Foundations of Formal Program Development [*]

Some Aspects of Research in the Ergo Project[†]
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

DRAFT IN PREPARATION — DO NOT DISTRIBUTE

Principal Investigators:
Dana Scott and John Reynolds.

Other Faculty:
Stephen Brookes, Robert Harper, Peter Lee,
Daniel Leivant, Frank Pfenning, and Eugene Rollins.

Gradute Students:
Ken Cline, Scott Dietzen, Conal Elliott, Tim Freeman, Nevin Heintze,
Spiro Michaylov, Robert Nord, Benjamin Pierce, Olin Shivers, Jean-Philippe Vidal.

Abstract

The Ergo Project at Carnegie Mellon University conducts a program of research into the foundations of formal program development. The overall goal of the project is to provide a framework for the development and maintenance of provably correct programs and demonstrate its utility through prototype implementations and examples. In this report we describe some selected aspects of our research towards this goal.

[*]This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

[†]This report was prepared by Robert Harper, Peter Lee, and Frank Pfenning.

1. Background

The central problem of software engineering is the development and maintenance of demonstrably correct programs. In order to accomplish this goal, a wide variety of approaches to the problem have been developed. These range from what might be called "software management techniques," devoted primarily to addressing the practical questions of how best to manage a software development project in order to improve the quality of the product, to fundamental research programs that seek to develop a rigorous body of theory on which software development methodologies may be based. These two approaches (among many others) are not opposed to one another, but rather address two equally important aspects of the problem: on the one hand, the question of what can be achieved now, and on the other, the question of what might be achievable at some point in the future. Research into the development of reliable and maintainable programs is an ongoing dialogue between workers on these two approaches, with fundamental research results continually being incorporated into the mainstream of software development, and with the problems of practical software development providing the framework in which fundamental research is being conducted.

The Ergo Project is a program of fundamental research in programming methodology [9]. The central thesis of the project is that the future of correct software development lies in the use of machine-assisted formal methods both to develop and maintain software. This approach to the correctness problem in software development is based on two guiding principles. First, any notion of correct program development can only be based on a comprehensive and rigorous theory of programming. Such a theory must include not only the design of programming notations, but also the design of logics and methodologies for reasoning about them. Second, the process of correct program development on such a basis must involve the manipulation of formal structures, including programs themselves, arguments for their correctness, and their development histories. These manipulations can only be made practical by the use of machine assistance.

2. Objectives

The overall objective of the Ergo Project is to develop a comprehensive theory of machine-assisted formal program development. This overarching goal naturally decomposes into a number of smaller objectives that we believe are crucial elements of the overall program of research, and are relevant to the work of other researchers in the area. The short-term objectives of the project include the following goals:

1. To consider the problem of the formal representation of structures that arise in program development. These structures include, for example, programs, program derivations, and proofs of correctness.

2. To develop the tools and techniques necessary for the manipulation of these structures; in particular, to determine how best to write *metaprograms* for working with these structures.

3. To experiment with these tools by conducting examples of formal program developments. These experiments serve not only to assess the feasibility of the methodologies under consideration, but also to test the tools and techniques developed for this purpose.

3. Research Issues and Approaches

The work of the project focuses at present on three fundamental research issues that may be summarized under the headings: representation, manipulation, and experimentation. In this section we present a brief overview of each of these topics, and defer the technical details to Section 4 where a summary of our research is presented.

Representation. Here the fundamental issue is to find a suitably general framework in which a wide variety of formal structures may be represented. The principal aim here is to choose a system of representation with sufficient structure to capture the uniformities that arise in a wide variety of formal systems without overcommitment or excessive generality. In particular, a significant portion of the implementation effort associated with representing formal structures in a machine should be "factored out" into the representation language, and implemented once for all. Moreover, the representation should lend itself to the sorts of manipulations needed for formal program development. Many of the classical techniques seem inadequate for representing programs, assertions, and proofs. For example, the representation of programs as trees (simplified parse trees or terms in an ML concrete data type) fails to capture the binding structure exhibited by most programming languages: variables have scope, and their meaning is governed by the scoping rules of the language. On the other hand, elaborations of the notion of abstract syntax trees by the attachment of attributes and attribute-computation functions seems to be excessively general. Although the relevant additional structure can be expressed using attributes, no general structural features are modeled in an attributed tree, rather each language is handled on a case-by-case basis. We were thus led to consider alternative representations that are more expressive than unattributed trees, and more restrictive than general attribution mechanisms, namely terms in a typed λ-calculus. The idea is that λ-abstractions capture the elementary properties of a wide range of binding operators, and the type structure is chosen so as to admit convenient expression of the combinatorial properties of term constructors and inference rules. The result is that formal structures are represented as well-typed λ-terms, and a substantial amount of the machinery associated with the implementation of proof and program development tools can be factored out into the implementation of the representing λ-calculus. This approach opens the way to the development of a variety of *language-independent tools*, allowing for rapid experimentation with a variety of programming languages and formal systems.

Manipulation. In addition to issues of representation, there are also the general problems of finding the appropriate means of handling the formal structures associated with program development. Here we have in mind formal program derivation, construction of formal proofs, the transformation of programs and proofs, and the construction and use of tools for conducting formal proof and program derivations. A variety of problems may be classified under this heading, including the design of "metaprogramming languages" and proof development environments. In connection with metaprogramming, issues such as the trade-offs between functional and logic programming and the choice of type structure seem particularly important. For example, the use of typed λ-calculi to represent formal structures immediately suggests two lines of research. In one direction it is natural to exploit variations of higher-order unification as the fundamental tool for manipulating programs and proofs. This leads to consideration of logic programming languages that employ such unification algorithms in the search process. Our work on eLP, HOAS, and Elf may be seen as developments along these lines. In another direction it is natural to consider the incorporation of the type structures used to represent programs and proofs into functional programming languages. A number of issues arise in the course of pursuing this line of research, including questions about adding dependent types to an ML-like language, adding a type of "intensional functions" that admit a decidable equality test, and the use of higher-order impredicative types to represent data structures. Our work on LEAP represents a significant effort in this direction.

In connection with formal proof and program development, a number of related issues arise. For example, logic programming languages that manipulate terms in variations of the typed λ-calculus seem to provide a convenient basis for building formal proof development systems, but much more work is needed to assess the practicality of this approach. Questions of modularity seem to be of central importance here, both from the point of view of metaprogramming, and from the point of view of the program development process. Just as the use of modularization constructs seems essential to the management of large programs, some form of modularity also seems to be of central importance in formal program development. For example, it seems important to provide the ability to organize the logical structure of the correctness proof of a program, thereby allowing for localization of proof obligations and control over the size of the search space associated with conducting formal proofs. Questions of modularity also impinge on the very notion of formal program development, for the ideas of modular development and stepwise refinement of program modules seem to be of fundamental importance to the refinement process. Since complete formal program development involves considerable effort, the ability to reuse earlier work is an important topic for research. Current work in this direction is concerned with the development of methods of analogical reasoning and generalization to support re-use of programs and their correctness proofs, and to allow the adaptation of program developments from one setting to another.

Experimentation. Here we include both the prototyping of tools for formal program development and their application in test cases. A number of engineering issues arise in the process of building prototype tools. For example, the use of rich type systems in metaprogramming languages leads to such issues as type inference and compilation techniques for these languages. There is a fundamental tension between the use of expressive type systems to encode properties of programs and data structures, and the practical need to alleviate the burden of supplying detailed type information in programs.

One of the crucial problems in building environments is that of communication among the many tools that comprise such environments. The channels of communication, or *abstract interfaces*, are very closely tied to the representation languages and success or failure in the evolution of large systems may very well depend on whether abstract interfaces are well-chosen and adaptable.

4. Progress

We now summarize the progress of our research in representations, manipulation, and experimentation for formal program development.

Representation. There are two main focuses for our work on representations, *higher-order abstract syntax* (HOAS) and the *Logical Framework* (LF).

HOAS [17] extends the classical notion of abstract syntax with a typed λ-calculus. With this simple extension, "higher-order" typed λ-terms are used to represent in a uniform way those constructs of a language that bind variables. Such uniformity facilitates the development of tools and analysis techniques that are generally applicable to a wide range of languages, independent of the manner in which variables are bound. For example, it is relatively easy to implement pattern matchers and rewriting systems that respect the scoping properties of any language represented in HOAS, since these problems can be solved via higher-order matching and unification. This idea is by no means new and is exploited in a variety of ways in other systems. The Ergo project's contribution here is two-fold: one is the realization that many practical problems with the use of the simply typed λ-calculus as a representation language can be solved by products and a simple form of polymorphism, the other is the implementation of these concepts in our experimental environment, the Ergo Support System (ESS) [10]: a central procedure in the ESS is a higher-order unifier, and HOAS is used as the standard representation for programs and proofs. In this way, the ESS provides language-independent matching and rewriting of programs and formulas.

LF [6] adds power to the simply typed λ-calculus as a representation language by using *dependent function types* to represent families of objects and their types. This extension is crucial for representing the syntax, and especially the inferential structure, of logical sytems. In particular, the notions of axiom and rule schemes and the fundamental mechanisms of natural deduction, and hypothetical and general reasoning, may be succinctly represented in LF. This leads to a particularly simple methodology for representing formal systems in a machine: a formal system is presented as an LF *signature* declaring a sequence of constants that serve as generators for the syntactical structures of that system. Under this methodology many "side conditions" on inference rules are eliminated, and the remainder are axiomatized directly using "auxiliary" judgement forms. The signature of a formal system is suitable for direct use by a "proof editor generator" or to induce operators in a search space for proof and program synthesis.

Recently, Harper has developed the theory of logical relations for LF to prove confluence and strong normalization for a restricted form of $\beta\eta$ reduction for the LF type theory. This result, together with the subject reduction property, yields a proof of decidability of the LF type system with full β and η conversion. This eliminates the need for a crucial assumption in the completeness proof of the unification algorithm for LF developed by Elliott [3,2] (which is discussed below).

Related to this is Harper's joint research on *Structure and Representation in the Logical Framework* [7] which is working towards a theory of modular and well-structured presentations of deductive systems in LF. This work is concerned with two forms of modularity in LF: in the presentation of theories within a given logical system, and in the presentation of logical systems themselves. The goal of this work is to develop a language for presenting theories and logics in an organized way, and to consider the ways in which the structure of a presentation may be exploited in the process of proof search.

Manipulation. To be useful in the activities surrounding formal program development (such as proof development, formal derivation, and metaprogramming), a higher-order abstract syntax representation requires both a higher-order unification procedure and an appropriate metalanguage. Our study of higher-order unification began with Huet's algorithm [8], but quickly we discovered the need to extend and improve it. This led us to conduct a formal derivation (on paper) of a family of unification algorithms [5,4] on which our present implementation in the Ergo Support System is based.

As for the metalanguage, we are, as mentioned in the previous section, developing and experimenting with both functional and logic programming languages. We experimented first with Miller and Nadathur's language, λProlog [11]. This language is based on higher-order unification and quite naturally supports the manipulation of HOAS representations in the setting of logic programming. λProlog is thus useful for the rapid prototyping and elegant expression of many metaprograms. We implemented λProlog as an important, but also separately exportable component of the ESS. Our implementation (eLP) makes heavy use of many components of the ESS and is thus also a good test case for our environment.

Along similar lines, we have proposed Elf [14], a logic programming metalanguage which combines ideas from λProlog and LF. The implementation of this language makes use of Elliott's work on higher-order unification with dependent types [3,2]. Elf gives an operational interpretation to types much in the same way that Prolog gives an operational interpretation to Horn clauses. Unlike in logic programming languages, however, the proofs that are built by the search process of the interpreter itself are first-class objects and can be shown or transformed further. The next logical step is to connect Elf to the interaction tools in the Ergo Support System to form a proof and program development environment that allows interactive input and the invocation of metaprograms. In another direction it is important to develop a theory of modularity for Elf programs. Ideas from λ-Prolog and on structured presentations in LF are relevant here.

Because of the central role of polymorphic and dependent types in representations, and also the link between type inference for the polymorphic λ-calculus and higher-order unification (as shown by Pfenning in [15]), it seems that a polymorphic functional language might also be appropriate for certain metaprogramming applications. Such a language would constitute an efficient alternative to the logic-programming-based metalanguages discussed above. We have been developing and experimenting with such a language. Our language, called LEAP, is an explicitly typed functional metalanguage in the ML tradition, but based on the ω-order polymorphic λ-calculus [18]. Recent developments for LEAP include the treatment of inductively defined data types [19], the development of compilation techniques for inductive definitions and primitive recursion, and a tutorial on programming in higher-order typed λ-calculi prepared by Benjamin Pierce, Scott Dietzen, and Spiro Michaylov [20]. An important and novel aspect of this work is that type information is used to guide code generation, allowing a conceptually simple, explicitly typed language to be compiled into efficient code for conventional architectures.

Experimentation. Our experimentation can be divided into two categories: experiments in formal program development, and the development of language implementation techniques.

Our formal, on-paper derivation of an algorithm for higher-order unification was in part an experiment in formal program development, but it also led to significant extensions to Huet's higher-order unification algorithm, culminating ultimately to Elliott's work on unification with dependent types [2]. Unfortunately, even though the derivation has motivated many aspects of the implementation of the ESS, our environment is not yet ready to support derivations at this level of complexity. The achievement of such environmental support is one of the major milestones of our work in experimentation, and one that we hope to achieve in the near future.

Frank Pfenning has been working on extending the traditional proofs-as-programs paradigm to include proof transformation as a significant aspect of verified program development. He derived a number of algorithms for graph reachability [16] which were subsequently implemented in the Calculus of Constructions by Christine Paulin-Mohring [13].

On the subject of larger-scale derivations, Rod Nord and Peter Lee have been experimenting with extensions to Scherlis' specialization system [22] as they apply to data type transformations [21] in order to handle the derivation of larger programs. Thus far they have accomplished the derivation (again, on paper) of a simple display editor [12]. They are currently considering collections of larger data types, and also considering extensions to the ESS to support larger-scale program derivation.

Scott Dietzen and Frank Pfenning have completed a number of experiments on explanation-based generalization in higher-order domains such as theorem proving and program derivation. This work, which has been conducted in the ESS using mainly eLP, is primarily directed towards the application of these techniques to the problem of analogy in program development. This work is described in [1].

The implementation of LEAP has been an excellent test case for the basic system design of the ESS, as our prototype LEAP compiler uses many components of the system: parsing and unparsing of LEAP programs is handled by the ESS's syntax facility, name binding information is specified as an attribute grammar, and both type inference and the prototype compiler are written in eLP (using the algorithm in [15]). All of these components communicate with eachother *via* standardized data structures provided primitively by the ESS. As aspects of the LEAP compiler are developed, further refinement of the ESS takes place. Already, we have found in our experimentation with LEAP that metalanguages such as eLP are an enormous aid in rapidly prototyping new ideas in compilation, type inference, and type argument synthesis. With such prototyping support of the ESS, we expect to learn a great deal about the practicality of the representations and metalanguages that we are developing.

Summary We are quite encouraged by our recent progress, as we believe we are moving well toward our goal of building an integrated system for formal program development. The work on LEAP, eLP, and Elf provide a spectrum of semantically based and logically based languages for metaprogramming and specification. These languages provide a great deal of expressive power while still retaining the ability to analyze and transform programs. Furthermore, implementing and integrating these languages into the Ergo Support System has led to a significant strengthening and enhancement of the various tools and abstract interfaces. We now feel confident that export of the ESS to other research sites will result in constructive feedback and further improvements.

5. Directions

On the basis of the research results that have so far been obtained in the area of formal program development (both by the present authors and others), we believe that research into the mathematical foundations of formal program development is an essential component of the overall program of building reliable and maintainable software. Although fundamental research of this kind does not usually lead to immediate changes in the practice of software development, the results obtained at the foundational level have, and will continue to, play a central role in the development of the field. To take a ready example, it is by now accepted in both theory and practice that types play a central role in the organization of programs and programming languages, and are crucially important to the software development process. This development was strongly affected by years of research into such seemingly remote topics as the typed λ-calculus, intuitionistic mathematics, and category theory. It is our opinion that fundamental research on the foundations of program development is now reaching maturity, and can be expected to play an increasingly significant role in the future of software development.

On the other hand, the progress of research into formal program development does not appear to proceed directly toward transferrable technology in the short run, although there have been a number of examples of the relatively short-term transfer of theoretical research to practical software development (the theory of context-free grammars leading to parser generators, and the theory of automata to event-driven approaches to concurrency, to take but two examples). We therefore think it important *not* to expect short-term technology transfer, and that it is a mistake to place too much stress on concrete gains for the near term in practical software development. It is only by taking the long-term view that we can imagine what might be possible, thereby providing the basis for future development of the field.

6. Grand Challenge

The grand challenge is to make large-scale formal program development feasible.

7. Research Transitions

As should be clear from the above discussion, the primary goal of research into the foundations of formal program development is to improve the reliability and maintainability of programs. The intended market for our results is therefore the software development community. It is our expectation that the influence of our work (and that of other researchers in related areas) will not take the form of a direct "technology transfer," but will instead have a less direct, yet significant, influence on "programming culture." That is to say, it is to be expected on the basis of current experience that fundamental research will have an important influence the practice of software development by providing the foundation on which rigorous methodologies may be based. In this sense there is a "transfer gap"—a significant time lag between the time that influential research takes place and the time that these ideas make their way into the practice of software development. Of course, not all research ideas will in fact pan out. To separate the important developments from the false steps takes a substantial amount of time that cannot (it seems) be shortened.

Still, we feel it is appropriate to consider what can be done to facilitate a greater amount of "technology transfer." In our opinion, the implementation and distribution of tools that embody ideas from foundational research form crucial links in the computer science community in general. We have learned a great deal from both using the tools distributed by others, as well as distributing our own and discussing the results of other's experiments.

The real challenge in technology transfer, of course, is to achieve the adoption of formal methods by the application community. Much "cultural" development will have to take place, for we believe the teaching of new languages and new ideas is crucial.

8. Technological Impacts

The widespread use of workstations has been an important influence on the development of many, if not most, fields in computer science, not least in software development. Aside from the expected growth in performance and capacity of these workstations, we do not envision the need for computer technology of any remarkably different character than is currently available to further the aims of our research.

9. Societal Issues

As the number of research and faculty positions in computer science is stabilizing, the need for support for post-doctoral research positions becomes increasingly important. We have found on several occasions that we would like to have arranged for a one- or two-year visiting position with the project, but were unable to do so for lack of funding for such positions. Moreover, as positions become more scarce than in the past, new Ph.D.'s more often find it attractive to take a temporary research position with a project in order to establish a basis for future research. We therefore recommend that support for post-doctoral research fellows be increased.

Our principal complaint with regard to the current funding situation is that we find the reporting requirements to be excessively burdensome and insufficiently beneficial to the research effort. At present we are required to file quarterly reports and an annual report, plus a variety of progress reports and reviews that arise from time to time (and, all too frequently, on very short notice). These requirements are all the more burdensome because we receive no significant feedback on the progress that we report. Without a significant response to our reports, the effort is, for us, essentially wasted. We therefore strongly recommend that the reporting requirements and procedures be reviewed with an eye toward requiring fewer reports, providing much longer advanced notice, and responding to these reports with a nontrivial evaluation of the progress of the research.

10. Recommendations for Funding Agencies

In summary, we recommend that:

- Support for fundamental research in formal program development be at least maintained at the present levels.
- The requirement for immediate transfer of research results to practical software development be de-emphasized.
- The system of project management be reviewed, especially with respect to the quantity and nature of the reporting requirements.

References

[1] Dietzen, S. and Pfenning, F. *Higher-Order and Modal Logic as a Framework for Explanation-Based Generalization.* In: **Sixth International Workshop on Machine Learning**, edited by A. M. Segre. Morgan Kaufmann Publishers, San Mateo, California, 1989, pp. 447–449. *Expanded version available as Technical Report CMU–CS–89–160, Carnegie Mellon University, Pittsburgh.*

[2] Elliott, C. *Extensions and Applications of Higher-order Unification.* Carnegie Mellon University, June 1989. *To appear.*

[3] Elliott, C. *Higher-Order Unification with Dependent Types.* In: **Rewriting Techniques and Applications**. Springer-Verlag LNCS 355, 1989, pp. 121–136.

[4] Elliott, C. *Some Extensions and Applications of Higher-order Unification: A Thesis Proposal.* Ergo Report, no. 88–061, Carnegie Mellon University, Pittsburgh, June 1988. *Thesis to appear June 1989.*

[5] Elliott, C. and Pfenning, F. *A Family of Program Derivations for Higher-Order Unification.* Ergo Report, no. 87–045, Carnegie Mellon University, Pittsburgh, November 1987.

[6] Harper, R., Honsell, F., and Plotkin, G. *A Framework for Defining Logics.* In: **Symposium on Logic in Computer Science**. IEEE, 1987, pp. 194–204.

[7] Harper, R., Sannella, D., and Tarlecki, A. *Structure and Representation in LF.* In: **Fourth Annual Symposium on Logic in Computer Science**. IEEE, 1989, pp. 226–237.

[8] Huet, G. *A Unification Algorithm for Typed λ-Calculus.* **Theoretical Computer Science**, vol. 1 (1975), pp. 27–57.

[9] Lee, P., Pfenning, F., Reynolds, J., Rollins, G., and Scott, D. *Research on Semantically Based Program-Design Environments: The Ergo Project in 1988.* Technical Report, no. CMU-CS-88-118, Carnegie Mellon University, Pittsburgh, March 1988.

[10] Lee, P., Pfenning, F., Rollins, G., and Scherlis, W. *The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments.* In: **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, edited by P. Henderson. ACM Press, 1988, pp. 25–34. *Also available as Ergo Report 88-054.*

[11] Miller, D. A. and Nadathur, G. *Higher-Order Logic Programming.* In: **Proceedings of the Third International Conference on Logic Programming**. Springer Verlag, 1986.

[12] Nord, R. L. *Deriving and Manipulating Module Interfaces — Thesis Proposal.* Ergo Report, Carnegie Mellon University, Pittsburgh, 1989. *In preparation.*

[13] Paulin-Mohring, C. *Extraction de programmes dans le Calcul des Constructions.* Université Paris VII, January 1989.

[14] Pfenning, F. *Elf: A Language for Logic Definition and Verified Meta-Programming.* In: **Fourth Annual Symposium on Logic in Computer Science**. IEEE, 1989, pp. 313–322. *Also available as Ergo Report 89-067.*

[15] Pfenning, F. *Partial Polymorphic Type Inference and Higher-Order Unification.* In: **Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah**. ACM Press, 1988, pp. 153–163. *Also available as Ergo Report 88-048.*

[16] Pfenning, F. *Program Development through Proof Transformation.* In: **Logic and Computation**, edited by W. Sieg. **Contemporary Mathematics**, AMS, Providence, Rhode Island, 1988. *To appear. Available as Ergo Report 88-047.*

[17] Pfenning, F. and Elliott, C. *Higher-Order Abstract Syntax.* In: **Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia**. ACM Press, 1988, pp. 199–208. *Available as Ergo Report 88-036.*

[18] Pfenning, F. and Lee, P. *LEAP: A Language with Eval and Polymorphism.* In: **TAP-SOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain**. Springer-Verlag LNCS 352, 1989, pp. 345–359. *Also available as Ergo Report 88–065.*

[19] Pfenning, F. and Paulin-Mohring, C. *Inductively Defined Types in the Calculus of Constructions.* In: **Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics**. Springer Verlag LNCS, 1989. *To appear. Available as Ergo Report 88–069.*

[20] Pierce, B., Dietzen, S., and Michaylov, S. *Programming in Higher-order Typed Lambda-Calculi.* Technical Report, no. CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1989.

[21] Scherlis, W. L. *Abstract Data Types, Specialization and Program Reuse.* In: **International Workshop on Advanced Programming Environments**. Springer-Verlag LNCS 244, 1986.

[22] Scherlis, W. L. *Program Improvement by Internal Specialization.* In: **Eighth Symposium on Principles of Programming Languages**, ACM. ACM, 1981, pp. 41–49.