Polymorphic Typing for Call-By-Name Semantics

B. Monsuez

Laboratoire d'Informatique de l'École Normale Supérieure. (LIENS)
45, rue d'Ulm 75005 Paris, FRANCE
monsuez@dmi.ens.fr
(Extended Abstract)

1 Introduction

The ML-polymorphic type system is sound, efficient, and supports a quite rich and
d Flexible type system. Since this is one of the most efficient algorithms that infer types
in the absence of any type declaration, the ML-type system is suitable for interactive
languages. Consequently, it has been used in other languages like Hope [BuMcQ80],
Miranda [Tur85] or Prolog [MyOK84]. But we should not conclude that the ML-type
system is the definitive answer: for lazy-languages, ML-types are too restrictive since
they do not preserve β-reduction. If we compute the ML-type of the expression \( S K \)
and its β-reduced form \( "\lambda xy.y" \), we find out that:

\[
\begin{align*}
S &: \lambda x y z . (x y)(x z) \\
&: (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma.
\end{align*}
\]

\[
\begin{align*}
K &: \lambda x y . x \\
&: \alpha \to \beta \to \alpha.
\end{align*}
\]

\[
\begin{align*}
S \ K &: (\alpha \to \beta) \to \alpha \to \alpha,
\text{ but } \lambda x y . y &: \alpha \to \beta \to \beta.
\end{align*}
\]

Much theoretical work has been done to define more generous type systems that
avoid this kind of restrictions. Coppo [Cop80] showed how to define a type system
which is much more flexible than ML’s, and preserves β-reduction, but this type
system is undecidable. As a consequence of the lack of appropriate type systems, lazy
languages like LML [Joh87], or GAML [Ma91] still use Milner’s algorithm which is
better suited for a call-by-value semantic.

The purpose of this work is to show that, despite the appearances the evaluation
mechanism (call-by-name or call-by-value) has a great influence on the type inference
system you can construct for a particular language.

For traditionally statically typed languages that include type constraints as part
of their definition, the language-evaluation mechanism is immaterial for the purpose
of type-checking. The language designer distinguishes between two distinct semantic
levels: the dynamic semantics describing the runtime evaluation process, and the static
semantics describing types with respect to the dynamic semantics.

It is not the only way to interpret type inference algorithms. Another way is to
consider type inference as an abstract interpretation of the dynamic semantics of the
language. Some work has already been done by Jones and Mycroft. In [JoMy85],
they showed that the Hindley/Milner type system is an abstract interpretation of
the untyped λ-calculus. More recently, A. Aiken and B. Murphy used the formalism
of abstract interpretation to define a type inference algorithm for FL [AiMu91]. In
[Mon92a], we show how to use abstract interpretation in a generic way to define type
systems, and how to use widening operators [Co78] to ensure algorithm termination. In the same paper, we show how to retrieve the ML-type algorithm using this generic method.

In this paper, we construct a more precise type system for call-by-name languages. We use the same abstraction scheme as the one described in [Mon92a], but instead of starting from a call-by-value semantics we start from a call-by-name semantics. Moreover we show that the Hindley/Milner type inference algorithm is an abstract interpretation of this larger type system. As a consequence, this type inference algorithm is more generous\(^1\) and is therefore better-suited for lazy languages.

2 Type inference by Abstract Interpretation

With respect to the formalism of abstract interpretation, a type is an abstraction of a set of values ensuring an error-free execution.

On the one hand, a type can be interpreted as an upper approximation. In ML for instance, the expression \(\lambda x.x + 1\) has the type "num\(\rightarrow\)num". But the most precise description of the behavior of a function is given by its functional graph: \(\{(x, x + 1) : x \in \text{num}\}\). Now we observe that the functional graph is strictly included in \(\text{num} \times \text{num}\), and therefore we can conclude that "num\(\rightarrow\)num" is an upper approximation of the behavior of the function.

On the other hand, a type cannot always be interpreted as an upper approximation. The ML-expression \((\lambda f.x.f(fx))(\alpha.0)\) has the type "num\(\rightarrow\)num". Since this function returns the numerical value 0 for any kind of arguments, this type is certainly not the most general. The previous expression has the far more general type: "\(\alpha \rightarrow \text{num}\)". This time, we observe that the ML principal type is a subset of a more general type.

The framework of abstract interpretation requires an abstract value to be either an upper or lower approximation. In [Mon92a], we have shown that characterizing a type as a well-behaved abstract value is much more subtle. We split the abstraction into two steps (i.e. an upper approximation followed by a lower approximation). First we define a huge type system \(T_g\) whose elements are the closest upper approximation of the set of values insuring an error-free execution. These general types are not effective in general, so that we can neither infer types nor perform type-checking. In order to define a useful type system, we have to restrict ourselves to less general types. Thus, we have to abstract the general types to smaller type—like ML-types.

To summarize, a type is an abstract value obtained by combining two pairs of abstraction functions. The first one is an upper approximation, the second one is a dual lower approximation.

\[
\alpha \quad \alpha^* \\
\text{Values} \models T_g \models T_{ML} \\
\gamma \quad \gamma^*
\]

In general, the lattice of abstract types does not verify the decreasing chain condition (any strictly decreasing chain is finite). The main consequence is that in the

\(^1\) The types are still elements of the ML-type algebra, but their meaning is quite different.