A Type Inference System for SETL

Fritz Henglein
Courant Institute of Mathematical Sciences
New York University
251 Mercer St.
New York, N.Y. 10012, USA
Internet: henglein@nyu.nyu.edu

June 19th, 1987

Abstract

We present a type inference system that should function as the basis for a strong typing discipline in SETL. It extends the polymorphism found in languages like ML in several respects. In particular, it handles unioned and implicit recursive types, and provides a form of restricted polymorphism which is derived from SETL's dynamic overloading.

1 Introduction

In this paper we present a type inference system for SETL. The reader is assumed to be familiar with the notions of weak typing and strong typing, type checking and type inference, type inference systems and typing algorithms. For a background on these topics [CW85] and [Car85] are sufficient.

We have attempted to separate the type inference system from the particular *type model* (i. e. what is considered the types in a language) as much as possible. We only assume that the types have a lattice structure. How this lattice is defined is part of the type model. In combination with the well-known parametric polymorphism from ML [Mil78] this represents a novel integration of a subtype discipline [Rey85] with parametric polymorphism.

The type model, described in a companion paper, encompasses union types, implicit recursive types, and finitary polymorphism. The latter is a form of restricted parametric polymorphism derived from the dynamic overloading discipline in current SETL (see [Hen87] for an exposition of SETL overloading).

The following section contains the core of the type inference system. Of course it is not complete for the whole language, but it contains all the crucial elements.

2 The Inference System

The reader is referred to [Lei83] and [Mit84b] for a more thorough discussion of the following concepts.

The type inference system is given as a collection of *rules*. Every rule consists of an *antecedent* and a *consequent*. The rules are to be read "Given the antecedent of a rule is true, then the consequent in the same rule is also true." The only statements that can occur in such rules are *typings* and *type containments*. A typing is a statement of the following form:

$$\mathcal{A} \vdash e : \tau$$

e is an expression or any other well-formed syntactic unit in the programming language, τ is a type, and finally, since the type of an expression is dependent on the types of the unbound identifiers in it, $\mathcal{A}$ represents a set of assumptions on the types of identifiers in e.

We present the rules in groups. Every rule is explained, an example for it given, and some remarks for clarification may be added.

2.1 Application

$$\frac{\begin{array}{l}\mathcal{A} \vdash f : \sigma \to \tau \\ \mathcal{A} \vdash e : \sigma' \\ \sigma' \leq \sigma\end{array}}{\mathcal{A} \vdash f(e) : \tau} \tag{1}$$

Given f of function type $\sigma \to \tau$ with domain type σ and codomain type τ and an argument e with a subtype σ' of σ, the type of $f(e)$ is simply the codomain type of f.

Because of the presence of a containment statement, this rule cannot be supplanted by simply providing a primitive operator **apply** with the type $\Delta\sigma.\Delta\tau.(\sigma \to \tau) \times \sigma \to \tau$ [1] since this operator would require the argument of a function f to have the exact same type as the domain type of f whereas the above rule allows for a *coercion*, that is implicit "upwards" type conversion [2], of the argument into the domain type σ as long as its type σ' is a subtype of σ. This extension is crucial in a system with union types. Note that we *cannot* add a universal coercion rule

$$\frac{\begin{array}{l}\mathcal{A} \vdash e : \sigma \\ \sigma \leq \sigma'\end{array}}{\mathcal{A} \vdash e : \sigma'}$$

to overcome this shortcoming of **apply** since the left-hand sides of assignments should never be coerced. This phenomenon is in marked contrast with functional languages. Our type inference system facilitates specification of the local context in which coercions are sound and thus permitted.

[1] This type expresses that **apply** has two arguments, one with a function type, the other with the domain type of the first argument; its result type is the codomain type of the first argument.

[2] Our notion of coercion is essentially the same as in [Mit84a]

2.2 Conditional Expression and Conditional Statement

$$\frac{\begin{array}{l}\mathcal{A} \vdash b : \textbf{boolean} \\ \mathcal{A} \vdash e_1 : \sigma_1 \\ \mathcal{A} \vdash e_2 : \sigma_2\end{array}}{\mathcal{A} \vdash \textbf{if}\, b\, \textbf{then}\, e_1\, \textbf{else}\, e_2\, \textbf{end if} : \sigma_1 \vee \sigma_2} \tag{2}$$

Given a boolean expression b and two expressions e_1 and e_2 with types σ_1 and σ_2, respectively, the type of the conditional expression above is the join of the types σ_1 and σ_2[3]

We might be tempted to mandate that the types of both branches of a conditional expression be equal for the conditional to be well-typed. Once again, our rule leaves room for coercions, and is thus more flexible. Consider, for example the following assignment to a variable x.

$$x := IF integerflag THEN 5 ELSE 5.0$$

This assigment makes perfect sense if x is defined as a union of **integer** and **real**. We can think of both 5 and 5.0 being coerced into values of the union type, which is the smallest type greater than both **integer** and **real**. This particular rule not only applies to conditional *expressions*, but also to conditional *statements*, if we have a type **void**, the type of a statment with no return value, in our type model.

2.3 Case Expression and Case Statement

$$\frac{\begin{array}{l}\mathcal{A}, x : \tau_1 \vdash e_1 : \sigma_1 \\ \mathcal{A}, x : \tau_2 \vdash e_2 : \sigma_2 \\ \ldots \\ \mathcal{A}, x : \tau_n \vdash e_n : \sigma_n\end{array}}{\mathcal{A} \vdash \textbf{case}\, x\, \textbf{of}\, \tau_1 : e_1; \tau_2 : e_2; \ldots \tau_n : e_n\, \textbf{end case} : \sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_n} \tag{3}$$

if $\mathcal{A}$ contains $x : \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n$.

Assuming the n branches above have types σ_1 through σ_n if x has type τ_1 through τ_n, respectively, in them, then a case distinction on exactly those n types of x has the join of all branch types as its type if x has indeed type $\tau_1 \vee \tau_2 \vee \ldots \vee \tau_n$.

This rule is similar to the conformity clause in ALGOL 68. We can provide a rule also for the case statement with an else clause.

$$\frac{\begin{array}{l}\mathcal{A}, x : \tau_1 \vdash e_1 : \sigma_1 \\ \mathcal{A}, x : \tau_2 \vdash e_2 : \sigma_2 \\ \ldots \\ \mathcal{A}, x : \tau_n \vdash e_n : \sigma_n \mathcal{A}, x : \omega \vdash e_{n+1} : \sigma_{n+1}\end{array}}{\mathcal{A} \vdash \textbf{case}\, x\, \textbf{of}\, \tau_1 : e_1; \tau_2 : e_2; \ldots\, \textbf{else}\, \tau_{n+1} : e_{n+1}\, \textbf{end case} : \sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_{n+1}} \tag{4}$$

if $\mathcal{A}(x) \geq \tau_1 \vee \tau_2 \vee \ldots \vee \tau_n$.

Here ω denotes the greatest type in the type lattice. We assume ω exists, of course.

[3]Remember that we stipulated that the types have a lattice structure.

References

[Car85] L. Cardelli. Basic polymorphic typechecking. *Polymorphism*, 2(1), Jan. 1985.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. Technical Report CS-85-14, Dept. of Computer Science, Brown University, August 1985.

[Hen87] F. Henglein. Overloading in setl is not (syntactic) overloading after all. Technical Report (SETL Newsletter) 220, New York University, April 1987.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, Jan. 1983.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mit84a] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

[Mit84b] J. Mitchell. Type inference and type containment. In *Proc. Int'l Symp. on Semantics of Data Types, LNCS 173*, pages 257–277, June 1984.

[Rey85] J. Reynolds. Three approaches to type structure. In *Proc. TAPSOFT*, LNCS, pages 97–138. Springer-Verlag, 1985.