polymorphic data structures ([HK04, Oka98]). The usage of polymorphic recursion in these cases can reduce the time complexity of some operations over some kinds of tree and list structures, and can guarantee some useful invariants on those data structures.

The Milner-Mycroft type inference problem is equivalent to the finite semi-unification problem, as demonstrated in [Hen93]. The reduction of Milner-Mycroft typability to semi-unification is out of the scope of this paper, but the reader can find it in [Hen93].

The following sections are devoted to the description of finite semi-unification, along with an semi-algorithm for it.

$$\frac{\Gamma, x : \sigma \vdash t : \sigma}{\Gamma \vdash \mathbf{fix}\, x.t : \sigma} \quad \textit{(FIX-P)}$$

Figure 3: Milner-Mycroft Type Inference Rule for recursive definitions.

3 The finite semi-unification problem

A **first order signature** Σ is a tuple (F, ar) F is a set of function symbols and $ar : F \mapsto \mathbb{N}$ is a function that assigns an arity to each function symbol $f \in F$. Given an infinite set V of variables, we define the set $T_\Sigma(V)$ of terms over V with signature Σ by the following definition:

$$\begin{array}{rcl} V & \subseteq & T_\Sigma(V) \\ t_1, \ldots, t_n \in T_\Sigma(V) & \wedge & f \in F \wedge \\ ar(f) = n & \rightarrow & f(t_1, \cdots, t_n) \in T_\Sigma(V) \end{array}$$

The function $Var : T_\Sigma(V) \mapsto \mathcal{P}(V)$ identifies the set of variables present in a given term, and is defined by induction over the structure of $T_\Sigma(V)$ as

$$\begin{array}{rcl} Var(v) & = & \{v\},\ v \in V \\ Var(f(t_1, t_2, \ldots, t_n)) & = & \bigcup_{i=1}^{n} Var(t_i) \end{array}$$

A substitution $S : V \mapsto T_\Sigma(V)$ is a function which assigns to each variable in V a term in $T_\Sigma(V)$. The usual notation for substitutions is to represent the function $S(x) = \begin{cases} t & , x = a \\ x & , x \neq a \end{cases}$ as $S = [a \mapsto t]$. This definition can be extended to define a substitution $S_T : T_\Sigma(V) \mapsto T_\Sigma(V)$:

$$\begin{array}{rcl} S_T(v) & = & S(v),\ v \in V \\ S_T(f(t_1, \ldots, t_n)) & = & f(S_T(t_1), \cdots, S_T(t_n)) \end{array}$$

A finite semi-unification system is a tuple (Σ, V, E, I), where Σ and V are as defined above, $E \subseteq T_\Sigma(V) \times T_\Sigma(V)$ is a finite set of equations and $I \subseteq T_\Sigma(V) \times \mathbb{N} \times T_\Sigma(V)$ is a finite set of equations defined over terms. A finite semi-unification system is solvable if and only if substitutions $S, S_0, S_1, \cdots, S_n$ can be found such that, for all equations $t_1 \equiv t_2 \in E$, $S(t_1) = S(t_2)$ and, for all inequations $t_1 \leq_i t_2 \in I$, $S_i(S(t_1)) = S(t_2)$.

4 A semi-decision algorithm for the semi-unification problem

Henglein proposed a semi-decision algorithm for the finite semi-unification problem ([Hen93]), which consists in a rewriting relation defined over the set of finite semi-unification systems which halts for all solvable and for some unsolvable semi-unification systems. This algorithm is depicted in figure 4. The following conventions are used in the algorithm description.

Symbols $x, y, z, x_1, \ldots, x_i$ range over V, t, t_1, t_2, $\ldots$, r, r_1,r_2, $\ldots$, range over $T_\Sigma(V)$, equations are represented as $t_1 = t_2$ and inequations as $t_1 \leq_i t_2$, where $t_1, t_2 \in T_\Sigma(V)$

$$
\begin{array}{rcl}
\{f(t_1,\ldots,t_k) = f(r_1,\ldots,r_k)\} \cup \Gamma & \Rightarrow & \left(\bigcup_{i=1}^{k}\{t_i = r_i\}\right) \cup \Gamma \\
\{f(t_1,\ldots,t_k) = g(r_1,\ldots,r_k)\} \cup \Gamma & \Rightarrow & \Box\text{(symbol clash)} \qquad (f \neq g) \\
\{f(t_1,\ldots,t_k) = r\} \cup \Gamma & \Rightarrow & \{r = f(t_1,\ldots,t_k)\} \cup \Gamma \\
\{x = f(t_1,\ldots,t_k)\} \cup \Gamma & \Rightarrow & \Box\text{(occurs check)} \qquad \left(x \in \bigcup_{i=1}^{k}(Var(t_i))\right) \\
\{x = t\} \cup \Gamma & \Rightarrow & [x \mapsto t]\Gamma \qquad (x \notin Var(t)) \\
\{x = x\} \cup \Gamma & \Rightarrow & \Gamma \\
\{f(t_1,\ldots,t_k) \leq_i f(r_1,\ldots,r_k)\} \cup \Gamma & \Rightarrow & \left(\bigcup_{j=1}^{k}\{t_j \leq_i r_j\}\right) \cup \Gamma \\
\{x \leq_i t\} \cup \{x \leq r\} \cup \Gamma & \Rightarrow & \{x \leq_i t\} \cup \{t = r\} \cup \Gamma \\
\{f(t_1,\ldots,t_k) \leq_i x\} \cup \Gamma & \Rightarrow & \Box\text{(extended occurs check)} \\
 & & \left(Clo(x,\Gamma) \cap \left(\bigcup_{i=1}^{k} Var(t_i)\right) \neq \emptyset\right) \\
\{f(t_1,\ldots,t_k) \leq_i x\} \cup \Gamma & \Rightarrow & \{f(t_1,\ldots,t_k) \leq_i x\} \cup \{x = f(x_1,\ldots,x_k)\} \cup \Gamma \\
 & & \left(Clo(x,\Gamma) \cap \left(\bigcup_{i=1}^{k} Var(t_i)\right) = \emptyset\right) \wedge \\
 & & \forall i \forall j (1 \leq i, j \leq k \wedge i \neq j \rightarrow x_i \neq x_j) \wedge \\
 & & \{x_1,\ldots,x_k\} \not\subseteq Var(\{f(t_1,\ldots,t_k) \leq_i x\} \cup \Gamma)
\end{array}
$$

Figure 4: Semi-unification rewriting relation

and $i \in \mathbb{N}$. The symbols Γ represents a set of equations and inequations.

The expression $\{t_1 = t_2\} \cup \Gamma_1$ and $\{t_1 \leq_i t_2\} \cup \Gamma_1$ represent, respectively, semi-unification systems containing the equation $t_1 = t_2$ and the inequation $t_1 \leq_i t_2$, along with equations and inequations present in Γ_1. The rewriting relation is represented by the symbol $\Rightarrow$. The symbol $\Box$ represent an unsolvable semi-unification system.

A substitution $S : V \mapsto T_\Sigma(V)$ can be applied to a set of inequations and equations, resulting in the same set with all the terms modified by the substitution.

The inequality closure $Clo : V \times \mathcal{P}(E \cup I) \mapsto \mathcal{P}(V)$, where E and I are as defined in the previous section, is defined as follows:

$$Clo(x, \Gamma) = \{x\} \cup$$
$$\{z \in V | \exists y \exists i (y \in Clo(x, \Gamma) \land \{y \leq_i z\} \subseteq \Gamma\}$$

5 Practicality of polymorphic recursion

No example of non-halting type inference has yet been found for the Milner-Mycroft type system in practice. This behaviour is similar to the behaviour of the Damas-Milner type system: although type inference for the Damas-Milner type system is exponential, its real usage shows polynomial behaviour.

Henglein ([Hen93]) conjectured that the apparent decidability (and polynomial complexity) of type inference for the Milner-Mycroft type system is due to the fact that programmers usually define functions whose types are smaller than the body of the function itself (small types), rather than functions with very complex types (Big types). However, this problem is still open.

6 Conclusion

This paper presented a survey of the current research about polymorphic recursion type inference and semi-unification. The Milner-Mycroft type system was described, an algorithm for semi-unification was shown, and the theoretical undecidability versus practical decidability behaviour of the problem was discussed.

For further information about polymorphic recursion, the reader is referred to [Hen93]. A general discussion about semi-unification (including the finite case) is found in [JK93].

References

[AJK93] Pawel Urzyczyn A. J. Kfoury, Jerzy Tiuryn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):290–311, April 1993.

[EL] Martin Emms and Hans LeiSS. Extending the type checker of sml by polymorphic recursion: A correctness proof.

[Hen89] Fritz Henglein. Polymorphic type inference and semi-unification. Technical Report 443, New York, N.Y., 1989.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, Abril 1993.

[HK04] Joseph J. Hallett and Assaf J. Kfoury. Programming examples needing polymorphic recursion. Technical Report BUCS-TR-2004-004, Department of Computer Science, Boston University, January 2004.

[JK93] Said Jahama and A. J. Kfoury. A general theory of semi-unification. Technical Report 1993-018, 1993.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.