Beyond ML

John C. Reynolds
Carnegie Mellon University

We believe that the most pressing problem in programming-language design is to control the interference that arises when functional and imperative programming are combined, and to give the user more control of storage allocation, without sacrificing the virtues of higher-order procedures, rigorous typing, and modularity that characterize ML. Towards this end, we intend to pursue research on Algol-like languages, linear typing, and the use of regions for storage allocation.

The programming language ML (or more properly, Standard ML) [Milner et al. 1990; Milner and Tofte 1991] can be regarded as a major triumph of modern language design: It is the first language to obtain a significant (and rapidly increasing) degree of popular acceptance whose design is soundly based upon theoretical principles. In particular:

—It is rigorously typed, i.e. its meaningful programs are exactly those that satisfy a concept of well-typedness that can be verified mechanically.
—It supports higher-order procedures.
—It provides a sophisticated type-based facility for modularization.

However, although ML is a major step forward in comparison with earlier languages that have achieved wide usage, it is not the ultimate programming language. Inevitably, its usage will be limited by two serious shortcomings:

—In addition to its functional core, ML provides imperative features (via the reference mechanism) that are subject to the problems of variable aliasing and interfering side effects, which make programs difficult to reason about (especially when the language is extended to concurrency). Moreover, the imperative aspects of ML programs are not well described by the type-theoretic specifications used for modularization, which are essentially oriented towards functional behavior.
—Storage in the language is fundamentally based on the use of a heap. Although more efficient storage allocation may occasionally be introduced by an optimizing compiler, it is difficult for the programmer to estimate or control the use of storage.

We believe that the most pressing current problem in programming-language research is to design a language that overcomes these shortcomings without sacrificing the advantages of a sound theoretical basis. Although research has not yet progressed to the point where it is clear how to meet this goal, there are three directions that have considerable promise:

(1) Algol-like languages. Although modern Algol-like languages [Reynolds 1981], such as Forsythe [Reynolds 1996], are rigorously typed, provide higher-order procedures, and can probably be provided with type-based modularization,

they can be implemented by using a stack rather than a heap. In their pure form, they are far more storage efficient than ML and give the programmer far more control over storage, but they are limited to programs that fit within a stack discipline. However, it is likely that such languages can be extended to achieve the generality of a heap-based language while providing a more flexible storage discipline.

Moreover, we understand how to restrict Algol-like languages so that aliasing and interference become syntactically detectable in a failsafe manner [Reynolds 1978; Reynolds 1989; O'Hearn et al. 1995]. With this restriction, it is possible to extend these languges to include either shared-variable or message-passing concurrency, without permitting processes to interact in an uncontrolled manner.

(2) Linear Typing. Linear logic provides a refined type system for a functional programming language where one can define functions that consume their arguments in the process of computing their result [Girard, J.-Y. 1987; Lafont 1988; Abramsky 1993]. As a consequence, one can describe imperative-like state transformations and mutable objects in a purely functional settting, where aliasing and side-effects cannot occur.

Recently, in joint current research with Peter O'Hearn, we have found how to translate Algol-like programs (with or without syntactical control of interference) into a functional language with polymorphic linear types. This demonstrates that a linearly typed functional language can be at least as expressive as Algol. Moreover, in such a language one can go beyond conventional imperative languages to construct state transformations that alter the type of the state, or even split or combine states.

Of course, linear typing does not obviate the need for heap-based storage allocation (using either garbage collection or reference counts). But the translation from Algol suggests that it may be possible to identify a cleanly demarcated sublanguage that obeys a stack discipline.

(3) Regions. It has been shown that ML programs can be implemented using a stack of regions some of which are heaps [Tofte and Talpin 1994]. Such an implementation can be obtained from a complex program analysis that yields programs of widely varying storage efficiency, ranging from pure heap to pure stack allocation.

Experience with region analysis indicates that the most efficient programs are obtained when the programmer understands the nature of the analysis. This suggests that a language with explicitly defined regions could provide a high degree of control over storage.

It is fascinating to note that a form of explicitly defined regions was provided more than thirty years ago by the record declarations of Algol W. It would be a major step forward if such a mechanism could be adopted to the functional setting.

In summary, we are confident that the vigorous pursuit of these research directions can lead to a programming language that combines rigorous typing, higher-order procedures, and type-based modularity with explicit and flexible control of storage allocation.

REFERENCES

ABRAMSKY, S. 1993. Computational interpretations of linear logic. *Theoretical Computer Science 111*, 1–2 (April 12), 3–57.

GIRARD, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50*, 1, 1–101.

LAFONT, Y. 1988. The linear abstract machine. *Theoretical Computer Science 59*, 1–2 (July), 157–180. Corrigendum in (3) 62, 1988, pp. 327–328.

MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts.

MILNER, R., TOFTE, M., AND HARPER, R. W. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts.

O'HEARN, P. W., POWER, A. J., TAKEYAMA, M., AND TENNENT, R. D. 1995. Syntactic control of interference revisited. In S. BROOKES, M. MAIN, A. MELTON, AND M. MISLOVE Eds., *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, Volume 1 of *Electronic Notes in Theoretical Computer Science* (Tulane University, New Orleans, Louisiana, March 29–April 1 1995). Elsevier Science (http://www.elsevier.nl).

REYNOLDS, J. C. 1978. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (1978), pp. 39–46.

REYNOLDS, J. C. 1981. The essence of Algol. In J. W. DE BAKKER AND J. C. VAN VLIET Eds., *Algorithmic Languages* (Amsterdam, 1981), pp. 345–372. North-Holland.

REYNOLDS, J. C. 1989. Syntactic control of interference, part 2. In G. AUSIELLO, M. DEZANI-CIANCAGLINI, AND S. RONCHI DELLA ROCCA Eds., *Automata, Languages and Programming*, Volume 372 of *Lecture Notes in Computer Science* (Berlin, 1989), pp. 704–722. Springer-Verlag.

REYNOLDS, J. C. 1996. Design of the programming language Forsythe. Report CMU–CS–96–146 (June), Carnegie Mellon University.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1994), pp. 188–201.