Fundamental Concepts of Programming Languages

Christopher Strachey

March 16, 2015

Context

- Notes prepared for lecture series given in 1967
- Lecture notes circulated in unpublished form until 2000
- Published as a paper in 2000

Section 1 - Preliminaries

- A philosophical discussion of why the study of programming languages matters

Section 1.1 - Introduction

- Formal jargon needs to be introduced because core concepts are so vague

Section 1.2 - Philosophical considerations

- An aside focused on philosophical “outlooks”
- Focuses on constructivist vs classical mathematics
- Some of this material seems irrelevant to me
- But I like the clear exposition of Strachey’s aesthetic

Strachey's Aesthetic

The advantages of rigour lie, not surprisingly, almost wholly with those who require construction rules.

Strachey's Aesthetic

I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development.

In these terms the urgent task in programming languages is to explore the field of semantic possibilities.

Strachey's Aesthetic

We are still intellectually at the stage that calculus was at when it was called the 'Method of Fluxions' and everyone was arguing about how big a differential was.

Strachey's Aesthetic

If we attempt to formalise our ideas before we have really sorted out the important concepts, the result, though possibly rigorous, is of very little value — indeed it may well do more harm than good by making it harder to discover the really important concepts. Our motto should be 'No axiomatisation without insight'.

However, it is equally important to avoid the opposite of perpetual vagueness.

Section 2 - Basic concepts

- The core PL concepts start being presented here
- I've translated examples to pseudo-Julia for clarity

Section 2.1 - Assignment statements

A computer has an “abstract store” with over-writable memory.

Basic assignment statements

```
x = 3
x = y + 1
x = x + 1
```

Complex assignment statements

```
i = a > b ? j : k
A[i] = A[a > b ? j : k]
A[a > b ? j : k] = A[i]
a > b ? j : k = i
```

General form

$$\epsilon_1 = \epsilon_2$$

Section 2.2 - L-values and R-values

L-value for the address-like object appropriate on the left of an assignment, and R-value for the contents-like object appropriate for the right.

The two essential features of a location are that it has a content - i.e. an associated R-value - and that it is in general possible to change this content by a suitable updating operation.

An L-value represents an area of the store of the computer

Not just a pointer! That comes later.

Section 2.3 - Definitions

Initial declaration of a new L-value:

```
p = 3.5
```

Aliasing declaration of a new L-value:

```
alias(q, p)
```

This is L-value aliasing, not R-value aliasing:

```
i = 2
alias(x, M[i, i])
i = 3
```

Section 2.4 - Name

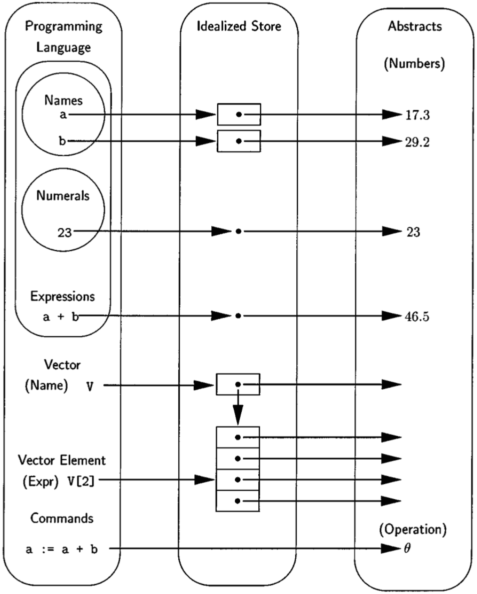
- This is essentially what most of us now call identifiers
- Confusion created by Algol's concept of “call by name”

Section 2.5 - Numerals

- Numeric literals, potentially written using different bases
- Strachey calls this “microsyntax”

Section 2.6 - Conceptual model

- Box and contents representation of L-values and R-values


Programming Language
Idealized Store
Abstracts
(Numbers)
Names
a
b
17.3
29.2
Numerals
23
23
Expressions
a + b
46.5
Vector
(Name) V
Vector Element
(Expr) V[2]
Commands
a := a + b
(Operation)
θ

Section 3 - Conceptual constructs

- Lays out definitions of expressions and commands

Section 3.1 - Expressions and commands

Expressions:

```
1
1 + 1
sin(1 + x + y / x^2)
```

Section 3.1 - Expressions and commands

Commands:

```
x = 1
y = 1 + 1
z = sin(1 + x + y / x^2)
```

Section 3.1 - Expressions and commands

- Expressions are pure and commands are impure
- Expressions involve only R-values
- Commands involve at least one L-value

Section 3.1 - Expressions and commands

Divide between expressions and commands is often blurry:

```
x = i++
```

Section 3.1 - Expressions and commands

To a large extent it is true that the increase in power of programming languages has corresponded to the increase in the size and complexity of the right hand sides of their assignment commands for this is the situation in which expressions are most valuable. In almost all programming languages, however, commands are still used and it is their inclusion which makes these languages quite different from the rest of mathematics.

Section 3.2 - Expressions

- Detailed examination of how expressions work

Section 3.2.1 - Values

The characteristic feature of an expression is that it has a value. We have seen that in general in a programming language, an expression may have two values — an L-value and an R-value. In this section, however, we are considering expressions in the absence of assignments and in these circumstances L-values are not required. Like the rest of mathematics, we shall be concerned only with R-values.

Referential Transparency

One of the most useful properties of expressions is that called by Quine **referential transparency***. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value.*

Referential Transparency

Examples:

```
sin(6)
sin(1 + 5)
sin(30 / 5)
```

Enables local and ordered reasoning about code.

Section 3.2.2 - Environments

An expression must be understood with reference to an **environment**, which provides the meaning of all variables.

Section 3.2.2 - Environments

Haskell's `where`:

```
a + 3/a where a = 2 + 3/7
a + b - 3/a where a = b + 2/b
```

Section 3.2.2 - Environments

```
let a = 2 + 3/7
    a + 3/a
end

let a = b + 2/b
    a + b - 3/a
end
```

Section 3.2.2 - Environments

$$(\lambda a.a + 3/a)(2 + 3/7)$$

$$(\lambda a.a + b - 3/a)(b + 2/b)$$

Section 3.2.2 - Environments

All three methods are exactly equivalent and are, in fact, merely syntactic variants whose choice is a matter of taste. In each the letter a is singled out and given a value and is known as the **bound variable**. *The letter b in the second expression is not bound and its value still has to be found from the environment in which the expression is to be evaluated. Variables of this sort are known as* **free variables**.

Section 3.2.3 - Applicative structure

Writing out S-expressions:

`a + b` corresponds to `+(a, b)`

`a + 3 / a` corresponds to `+(a, /(3, a))`

Section 3.2.3 - Applicative structure

In the examples so far given all the operators have been either a λ-expression or a single symbol, while the operands have been either single symbols or sub-expressions. There is, in fact, no reason why the operator should not also be an expression. Thus for example if we use `D` for the differentiating operator, `D(sin) = cos` so that `(D(sin))(*(3, a))` is an expression with a compound operator whose value would be `cos(3 * a)`.

Section 3.2.4 - Evaluation

1. Evaluate the operator and the operand(s) in any order.
2. After this has been done, apply the operator to the operand(s).

This defines a partial order on expressions: which expressions *must* be evaluated before others

Section 3.2.4 - Evaluation

Many languages make (1) more formal and choose an order

Language features that allow us to express partial orders are much weaker

Section 3.2.5 - Conditional expressions

There is one important form of expression which appears to break the applicative expression evaluation rule. A conditional expression such as: `x = 0 ? 0 : 1/x`

Section 3.3 - Commands and sequencing

Ordering of evaluation becomes more important with commands

Section 3.3.1 - Variables

One important characteristic of mathematics is our habit of using names for things. Curiously enough mathematicians tend to call these things 'variables' although their most important property is precisely that they do not vary.

Section 3.3.1 - Variables

The cost of doing this is considerable. We are obliged to consider carefully the relationship between L- and R-values and to revise all our operations which previously took R-value operands so that they take L-values. I think these problems are inevitable and although much of the work remains to be done, I feel hopeful that when completed it will not seem so formidable as it does at present, and that it will bring clarification to many areas of programming language study which are very obscure today. In particular the problems of side effects will, I hope, become more amenable.

Section 3.3.2 - The abstract store

- Introduces a σ-notation for relating L-values and R-values
- σ is essentially an immutable symbol table

Section 3.3.3 - Commands

Definition of commands in terms of symbol table derivations

Section 3.4 - Definition of functions and routines

- Functions are pure, portable expressions
- Routines involve impure, portable commands

Section 3.4.1 - Functional abstractions

```
f(x) = 5 * x^2 + 3 * x + 2 / x^3

f = x -> 5 * x^2 + 3 * x + 2 / x^3
```

Section 3.4.2 - Parameter calling modes

When the function is used (or called or applied) we write $f(e)$ where e can be an expression. If we are using a referentially transparent language all we require to know about the expression e in order to evaluate $f(e)$ is its value. There are, however, two sorts of value, so we have to decide whether to supply the R-value or the L-value of e to the function f. Either is possible, so that it becomes a part of the definition of the function to specify for each of its bound variables (also called its formal parameters) whether it requires an R-value or an L-value. These alternatives will also be known as calling a parameter by value (R-value) or reference (L-value).

Section 3.4.2 - Parameter calling modes

Existing programming languages show a curious diversity in their modes of calling parameters. FORTRAN calls all its parameters by reference and has a special rule for providing R-value expressions such as `a + b` *with a temporary L-value. ALGOL60, on the other hand, has two modes of calling parameters (specified by the programmer): value and name. The ALGOL call by value corresponds to call by R-value as above; the call by name, however, is quite different (and more complex). Only if the actual parameter (i.e., the expression* `e` *above) is a simple variable is the effect the same as a call by reference. This incompatibility in their methods of calling parameters makes it difficult to combine the two languages in a single program.*

Section 3.4.2 - Thunks

ALGOL 60 call by name
Let f be an ALGOL procedure which calls a formal parameter x by name. Then a call for f with an actual parameter expression e will have the same effect as forming a parameterless procedure `() -> e` and supplying this by value to a procedure `f'` which is derived from f by replacing every written occurrence of x in the body of f by `x()`. The notation `() -> e` denotes a parameterless procedure whose body is e while `x()` denotes its application (to a null parameter list).

Section 3.4.3 - Modes of free variables

The obscurity which surrounds the modes of calling the bound variables becomes much worse when we come to consider the free variables of a function. Let us consider for a moment the very simple function:

```
f(x) = x + a
```

Free variable by R-value

```
a = 3
f(x) = x + a
# f(5) => 8
a = 10
# f(5) = 8
```

Free variable by L-value

```
a = 3
f(x) = x + a
# f(5) => 8
a = 10
# f(5) = 15
```

Section 3.4.4 - Own variables

This is essentially a discussion of what C calls static variables

Section 3.4.4 - Own variables

Can be imitated using closures:

```
function make_f()
    i = 0
    function f()
        i += 1
        return i
    end
    return f
end
```

Section 3.4.5 - Functions and routines

Functions and routines are as different in their nature as expressions and commands. It is unfortunate, therefore, that most programming languages manage to confuse them very successfully. The trouble comes from the fact that it is possible to write a function which also alters the store, so that it has the effect of a function and a routine. Such functions are sometimes said to have side effects and their uncontrolled use can lead to great obscurity in the program. There is no generally agreed way of controlling or avoiding the side effects of functions, and most programming languages make no attempt to deal with the problem at all—indeed their confusion between routines and functions adds to the difficulties.

Section 3.4.5 - Functions and routines

Any departure of R-value referential transparency in a R-value context should either be eliminated by decomposing the expression into several commands and simpler expressions, or, if this turns out to be difficult, the subject of a comment.

Section 3.4.6 - Constants and variables

There is another approach to the problem of side effects which is somewhat simpler to apply, though it does not get round all the difficulties. This is, in effect, to turn the problem inside out and instead of trying to specify functions and expressions which have no side effect to specify objects which are immune from any possible side effect of others.

- hiding
- freezing

Section 3.4.6 - Constants and variables

The characteristic thing about variables is that their R-values can be altered by an assignment command. If we are looking for an object which is frozen, or invariant, an obvious possibility is to forbid assignments to it. This makes it what in CPL we call a constant.

Constancy is thus an attribute of an L-value, and is, moreover, an invariant attribute.

Section 3.4.7 - Fixed and free (functions)

Functions which call their free variables by reference (L-value) are liable to alteration by assignments to their free variables. This can occur either inside or outside the function body, and indeed, even if the function itself is a constant. Furthermore they cease to have a meaning if they are removed from an environment in which their free variables exist. (In ALGOL this would be outside the block in which their free variables were declared.) Such functions are called free functions.

Section 3.4.7 - Fixed and free (functions)

The converse of a free function is a fixed function. This is defined as a function which either has no free variables, or if it has, whose free variables are all both constant and fixed. The crucial feature of a fixed function is that it is independent of its environment and is always the same function. It can therefore be taken out of the computer (e.g., by being compiled separately) and reinserted again without altering its effect.

Section 3.4.8 - Segmentation

- Fixed functions can be compiled separately
- So can fixed functions that generate free functions at run-time

Section 3.5 - Functions and routines as data items

- Introduction of first-class objects distinction

Section 3.5.1 - First and second class objects

```
(x > 1 ? a : b) + 6

(x > 1 ? sin : cos)(6)
```

Section 3.5.1 - First and second class objects

Historically this second class status of procedures in ALGOL is probably a consequence of the view of functions taken by many mathematicians: that they are constants whose name one can always recognise.

Section 3.5.2 - Representation of functions

Thus the R-value of a function contains two parts — a rule for evaluating the expression, and an environment which supplies its free variables. An R-value of this sort will be called a closure.

The most straightforward way of representing the environment part is by a pointer to a Free Variable List (FVL).

Section 3.6 - Types and polymorphism

- Basic introduction of types and how functions use types

Section 3.6.1 - Types

A possible starting point is the remark in the CPL Working Papers [3] that “The Type of an object determines its representation and constrains the range of abstract object it may be used to represent. Both the representation and the range may be implementation dependent”. This is true, but not particularly helpful. In fact the two factors mentioned — representation and range — have very different effects. The most important feature of a representation is the space it occupies and it is perfectly possible to ignore types completely as far as representation and storage is concerned if all types occupy the same size of storage. This is in fact the position of most assembly languages and machine code — the only differences of type encountered are those of storage size.

Section 3.6.1 - Types

We call ambiguous operators of this sort polymorphic as they have several forms depending on their arguments.

Section 3.6.2 - Manifest and latent

It is natural to ask whether type is an attribute of an L-value or of an R-value — of a location or of its content.

Section 3.6.2 - Manifest and latent

In CPL the type is a property of an expression and hence an attribute of both its L-value and its R-value. Moreover L-values are invariant under assignment and this invariance includes their type. This means that the type of any particular written expression is determined solely by its position in the program. This in turn determines from their scopes which definitions govern the variables of the expression, and hence give their types. An additional rule states that the type of the result of a polymorphic operator must be determinable from a knowledge of the types of its operands without knowing their values. Thus we must be able to find the type of a + b without knowing the value of either a or b provided only that we know both their types.

Section 3.6.2 - Manifest and latent

The result of these rules is that the type of every expression can be determined at compile time so that the appropriate code can be produced both for performing the operations and for storing the results.

Section 3.6.2 - Manifest and latent

We call attributes which can be determined at compile time in this way manifest; attributes that can only be determined by running the program are known as latent.

Section 3.6.3 - Dynamic type determination

The opposite extreme is also worth examining. We now decide that types are to be attributes of R-values only and that any type of R-value may be assigned to any L-value.

Assembly languages and other 'simple' languages merely forbid polymorphism. An alternative, which has interesting features, is to carry around with each R-value an indication of its type. Polymorphic operators will then be able to test this dynamically (either by hardware or program) and choose the appropriate version.

Section 3.6.4 - Polymorphism

The difficulties of dealing with polymorphic operators are not removed by treating types dynamically (i.e., making them latent).

Section 3.6.4 - Polymorphism

In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments.

Section 3.6.4 - Polymorphism

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` *is a function whose argument is of type* `A` *and whose results is of* `B` *(so that the type of* `f` *might be written* `A -> b`*), and that* `L` *is a list whose elements are all of type* `A` *(so that the type of* `L` *is* $List\{A\}$*).*

Section 3.6.5 - Types of functions

The type of a function includes both the types and modes of calling of its parameters and the types of its results.

Section 3.6.5 - Types of functions

Some programming languages allow functions with a variable number of arguments; those are particularly popular for input and output. They will be known as variadic functions, and can be regarded as an extreme form of polymorphic function.

Section 3.6.5 - Types of functions

Although this is not impossible, it causes a considerable increase in the complexity of the compiler and exerts a strong pressure either to forbid programmers to define new polymorphic functions or even to reduce all polymorphic functions to second class status.

Section 3.7 - Compound data structures

Section 3.7.1 - List processing

It was not until mathematicians began using computers for non-numerical purposes — initially in problems connected with artificial intelligence — that any general forms of compound data structure for programming languages began to be discussed.

In LISP, for instance, there are only two sorts of object, an atom and a cons-word which is a doublet.

Section 3.7.2 - Nodes and elements

A node may be defined to consist of one or more components; both the number and the type of each component is fixed by the definition of the node. A component may be of any basic or programmer-defined type (such as a node), or may be an element. This represents a data object of one of a limited number of types; the actual type of object being represented is determined dynamically.

Section 3.7.3 - Assignments

In order to understand assignments in compound data structure we need to know what are the L- and R-values of nodes and their components.

Section 3.7.4 - Implementation

- Describes pointer bit hacks

Section 3.7.5 - Programming example

- Some examples of how type concepts might show up in practice

Section 3.7.6 - Pointers

There is no reason why an R-value should not represent (or be) a location; such objects are known as pointers.

- Follow
- Pointer

Section 3.7.7 - Other forms of structure

- List
- Ntuple
- Set
- Bag or Coll

Section 4 - Miscellaneous topics

Section 4.1 - Load-Update Pairs

- Formal exposition of what behaviors L-values implement

Section 4.2 - Macrogenerators

A much more conventional view is that a program is a symbol string (with the strong implication that it is nothing more), a programming language the set of rules for writing down local strings, and mathematics in general a set of rules for manipulating strings.

Section 4.2 - Macrogenerators

The outcome of this attitude is a macrogenerator whose function is to manipulate or generate symbol strings in programming languages without any regard to their semantic content.

Section 4.3 - Formal semantics

- Skipping