Strong Typing in SETL — An overview of work on strong typing in the NYU/SETL project

Fritz Henglein
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th floor
New York, N.Y. 10003
Internet: henglein@nyu.edu

March 31st, 1988

Abstract

SETL was conceived as a weakly-typed language. In an attempt to provide a strong typing discipline for SETL without compromising the style of declaration-free programming prevalent in SETL, we present some results and ongoing work on a flexible type model for SETL and on the associated (automatic) type inference problems.

1 Introduction

SETL's weak typing discipline has been repeatedly perceived to be a weakness, especially in large-scale prototyping applications [Sch87]. Yet the programming style SETL is conducive to has proved to be valuable for concise top-down program development. The characteristics of this style are, most notably, declaration-freeness, uniform polymorphism, extensive operator overloading, and certain data-structuring techniques such as "nesting" to simulate recursive data types and implicit, yet deliberate, use of union types. A strong typing discipline for SETL will have to enforce typing constraints while respecting these particular programming elements.

Furthermore, flexible records, procedures as first-class objects as well as abstraction and modularization facilities are included in the design of SETL-

2 [Smo88b] and are, consequently, addressed throughout in the typing issues outlined below.

2 Parametric Polymorphism

Parametric polymorphism refers to the ability of program procedures to be used with arguments whose types can be characterized by parametric type expressions. For example, the function `length`, which returns the length of a list, is applicable to arguments of type `list` α, where α is a type parameter that indicates that *any* concrete list type is permitted.

The theory of parametric polymorphism is quite old and goes back to Curry and Feys [CF58], Morris [Mor68], and Hindley [Hin69]. It entered the programming language arena through the seminal paper by Milner [Mil78]; it has been formalized as a typed λ-calculus by Damas and Milner [DM82].

Mycroft [Myc84] noticed that Milner's polymorphism was inadequate for recursive definitions. The problem is that recursively defined functions may only be used polymorphically in the "code section" of a program, not the declaration section where they are defined. He proposed an extension that he proved to preserve most of the properties of the pure Milner-style polymorphism. The resulting type inference problem was not known to be decidable or undecidable, though. Recently, Kfoury *et al.* [KTU88] showed, nonconstructively, that this problem is decidable. We had developed a constructive proof of the same result when [KTU88] was published. In the meantime we have shown that it is polynomial-time decidable [Hen88b] by providing a polynomial-time algorithm for the fundamental problem of semi-unification [Hen88a].

The practical implications are that no elaborately nested declarations and definitions are necessary to provide polymorphism everywhere in a program, including the declaration/definition sections themselves.

3 Dynamic Overloading

SETL makes extensive use of operator overloading. For example, `+` denotes integer addition, floating point addition, set union, tuple concatenation, and string concatenation. There are many more examples of overloading. Overloading has been regarded as beneficial for a clear and concise language design if the overloaded operators denote similar, that is — roughly — homomorphic, operations. Since many operators in SETL denote both tuple and set operations, sets can be changed to tuples (and the other way around) often without changing the code.

The programming language ML [Har86], whose core is based on Milner's polymorphism, provides minimal overloading. The problem is that "liberal", Ada-style overloading in a declaration-free language such as ML leads to an NP-hard type inference problem [ASU86, exercise 6.25]. Furthermore, ML's overloading calls for resolution in the syntactic context of operators, while SETL demands a more dynamic discipline [Hen87a]. For example, a generic sorting routine that uses the comparison operator `<`, which might denote integer, floating point, and lexicographic string comparison, is not possible in ML, whereas it is legal and an essential source of (restricted) polymorphism in SETL.

We have shown how dynamic overloading in SETL can be captured by a form of restricted polymorphism, which we have termed *oligotypes* [Hen87b]. *Polytypes* are quantified type expressions that express the polymorphic nature of a type; e. g. $\forall\alpha.\mathrm{list}\alpha \rightarrow \mathrm{integer}$ is the polytype of the `length` function referred to earlier. The crucial idea behind oligotypes is that they are simply polytypes with *restricted* quantified variables; e. g. $\forall\alpha \in \{\mathrm{integer}, \mathrm{real}, \mathrm{string}\}.\alpha \times \alpha \rightarrow \mathrm{boolean}$ is the oligotype of the abovementioned comparison operator `<`.

The general theory of oligotypes and its specialization to SETL remain to be investigated, but we have made a start by formulating a new axiomatization of parametric polymorphism and dynamic overloading [DH88] that is based entirely on first-order types (no type quantification) and is thus especially conducive to specification in the logic programming language Typol [CDD$^+$85]. Thus we have an executable type inference specification that can be used for purposes of experimentation.

4 Recursive Types and Union Types

Binary trees and other recursive data types are often simply modelled by nested tuples in SETL. Detection of such recursive types [Wei86] has led to improved performance of the data-flow oriented type finding algorithms developed earlier in a lattice-theoretic framework [Ten74]. It is well-known that recursive types can be inferred in a polymorphic framework by omitting the socalled "occurs check" in unification steps of the type inference algorithm (see, e. g., [Mil78, algorithm W].

Union types are most useful in connection with recursive types. For example, the type of binary trees with integer-labelled leaves is the union of a pair of such binary tree types (the reoccurrence of binary tree type in the definition makes this definition recursive) and the integer type. (Disjoint) union types that have components distinguished solely by their types (pair and integer in the example above), are called *free*, and are currently not included in our proposed type system for SETL. Instead we provide *tagged* union types in which the components carry tags (names) to disambiguate from which component of the union they are. For example, `[ node <- [t1, t2] ]` denotes a binary tree with two subtrees `t1` and `t2`; the tag `node` indicates that it is an internal node. `[ leaf <- 5 ]` stands for the binary tree consisting solely of the leaf labelled with the integer value `5`; the tag `leaf` indicates that it is a leaf.

Mishra and Reddy [MR85] report on type inference with parametric polymorphism, recursive types and free union types. They claim to have an "effective" algorithm for inferring the type of any expression, although they present no complexity-theoretic results. Since we believe that their type inference problem is hard (probably NP-hard), we intend to tackle the computational questions in their type inference problem in the near future.

5 Type Abstraction

SETL provides abstract types such as sets and maps, but has no *abstraction facilities* that let the programmer encapsulate his code and prevent a user from using his code in an unintended fashion.

Ernie Campbell, a student of Prof. Schonberg's, is currently working on developing type abstraction and modularization facilities suitable for SETL-2. His work is based on [MP85], [CW85], [Mac86], and [GP85] as well as the package facilities of Ada [Uni83].

6 Conclusion and Outlook

We have indicated some preliminary results and ongoing research at NYU on strong typing in SETL. We think the resulting type system will be flexible enough to support the sort of programming SETL users have grown accustomed to while providing a compile-time safety net considered indispensible in large-scale applications.

In connection with the language changes in SETL-2, notably procedures as first-class objects, records, and abstraction facilities [Smo88b], we feel very optimistic about the future of SETL. With the development of a programming environment [Kel87], the publication of reference material [SDDS86], and easily portable reimplementations ([Smo88a] and [Kel87, working group S1]), the major obstacles that have kept SETL from expanding into academic and industrial markets may well be overcome soon.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* 1986.

[CDD+85] D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, June 1985.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[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.

[DH88] C. Dubois and F. Henglein. A first-order axiomatization of parametric polymorphism and dynamic overloading and its typol specification. Technical Report (SETL Newsletter) 224, New York University, May 1988.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[GP85] D. Gries and J. Prins. A new notion of encapsulation. *Proc. ACM SIGPLAN '85 Symp. on Language Issues in Programming Environments, SIGPLAN Notices*, 20(Part 2):131–139, 1985.

[Har86] R. Harper. Introduction to standard ml (preliminary draft). Technical report, University of Edinburgh, February 1986.

[Hen87a] F. Henglein. Overloading in setl is not (syntactic) overloading after all. Technical Report (SETL Newsletter) 220, New York University, April 1987.

[Hen87b] Fritz Henglein. A polymorphic type model for SETL. SETL Newsletter 221, New York University, July 1987.

[Hen88a] Fritz Henglein. The Milner-Mycroft calculus. SETL Newsletter 223, New York University, April 1988.

[Hen88b] Fritz Henglein. Semi-unification. SETL Newsletter 222, New York University, April 1988.

[Hin69] R. Hindley. The principal type-scheme of an object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[Kel87] J. Keller, editor. *Athens Review Meeting.* ESPRIT Project 1227: SETL Expermination and Demonstrator, July 1986 - July 1987.

[KTU88] A. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 58–69. ACM, ACM Press, Jan. 1988.

[Mac86] D. MacQueen. Using Dependent Types to Express Modular Structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, January 1986.

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

[Mor68] J. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.

[MP85] J. Mitchell and G. Plotkin. Abstract Types have Existential Type. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 37–51. ACM, January 1985.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 7–21. ACM, January 1985.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[Sch87] E. Schonberg. Personal communication, 1987.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Smo88a] M. Smosna. The implementation of SETL-2 – an experiment in the use of ada as an implementation language. Technical Report (SETL Newsletter) 226, New York University, June 1988.

[Smo88b] M. Smosna. SETL-2 – a rationale for the redesign of SETL. Technical Report (SETL Newsletter) 225, New York University, June 1988.

[Ten74] A. Tenenbaum. Type determination for very high level languages. Technical Report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974.

[Uni83] United States Department of Defense. *Reference Manual for the ADA Programming Language*. Springer-Verlag, 1983.

[Wei86] G. Weiss. Recursive data types in setl: Automatic determination, data language description, and efficient implementation. Technical Report 201, Courant Institute of Mathematical Sciences, New York University, March 1986.