Puzzles in Programming Logic*
DRAFT October, 1986

Albert R. Meyer†
M.I.T. Laboratory for Computer Science
545 Technology Square, Rm. 801
Cambridge, MA 02139 USA

Very abstract concepts underlie the design of modern programming languages—higher-order functions, abstract data types, types as values, names of names. These concepts arise naturally in the course of program design; with an intuitive explanation, programmers generally understand and use them effectively.

Nevertheless, intuition has its limits. Situations arise repeatedly where firmer guidelines are needed to resolve confusions and inconsistency. This note illustrates some of the places where intuitive programming concepts are not adequate. Some specific puzzling cases are collected below which highlight areas where more guidance would be valuable. The statements of the puzzles are, I hope, almost all accessible to novice programmers, though they vary in difficulty as well as significance.

My emphasis is on puzzles which come up in most programming languages, not on problems reflecting the idiosyncrasies of some particular language. "Solutions" for the puzzles are supplied, but they are really brief hints about the current state of the theoretical answers. Simple as the puzzles appear, several of them currently lack satisfactory solutions.

The puzzles are intended to make the point that reasoning about program behavior raises challenges with a significant mathematical component. One aim of programming language theory is to provide a mathematical foundation for the abstract concepts in programming.

Software engineering, in common with other design disciplines, deals with the organization and management of large systems interacting with complex environments. At the same time it contrasts with other design disciplines: programs, like essays or

*This paper was presented at *A Workshop on Formal Software Development: Combining Specification Methods*, Nyborg, Denmark, May, 1984, D. Bjorner, chair. No proceedings were published.

†This work was supported in part by The National Science Foundation, Grant No. MCS 8010707 and by DARPA grant ???

mathematical proofs, are built of symbols and words, and lack any physical realization. The constructs of software engineering are more purely conceptual than those of other engineering disciplines.

Programs do, of course, direct the behavior of physical devices—computers — so once a computer is at hand, there is a sure way to resolve any doubt about how the program will behave in a particular instance—run it. Some theoretical analysis is needed to certify how the program will behave in general. But a central question for programming language theory is less how does a program behave, but how *should* it behave? The puzzles are also intended to illustrate some of the unexpected complexity of making such design decisions.

There are surely many more puzzles in the style of the ones below about the numerous features typical in modern languages. If the reader has some to contribute, I would be grateful to hear about them.

1 The Puzzles

Puzzle 1 *Exhibit a declaration of a procedure E which takes no arguments and returns an integer value such that the conditional expression*

$$\mathbf{if}\ E = E\ \mathbf{then}\ 0\ \mathbf{else}\ 1\ \mathbf{fi}$$

evaluates to 1 *in most programming languages.*

I was disappointed to discover that most people actually solve Puzzle 1 so quickly that they aren't struck by the *real* puzzle: *how is one supposed to make sense of programming languages in which this kind of thing happens?*

Puzzle 2 *Let E be an integer procedure whose evaluation never properly terminates, e.g., E might be declared by*

$$\begin{array}{l}\mathbf{integer\ procedure}\ E;\\ \quad \mathbf{return}(E+1);\\ \mathbf{end}\ E.\end{array}$$

What simple integer expression e is a counter-example to the rule

$$E + e = e + E?$$

Puzzle 3 *Exhibit a simple context into which either of the phrases* $(1+2)$ *or* $(2+1)$ *can be substituted so that in essentially all programming languages the resulting substitutions yield different results.*

It is too easy to dash expectations of sensible program behavior if we exploit the features on which the previous puzzles were based. So henceforth we shall assume that expressions are without side-effects, i.e., expression evaluation does not cause detectable changes in the state of the computer store or other memory. Expressions which induce breaks in flow-of-control, e.g., whose evaluation can cause exceptions, can also be pathological, and we shall disallow these too in the discussion which follows. Finally, we shall assume that program phrases have and are used in appropriately typed contexts—not, for example, as character strings.

Puzzle 4 *Consider the side-effect-free procedure G declared as follows:*

> **integer procedure** $G(x, y)$;
> x, y : **integer**;
> **if** $x = 0$ **then return**(y) **else return**$(G(x+1, y))$ **fi**;
> **end** G.

In the scope of this declaration, does the test $G(0, y) = G(0, y)$ evaluate to true? How about $G(1, y) = G(1, y)$? Should a compiler optimize by skipping these tests?

This puzzle emphasizes the familiar fact that one has to take account of the possibility of nonterminating computations in reasoning about expression evaluation—even after making the simplifying assumption that expressions do not have side-effects or error-breaks. The real puzzle, then, is understanding the logic of nonterminating expressions and procedures. This is easier than understanding expressions with side-effects and has been pretty satisfactorily worked out in the last fifteen years [26, 21, 10], though the last word has not been said (cf. [22]).

Puzzle 5 *Many programming languages allow procedures which can take themselves as arguments. But type-violations like self-application lead to contradictions in a few lines:*

> *Let* $P(f) =^{def}$ **if** $f(f) \neq 0$ **then** 0 **else** 1 **fi**. *By definition,* $P(f) \neq f(f)$ *for all functions* f. *Now letting* f *be* P *yields the contradiction* $P(P) \neq P(P)$*!*

What actually happens when $P(P)$ is evaluated in a language which allowed this sort of definition (e.g., LISP or ALGOL)? Why aren't these contradictions applicable in programming?

The most familiar genus of programming languages consists of *imperative* languages defining procedures with side-effects. This genus is immediately identifiable as the languages containing an *assignment statement* of the typical form $x := e$. Reasoning about "pure" side-effect-inducing procedures requires a special style as suggested by the next puzzles.

Puzzle 6 *After executing the assignment statement $x := 0$, it will be the case that $x = 0$. Replace x by an array reference $a[1]$, and it's still true. But replace x by an array reference like $a[a[1]]$, and it's not true anymore. Explain!*

Puzzle 7 *Suppose x and y are distinct identifiers of type integer. Exhibit a simple context into which either of the code fragments $x := 0; y := 1$ or $y := 1; x := 0$ can be substituted (as phrases of type* program, *cf. Puzzle 3) so that in most programming languages the resulting substitutions yield different results.*

Puzzle 8 *Let Q be a pure procedure identifier, i.e., a call of Q is made for its side-effects and returns no value. Argue that in block structured programming languages the block*

$$\begin{array}{l}\textbf{begin}\ x : \textbf{integer-var};\\ \quad x := 0;\\ \quad Q;\\ \textbf{end}\end{array}$$

ought to behave equivalently to the call Q. Now give several reasons why it might not.

Here the syntax **integer-var** indicates that the evaluation of G or x yields what in programming language jargon are called *variables*, i.e., memory locations, which contain integers. (Calling locations "variables" conflicts with mathematical usage, but is fairly standard in Computer Science.)

Readers who have been exposed to languages like LISP which use *dynamic* scoping conventions will find this puzzle too easy; so assume instead that the more familiar *static* scoping conventions are in use.

Joseph Stoy suggested the following lovely puzzle exposing the muddiness surrounding implicit type-coercions. In practice, the behavior of compilers on this kind of example is inconsistent and unpredictable, reflecting the fact that a satisfactory theory of data-type coercions, containments, and equivalences in programming has just begun to be developed (cf. [24, 20, 6].)

Puzzle 9 *In a programming language which treats integers as a subtype of reals and automatically coerces reals used in integer contexts by rounding down, exhibit a simple context which distinguishes the calls $P(x)$ and $Q(x)$ where P and Q are declared as follows:*

$$\begin{array}{ll}\textbf{procedure}\ P(y); & \textbf{procedure}\ Q(y);\\ \quad y : \textbf{integer-var}; & \quad y : \textbf{integer-var};\\ \quad \textbf{skip}; & \quad y := y\\ \textbf{end}\ P; & \textbf{end}\ Q\end{array}$$

Nondeterministic and concurrent programs are a rich source of puzzles. Researchers are busily seeking satisfactory explanations of nondeterminism, parallelism and concurrency in programming, and there is still debate about what the basic models should be [15, 18, 4, 8]. The next subtle puzzle was suggested by David Park, who observes that it is the simplest case of what is known as the Brock-Ackerman Anomaly.

Puzzle 10 *Consider a programming language with the nondeterministic primitive construct* **amb** *taking two integer arguments, with* $\mathbf{amb}(E, F)$ *yielding the same value as expression* E *or* F *if evaluation of* E *or* F *terminates, choosing nondeterministically to yield one of these values if both terminate. Now consider the recursive declaration*

$$\begin{array}{l}\textbf{integer procedure } C;\\ \quad \textbf{return}(\mathbf{amb}(0, \mathbf{max}(1, C)));\\ \textbf{end } C;\end{array}$$

Give an operational argument supporting the claim that the proper result of evaluating C *is* 0*, not* 1*!*

Puzzle 11 *The code fragments* $x := x$ *and* **skip** *are essentially equivalent in ordinary* sequential *programming languages in contexts where* x *denotes a variable (and no coercions occur, cf. Puzzle 9). What about in programming languages with* concurrent processes*?*

This last puzzle emphasizes, as did Puzzle 9, that a change in one part of a language, in this case the introduction of a new feature allowing instructions to run concurrently, can change basic properties of even the original phrases of the language which have not been altered. The following variation on Puzzle 7 provides another illustration of this point.

Puzzle 12 *Suppose* x *and* y *denote distinct variables (locations) of type integer. Exhibit a simple context in a programming language with concurrent processes which distinguishes the code fragments* $x := 0; y := 1$ *and* $y := 1; x := 0$*, even if assignments of the form* $w := z$ *are atomic actions.*

The final puzzle is the most purely logical of all.

Puzzle 13 *No set of true first-order formulas about arithmetic implies that the program*

$$\textbf{while } 0 < x \textbf{ do } x := x - 1 \textbf{ od}$$

always halts, because there are nonstandard *models of the integers with exactly the same first-order properties as the standard integers, and this program does not halt when* x *is a nonstandard positive integer. But the first-order theory of the integers (nonstandard ones too) allows induction, and it is easy to prove by induction from a few first-order axioms about the integers that this program halts. Explain!*

This puzzle clearly requires sophistication in formal logic to appreciate, let alone to resolve satisfactorily. My objective in including it in the list is to emphasize that various reasoning systems—equational, first-order, higher-order, etc.—have rich technical properties. My impression is that students in the automatic theorem-proving and program verification areas too often simply understand all these systems in a manipulative/algorithmic way, and are insensitive to their more abstract model- and proof-theoretic properties. The moral of the puzzle is that committing oneself to first-order reasoning is a technical decision which may have unexpected consequences.

2 The Solutions

Solution 1 *Let E cause* side-effects*, e.g.,*

$$\begin{array}{l}\textbf{integer procedure } E;\\ \quad x := x + 1;\\ \quad \textbf{return}(x);\\ \textbf{end } E.\end{array}$$

My answer to the real puzzle is that it probably isn't worthwhile trying to understand the "logic" of expressions with side-effects (see [5], however, for an attempt). Using such expressions strikes me as an unfortunate pun whereby expressions—which in familiar scientific use denote algebraic values—are reinterpreted as defining computational evaluation procedures. This wouldn't be so bad if there was a unique, natural way to evaluate expressions, but there isn't—evaluation strategies with different properties have been developed in various programming languages—and rather than try to figure out how to reason about systems in which E isn't equal to itself, I think it makes more sense to eliminate the pun: procedures with side-effects should not return values, and expressions which return values should not have side-effects [25]. The idea of expressions which both return values and have side-effects seems most deeply embedded in LISP. A recent study [11] provides preliminary confirmation of my suspicion that separating value-returning expressions from side-effect-inducing procedures will not hinder LISP-style programming. The gain in program comprehensibility from maintaining this separation seems substantial.

Solution 2 *Let e be* $1/0$.

In most programming languages, the attempt to divide by zero causes an "error-exception" which aborts evaluation. If sums are evaluated left-to-right, then $E + 1/0$ does not terminate, but $1/0 + E$ terminates with a "divide-by-zero" exception value.

Here again the contrast between the semantical view of an expression as denoting a value versus the operational view as denoting an algorithm yields an unexpected result.

Solution 3 *The context is* **print**(" ... ").

Again the solution is obvious, although even some experts stumble on it because one naturally expects the given phrases to be used in a context where they are treated as arithmetic expressions rather than text strings. Had this type-ambiguity been highlighted in the puzzle statement, presumably everyone would get it. So a reasonable attack on the *real* puzzle I have in mind, which is when two phrases such as those in the puzzle are equivalent for programming purposes, can be mounted by taking account of program phrase types.

I first thought that in languages with a **quote-eval** feature such as LISP [16] there was no type distinction between syntactic and executable objects. In fact **quote-eval** seems safe enough in LISP because, although **quote** syntactically *looks* like a *bona fide* operator, like say **cond**, it really doesn't behave like one (it only has behavior at parse time, not run time, e.g., one can't pass **quote** as a parameter to a procedure). Still, I have yet to see convincing examples where the **quote-eval** feature was worthwhile.

There is a less known construct in LISP known as **fexpr** which indeed allows dynamically created values to be converted back to syntactic objects (lists). The conceptually disastrous consequence of blurring the distinction between syntax and values in this way is that no two programs which differ syntactically in any respect are computationally equivalent!

Solution 4 *The first test is equivalent to $y = y$ which could reasonably be assumed equivalent to true, but the second is not since the evaluation of $G(1, y)$ will not terminate. Still, replacing the second test by true is an optimization which can only help, and could appropriately be done even though it does not preserve equivalence—the optimized program never disagrees with the original, but may terminate in more cases.*

Sophisticates may, by the way, question the remark that the test $y = y$ evaluates to *true*. In languages with binding rules (such as ALGOL's "call-by-name" [19]) by which the identifier y could itself be bound to an expression whose evaluation could fail to terminate (such as $G(1, z)$), the test $y = y$ might also fail to terminate.

Solution 5 *Naive solutions to this puzzle revolve around the observation that, in a programming language context, P is a* partial *function whose values at certain arguments may be undefined because they lead to nonterminating computations. Indeed, this is exactly what happens in evaluating $P(P)$ in any programming language in which this kind of definition is allowed. Since $P(P)$ is undefined, there is no contradiction in the conclusion that $P(P) \neq P(P)$.*

The naive explanation sounds OK, but really begs the question. To see that partialness is no explanation, just consider the modified definition

$$R(f) =^{def} \textbf{ if } f(f) \text{ is undefined } \textbf{or} f(f) \neq 0 \textbf{ then } 0 \textbf{ else } 1 \textbf{ fi}$$

which yields the same contradiction and cannot be explained away by claiming that $R(R)$ is undefined (since if it were, then it would equal 0, another contradiction).

The first step in resolving the contradiction alleged above comes by observing that it hinges on a prior understanding of the notion of function and application of a function to an argument. Ordinary mathematical functions are not self-applicable, so before pinpointing flaws in the reasoning above, the first obligation is explaining what objects are being reasoned about. What is the new notion of function—whether partial or total?

A rich mathematical theory of models of self-applicable functions has been developed in the past decade [27, 29, 28, 17]. (Here is where Category Theory, that most abstract of mathematical disciplines, engages with Computer Science, since it provides a persuasive general notion of function including self-applicable ones.) Some of these models are understandable with only a minimal mathematical background, but this is not the place to describe them in any detail. Enough to say that the theory admits the definition of P above as well-formed, but disallows the definition of R. The theory provides specialized rules appropriate for reasoning about values corresponding to nonterminating computations. For example, the theory confirms the computational fact that $P(P)$ denotes a nonterminating value, but the conditional **if** $0 = 0$ **then** 1 **else** $P(P)$ **fi**, which contains the nonterminating subexpression $P(P)$, *does* nevertheless terminate with value 1. The rules for nonterminating values lead to a distinction between the computable equality tests which can appear in programs and the usual mathematical equality between values. Anything, including a nonterminating value, is *mathematically* equal to itself, but the result of applying the *computable* equality test to a nonterminating value is the nonterminating value. From this perspective, the fallacy in the three line argument above can be identified as the confusion between the computable equality test occurring in the line which defines P and the mathematical equality test which occurs in the subsequent line.

Solution 6 *Suppose $a[1] = 1$ and $a[0] = 2$ initially. Then setting $a[a[1]]$ to 0 means setting $a[1]$ to 0, after which $a[1] = 0$, so $a[a[1]] = a[0] = 2$.*

The puzzle here revolves around another well-known programming language pun in which languages typically fail to maintain a syntactic distinction between use of an array reference $a[j]$ to denote a memory *location* and its use to denote the integer value which is stored in that location. As soon as one realizes the distinction, it becomes clear that when the location denoted by an expression on the lefthand side of an assignment statement depends on the memory store, then the expression may denote different locations before and after execution of the assignment instruction, and there may not be any relation between the contents of these locations.

More general puzzles here include explaining how to determine from expressions L which evaluate to locations and expressions I evaluating to integers, whether after executing $L := I$, it will the case that $\mathbf{contents}(L) = I$, and if not, just what assertions can be made about their values after the assignment [9, 14, 3]. Efficient mechanical procedures are known for making these kind of inferences for expressions involving arrays, **if...then...else**, and algebraic operators.

Solution 7 *The context is*

$$
\begin{array}{l}
\textbf{procedure } P(x, y); \\
\quad x, y : \textbf{integer-var}; \\
\quad \ldots; \\
\textbf{end } P; \\
\textbf{do } P(z, z)
\end{array}
$$

In the case that the first code fragment is substituted for the ellipsis, the effect is to set z to 1*, while in the second case z gets set to* 0*.*

Once one realizes that distinct identifiers x and y can be *aliases* for the same location, whatever surprise this puzzle may elicit quickly disappears. Still, the expectation that distinct identifiers denote distinct locations has seemed so reasonable that at least one programming language, EUCLID [23], was designed with syntactic restrictions which guarantee that such shared references cannot appear, e.g., procedure calls of the form $P(z, z)$ are forbidden.

Here I feel reasonably optimistic that, with a little care in keeping track of the type distinction between locations and their contents, one can work out an understandable logic of sharing without suffering the awkward language restrictions apparently necessary to prevent it [30].

Solution 8 *This is one of the deeper puzzles and even the experts don't agree on the answers.*

The intuitive idea of local storage allocation in blocks is that **begin** x : **integer-var** *causes allocation of a "new" location denoted by x until the* **end** *of the block. With this idea in mind, setting the new location x to* 0 *shouldn't have any effect on Q, so the block and call above clearly ought to be equivalent.*[1] *On the other hand, local*

[1]...unless the language observes dynamic scoping. In this case, if x occurs in the body of the declaration of Q, then within the block, Q can read the "new" x and discover that it has been set to 0. Dynamic scope is the basic mechanism of most LISP dialects, but finds it way into features of other languages as well. Dynamic scoping can be implemented a bit more efficiently than static scoping and is commonly viewed as more convenient for writing systems with numerous modules which may be independently updated—a view which is questioned by the designers of SCHEME, a LISP dialect which adopts static scoping [1]. Under dynamic scope, basic logical rules about bound variables fail, leading to the so-called *funarg* problems in LISP and similar languages.

storage is usually implemented by keeping a stack or list of "free" locations which can be allocated as needed. For efficiency, languages usually adopt a "stack discipline" or "heap-storage with garbage collection" procedure to recover allocated locations when they are no longer accessible in an ongoing computation.

With "free-list" semantics, the block leaves the machine in a different state than the simple call Q because the free-list has gotten shorter, although the new state could be regarded as equivalent to the old one since the new allocated location will eventually be restored by a garbage collector. Stack discipline avoids this by restoring the stack immediately on block exit. But in either case, when we consider the possibility of running out of free storage, the block is not equivalent to the call because the block might generate a storage overflow error where the call does not.

Acknowledging the possibility of storage overflow, however, is really an efficiency consideration which it would be nice to keep separate from the question of whether the program is correct assuming its time and storage requirements were met. By abstractly modeling the stack or free-list as an infinite list, we ought to be able to reach the desired conclusion that the block and the call are equivalent. But a more subtle and interesting question remains: what property of the locations on the free-list makes them free*? For example, how does the allocator know that Q doesn't refer to the top item on the free-list? Suppose Q was itself a library routine written in machine language which had access to all of memory, e,g,, if Q was itself the garbage collector? What does the allocator use as a "free" location then? Moreover, programming languages all too frequently allow integers to be used as memory references. For example, a procedure might calculate some integer, say* 7462 *(octal), and then assign some value to the* 7462^{nd} *memory location which might well be on the free-list. In fact, it really isn't possible to allocate new locations for such Q, and if we take them into consideration, then the equivalence proposed in the puzzle fails. So the equivalance holds for some languages, for example ALGOL-like languages, but fails for others, for example, the language C.*

So the *real* puzzle here is to determine what properties of languages are compatible with the expected properties of local storage such as the equivalence between the block and the call above. Note that part of this real puzzle involves formulating a theory which specifies just what properties are expected of local storage. An approach to this puzzle has only recently been proposed [12].

To understand more of the real puzzle, consider the block

$$\begin{array}{l}\textbf{begin } x : \textbf{integer-var};\\ \quad \textbf{procedure } P;\\ \quad\quad x := x + 2;\\ \quad \textbf{end } P;\\ \quad x := 0;\\ \quad Q(P);\\ \quad \textbf{if } \mathit{even}(x) \textbf{ then run-forever else } \ldots \textbf{ fi}\\ \textbf{end}\end{array}$$

Here we want to argue that, because x is "new" as far as Q is concerned, Q can only affect x indirectly by executing P. Since x is initialized to 0 and P only adds 2 to it, it must therefore contain an even integer if and when the call $Q(P)$ terminates. Thus the block above is equivalent to **run-forever** in any ALGOL-like context in which Q is declared. But it can be proved that none of the currently proposed methodologies for reasoning about programs provides rules by which this equivalence can be verified!

Solution 9 *The context is*

$$\begin{array}{l}\textbf{begin } x : \textbf{real-var};\\ \quad x := 0.1;\\ \quad \ldots;\\ \quad \textbf{return}(x + 0.2);\\ \textbf{end}\end{array}$$

As typically compiled, a call $P(x)$ would have no effect, viz., would be the same as **skip**. *Hence, when $P(x)$ is substituted for the ellipsis, the value returned would simply be* $0.1 + 0.2 = 0.3$. *On the other hand, if $Q(x)$ is substituted, the real value* 0.1 *contained in x would presumably be rounded to* 0 *in the process of being assigned to the integer variable y, and the final value returned would be* $0 + 0.2 = 0.2$.

An explanation for the misbehavior here might be laid to the fallacious deduction that because integers automatically coerce into real, so too integer *variables* should coerce into real *variables* (if anything, the reverse coercion should occur). In general, rules for implicit coercions are not well understood even in contexts without variables.

Solution 10 *In evaluating* $\mathbf{amb}(0, \mathbf{max}(1, C))$ *as the first subgoal in the evaluation of C, the only way the recursive call to C within the second branch could terminate would be* after *the value of C had been committed to be* 0 *by the choice of the first branch in some more deeply nested recursive call. Note that backtracking to finish computing the value* 1 *for the second branch after a value for the recursive call has been found is not appropriate: the only point in backtracking would be to find a value for C, but the backtracking is possible only after a value has already been found to be* 0.

Having presented this argument, let me admit some doubt about its persuasiveness. Slightly more complex examples using *streams* better motivate the answer, cf. [8].

Solution 11 *The answer depends on whether the assignment is regarded as an atomic action, or is divisible into more primitive atomic steps such as fetching the contents of x and then copying the fetched value back into x. In the second case, $x := x$ is not equivalent to* **skip***, since if the assignment is run concurrently with a procedure for incrementing x, there is a possibility that the incrementing procedure will take effect after the fetch and before the copy, with the result that when the copy phase completes, the contents of x is restored to its original value. On the other hand, running concurrently with an incrementing procedure leaves no possibility that the contents of x will remain unchanged.*

Solution 12 *The context is* **cobegin** ... **and** $x := y$ **coend**. *The reasoning is much the same as for Puzzle 11.*

Solution 13 *Although induction is a sound way to try to prove* any *assertion about the integers, it is only sound for proving* first-order *assertions about nonstandard models of the integers. The assertion that the program terminates, however, is not expressible as a first-order assertion [13].*

There is a subtlety here which can be confusing. Using the technique of Gödel-numbering, one can construct a formula of the first-order language of the integers which, *interpreted over the standard model of the integers* asserts that the program above terminates for all x. Moreover, this formula, being first-order and true of the integers—indeed easily provable by induction—is also true in all nonstandard integer models. Unfortunately, when this or any such formula is interpreted over nonstandard integers, it *no longer is equivalent to the assertion that the program terminates*! (The "nonstandard" notion of termination which the formula does define has been studied in [2, 7].)

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[2] H. Andreka, I. Nemeti, and I. Sain. A complete logic for reasoning about programs via nonstandard model theory. *Theor. Comput. Sci.*, 17:193–212, 259–278, 1982.

[3] K. Apt. Ten years of Hoare's logic: a survey, part I. *ACM Trans. Prog. Lang. Syst.*, 3:431–483, 1981.

[4] K. Apt. Ten years of Hoare's logic: a survey, part II: nondeterminism. In J. De Bakker and J. van Leeuwen, editors, *Distributed Systems: Part 2, Semantics and Logic*, volume 159 of *Foundations of Computer Science IV, Mathematical Centre Tracts*, pages 101–132. Mathematisch Centrum, Amsterdam, 1983.

[5] H. Boehm. A logic for expressions with side effects. In 9^{th} *ACM Symp. on Principles of Programming Languages*, pages 268–280. 1982.

[6] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types. Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[7] R. Cartwright. Recursive programs as definitions in first order logic. *SIAM J. Comput.*, 13:374–407, 1984.

[8] J. de Bakker, J. Meijer, and J. Zucker. Bringing color into the semantics of nondeterministic dataflow. Extended abstract, Centre for Mathematics and Computer Science, Amsterdam., 1985.

[9] R. Floyd. Assigning meaning to programs. In J. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, pages 19–32. AMS, 1967.

[10] M. J. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanical Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[11] M. Hailperin. What price for eliminating expression side-effects? MIT/LCS/Technical Memo 281, MIT, 1985.

[12] J. Y. Halpern, A. R. Meyer, and B. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In 11^{th} *ACM Symp. on Principles of Programming Languages*, pages 245–257. 1984.

[13] P. Hitchcock and D. Park. Induction rules and termination proofs. In M. Nivat, editor, *Automata, Languages and Programming*, pages 225–251. North-Holland/American Elsevier, 1973.

[14] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[15] C. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, 1978.

[16] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Commun. ACM*, 3:184–195, 1960.

[17] A. R. Meyer. What is a model of the lambda calculus? *Information and Computation*, 52:87–122, 1982.

[18] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[19] P. Naur. Revised report on the algorithmic language ALGOL 60. *Commun. ACM*, 6:1–20, 1963.

[20] F. J. Oles. *A category-theoretic approach to the semantics of programming languages.* PhD thesis, Syracuse Univ., 1982.

[21] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[22] G. D. Plotkin. Types and partial functions. Manuscript in preparation, Univ. of Edinburgh., 1986.

[23] G. Popek, J. Horning, B. Lampson, J. Mitchell, and R. London. Notes on the design of Euclid. In *ACM Conf. on Lang. Design for Reliable Software.* SIGPLAN Notices **12**, no. 3, 1977.

[24] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. Jones, editor, *Proc. Aarhus Workshop on Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer Verlag, 1980.

[25] J. C. Reynolds. The essence of ALGOL. In *Algorithmic Languages*, pages 345–372. IFIP, North-Holland, 1981.

[26] D. S. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Manuscript, Oxford Univ., 1969.

[27] D. S. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–587, 1976.

[28] D. S. Scott. Domains for denotational semantics. In 9^{th} *International Coll. on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[29] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[30] B. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer. From denotational to operational and axiomatic semantics. In *Logic of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 474–500. Springer-Verlag, 1984.