Norman Ramsey

Where Theory and Practice Meet: POPL Past and Future

John C. Reynolds
Carnegie Mellon University

January 19, 1998

Program Design

A good programming language ... should assist in establishing and enforcing conventions and disciplines that will ensure harmonious co-operation of the parts of a large program when they are developed separately and finally assembled together.

Programming Documentation

A good programming language will encourage and assist the programmer to write clear self-documenting code, and even perhaps to develop and display a pleasant style of writing. The readability of programs is immeasurably more important than their writeability.

— C. A. R. Hoare, "Hints on Programming Language Design", POPL I.

Variables and References

In a high-level language, the programmer is deprived of the dangerous power to update his own program while it is running. Even more valuable, he has the power to split his machine into a number of separate variables, arrays, files, etc.; when he wishes to update any of these he must quote its name explicitly on the left of the assignment, so that the identity of the part of the machine subject to change is immediately apparent; and, finally, a high-level language can guarantee that all variables are disjoint, and that updating any one of them cannot possibly have any effect on any other.

⋮

[In contrast,] references are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover.

— C. A. R. Hoare, "Hints on Programming Language Design", POPL I.

Three Themes in Programming Language Research

- Evaluation Order
- Types
- Program Specification and Proof

Evaluation Order (Call by Name versus Call by Value)

λ-calculus	(Church,Rosser)	Name	1936
λ-calculus	(Bernay)	Value	1936
Algol 60	(Naur et al.)	Name	1960
LISP	(McCarthy et al.)	Value	1960
ISWIM	(Landin)	Value	1964
Algol 68	(van Wijngaarden et al.)	Value	1968
Domains	(Scott)	Name	1970
Graph Reduction	(Wadsworth)	Name	1971
SCHEME	(Sussman, Steele)	Value	1975
Lazy Evaluation	(Henderson, Morris, Friedman, Wise)	Name	1976
ML	(Milner et al.)	Value	1977
Combinators	(Turner)	Name	1979
Miranda	(Turner)	Name	1985
Standard ML	(Milner et al.)	Value	1990
Monads	(Moggi)		1991
Haskell	(Hudak et al.)	Name	1992
Stack-implemented ML	Tofte, Talpin	Value	1994

... the reviewer suggests that Rules II and III of conversion might be restricted by the condition that the part N of $\{\lambda x.M\}(N)$ must be in normal form. Under this restriction both theorems on the normal form remain valid even in the case of the extension [to the λK-calculus], and at the same time the proofs of the theorems become simpler.

— Paul Bernays, Review of "Some Properties of Conversion" by Alonzo Church and J. B. Rosser, 1936.

Fiction

Programming languages were developed to program computers.

Fact

Computers were developed to implement programming languages.

4.7.3.1. Value assignment (call by value). All formal parameters quoted in the value part of the procedure declaration heading are assigned the values ... of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. ...

4.7.3.2. Name replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses whenever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. ...

— Naur et al., "Revised Report on the Algorithmic Language Algol 60", 1963.

Landin's Definition of Call by Name

If all the identifiers involved are straightforward the treatment of subscripted identifiers and function designators is as follows.

$$A[i+j,k] \qquad A(i+j,k)$$
$$f(x+y,z) \qquad f(\lambda().x+y,\lambda().z)$$

The transformation of actual parameters into none-adic functions is associated with the possibility that a procedure might call its formals by name. It is complemented by a peculiarity in the treatment of procedure declarations as noted below.

If the identifiers involved are all formals called by name than the transcription to IAEs is

$$A()(i()+j(),k())$$
$$f()(\lambda().x()+y(),\lambda().z())$$

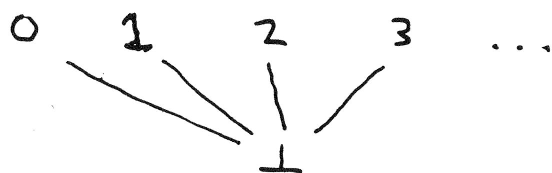
— Peter Landin, "A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I", 1965.

The Argument for Call by Value

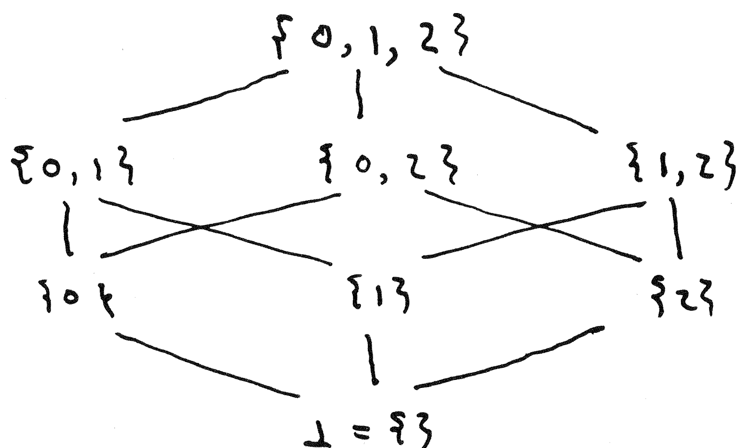
Lazy evaluation says that $zero(E) = 0$ even if E fails to terminate. This flies in the face of mathematical tradition: an expression is meaningful only if all its parts are.

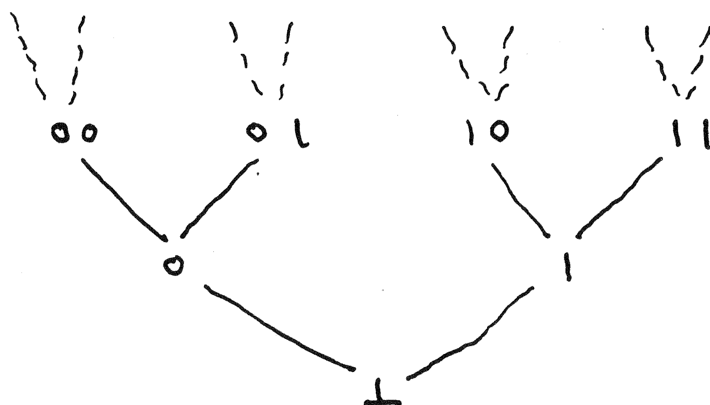
— L. C. Paulson, *ML for the Working Programmer*, 2nd edition, p. 47, 1996.

A Flat Domain



Some More Interesting Domains





In particular, a sub-graph which is a redex can represent a set of redices in the linear expansion; by contracting the one redex in the graph we can thus effect contraction of several (identical) redexes *simultaneously*.

— C. Wadsworth, *Semantics and Pragmatics of the Lambda-Calculus*, Ph.D. Thesis, p. 138, 1971.

A Goal for the Future

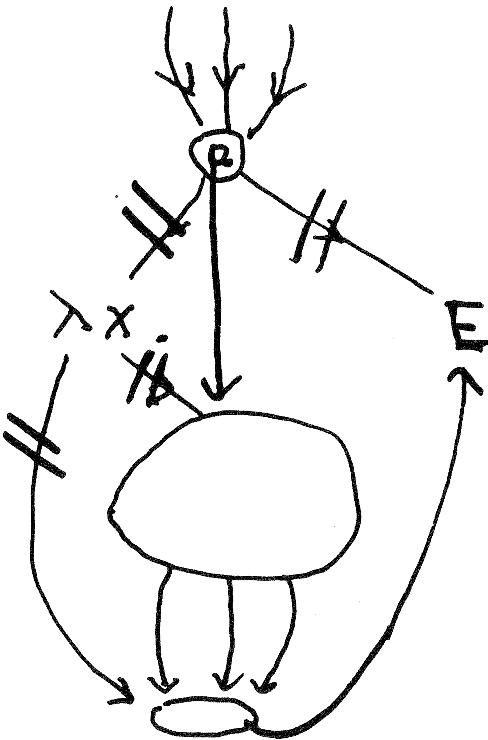
(my own hobby horse)

The advantages of high-level languages:

- Higher-order procedures
- Type checking
- Modularity and Abstraction

must be extended to low-level languages.

Graph Reduction is High Level



Monads (Eugenio Moggi)

$\theta = \text{int} \mid \theta \to \theta \mid T\theta$

$$\frac{\pi \vdash e : \theta}{\pi \vdash [e] : T\theta} \qquad \frac{\pi \vdash e : T\theta \quad \pi, x : \theta \vdash e' : T\theta'}{\underline{\text{letval}}\ x = e\ \underline{\text{in}}\ e' : T\theta'}$$

$$\begin{aligned}
\text{multv} \equiv\ & \underline{\text{rec}}\ \lambda f : \text{int} \to \text{int} \to T\text{int}.\ \lambda m : \text{int}.\ \lambda n : \text{int} \\
& \quad \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ [0]\ \underline{\text{else}} \\
& \qquad \underline{\text{letval}}\ r = f\ m\ (n-1)\ \underline{\text{in}}\ [r+m]
\end{aligned}$$

$$\begin{aligned}
\text{multn} \equiv\ & \lambda mt : T\text{int}.\ \lambda nt : T\text{int}. \\
& \quad \underline{\text{letval}}\ m = mt\ \underline{\text{in}}\ \underline{\text{if}}\ m = 0\ \underline{\text{then}}\ [0]\ \underline{\text{else}} \\
& \qquad \underline{\text{letval}}\ n = nt\ \underline{\text{in}}\ \text{multv}\ m\ n
\end{aligned}$$

But we also want :

$$\begin{aligned}
\text{multnp} \equiv\ & \lambda mt : T\text{int}.\ \lambda nt : T\text{int}. \\
& \quad \underline{\text{letval}}\ m = mt \parallel n = nt\ \underline{\text{in}}\ \text{multv}\ m\ n
\end{aligned}$$

Types

Typed Set Theory	(Russell)	1908
Simply Typed Lambda Calculus	(Church)	1940
FORTRAN	(Backus et al.)	1957
Algol 60	(Naur et al.)	1960
Algol 68	(van Wijngaarden et al.)	1968
Type Inference	(Hindley)	1969
System F	(Girard)	1971
Types are not Sets	(Morris)	1972
Polymorphic Lambda Calculus	(Reynolds)	1974
ML	(Milner et al.)	1977
Intersection Types	(Coppo et al., Sallé)	1978
Subtypes and Generic Operators	(Reynolds)	1980
Coercion and Type Inference	(Mitchell)	1984
Existential Abstract Types	(Mitchell, Plotkin)	1985
Bounded Fun	(Cardelli, Wegner)	1985
Dependent Types and Modularity	(MacQueen)	1986
Linear Types	(Girard)	1987
SML	(Milner et al.)	1990
Typed Assembly Language	(Morrisett et al.)	1998

The Importance of Types

Apart from the elimination of the risk of error, the concept of type is of vital assistance in the design and documentation phases of program development. The design of abstract and concrete data structures is one of the first tools for refining our understanding of problems, and for defining the common interfaces between the parts of a large program. The declaration of the name and structure or range of values of each variable is a most important aspect of clear programming, and the formal description of the relationship of each variable to other program variables is a most important part of its annotation; and finally an informal description of the purpose of each variable and its manner of use is a most important part of program documentation.

— C. A. R. Hoare, "Hints on Programming Language Design", POPL I.

The Value of Typechecking

... when a programmer undertakes to implement a class of abstract objects to be used by many other programmers ... h usually proceeds by choosing a representation for the objects in terms of other objects and then writes the required operations to manipulate them.

There are two problems ... . First, users of his programs may ask them to operate on a value that is not a valid representation of any of the objects he has undertaken to process. ... Second, users may write programs that depend upon the particular representation he chooses for objects. ...

Type checking is a way to prevent such things from happening. All values used to represent the abstract objects are considered to be of a certain type. The rules are:

1. Only values of that type can be submitted for processing (authentication).
2. Only the procedures given can be applied to objects of that type (secrecy).

— J. H. Morris, "Types are not Sets", POPL I.

Closures have Existential Type

$\lambda x : int.\ x \times y + z$

in the environment $y : 4,\ z : 5$ has the value

$\langle \lambda \langle y{:}int,\ z{:}int \rangle .\ \lambda x{:}int.\ x \times y + z,\ \langle 4, 5 \rangle \rangle$

of type $((int \times int) \to int \to int) \times (int \times int)$.

$\lambda x : int.\ \underline{if}\ b\ \underline{then}\ x\ \underline{else}\ y$

in the environment $b : \underline{true},\ y : 10$ has the value

$\langle \lambda \langle b : bool,\ y : int \rangle .\ \lambda x : int.\ \underline{if}\ b\ \underline{then}\ x\ \underline{else}\ y,\ \langle \underline{true}, 10 \rangle \rangle$

of type $((bool \times int) \to int \to int) \times (bool \times int))$.

The common type is:

$\exists \tau .\ (\tau \to int \to int) \times \tau$.

Fancy Types are Difficult

System	Type Checking	Type Inference
ML polymorphism	efficient	intractable
Full polymorphism	efficient	impossible
Intersection types	intractable	impossible
Bounded polymorphism	impossible	

Imperative Types

If types are propositions, what is the type of a Hoare triple?

— Jeannette Wing, around 1995

Example: Since $x := x + 1$ satisfies

$$even(x)\,\{x := x + 1\}\,odd(x),$$

does $x := x + 1$ have the type

$$even(x) \rightarrow odd(x)?$$

If so, then a command can change the type of the state.

From Imperative to Linear Functional Programs

The ordinary functional translation of

$$a[0] := a[1] + 1$$

is

$$\lambda a.\ update(a, 0, lookup(a, 1) + 1),$$

where

$$update : array \times int \times int \to array$$
$$lookup : array \times int \to int,$$

which has the same type as

$$\lambda a.\ update(a, 0, lookup(sort\ a, 1)).$$

The linear translation is

$$\lambda a.\ \underline{let}\ (a', v) = lookup(a, 1)\ \underline{in}\ update(a', 0, v+1),$$

where

$$update : array \otimes int \otimes int \multimap array$$
$$lookup : array \otimes int \multimap array \otimes int,$$

which has the type $array \multimap array$.

More Flexible Linear Programs

```
quicksort = λa.
    let (a', l) = length a in if l ≤ 1 then a'
      else let (a'', t) = lookup (a', 1) in
        let (al, am, ar) = partition (a'', t) in
          conc (quicksort al, conc (am, quicksort ar))
```

where

$$\begin{aligned}
\text{length} &: \text{array} \multimap \text{array} \otimes \text{int} \\
\text{lookup} &: \text{array} \otimes \text{int} \multimap \text{array} \otimes \text{int} \\
\text{partition} &: \text{array} \otimes \text{int} \multimap \text{array} \otimes \text{array} \otimes \text{array} \\
\text{conc} &: \text{array} \otimes \text{array} \multimap \text{array}
\end{aligned}$$

A Problem

How do we know quicksort is in-place?

A Problem: Freezing

The translation of

```
a[0] := a[1] + a[2]
```

is

```
λa. let (a', v) = lookup (a, 1) in
    let (a'', w) = lookup (a', 2) in
    update (a'', 0, v + w).
```

But we would prefer

```
λa. let (a', v) =
        freeze a in lookup (a, 1) + lookup (a, 2)
    in update (a', 0, v),
```

where

update : $array \otimes int \otimes int \multimap array$

lookup : $(¡array) \otimes int \multimap int$.

Program Specification and Proof

Brouwer (Constructive Mathematics)	1913
Heyting (Constructive Mathematics)	1930
Goldstine and von Neumann	1947
Turing	1950
Naur	1966
Floyd	1967
Hoare (Axiomatic Basis)	1969
Curry-Howard Isomorphism	1969
Hoare (Proof of a Program: FIND)	1971
Dijkstra (Weakest Preconditions)	1975
Milner (LCF)	1977
Martin-Löf (Intuitionistic Type Theory)	1984
Huet (Theory of Constructions)	1984
Constable (Nuprl)	1984
Necula (Machine-Language Proofs)	1997

Program Specification

In many cases, the validity of the results of a program (or part of a program) depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q) and a description of the result of its execution (R), we introduce a new notation:

$$P\,\{Q\}\,R.$$

This may be interpreted 'If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion.

— C. A. R. Hoare, "A Axiomatic Basis for Computer Programming", 1969.

Program Proof

In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

— C. A. R. Hoare, "Proof of a Program: *Find*", 1971.

The Curry-Howard Isomorphism

$(-)^* \in$ Propositional Calculus $\to$ Simple Types

$(A \text{ \underline{and} } B)^* \mapsto A^* \times B^*$

$(A \text{ \underline{or} } B)^* \mapsto A^* + B^*$

$(A \text{ \underline{implies} } B)^* \mapsto A^* \to B^*$

Then $e \in A^*$ is a proof of A.

Extracting Programs from Proofs

... Given a constructive existence proof the system can use the computational information in the proof to build a representation of the object which demonstrates the truth of the assertion. ... Moreover, a proof that for any object x of type A we can build an object y of type B satisfying relation $R(x, y)$ implicitly defines a computable function f from A to B. The system can build f from the proof, and it can evaluate f on inputs of type A.

— R. L. Constable, *Implementing Mathematics with the Nuprl Proof Development System*, 1986

Safe Programming Languages

A programming language is <u>safe</u> if no program accepted by its compiler can "run wild".

Having a good proof theory is not an adequate substitute for safety.

Safe Programming Languages

A programming language is _safe_ if no program accepted by its compiler can "run wild".

Having a good proof theory is not an adequate substitute for safety.

Program Debugging

Firstly, the notations should be designed to reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler, before the program even begins to run. Certain programming errors cannot always be detected in this way, and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine- or implementation-dependent effects, which are inexplicable in terms of the language itself.

Security

... it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?

— C. A. R. Hoare, "Hints on Programming Language Design", POPL I.