MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 353 March 10, 1976

LAMBDA

THE ULTIMATE IMPERATIVE

by

Guy Lewis Steele Jr. and Gerald Jay Sussman

Abstract:

We demonstrate how to model the following common programming constructs in terms of an applicative order language similar to LISP:

Simple Recursion
Iteration
Compound Statements and Expressions
. . .

{{Transcription of AIM-353 by Roger Turner; links added, including cite backlinks in Notes and Bibliography.}}

Contents

People who like this sort of thing will find this is the sort of thing they like.
– Abraham Lincoln

Introduction

We catalogue a number of common programming constructs. For each construct we examine "typical" usage in well-known programming languages, and then capture the essence of the semantics of the construct in terms of a common meta-language.

The lambda calculus {Note Alonzowins} is often used as such a meta-language. Lambda calculus offers clean semantics, but it is clumsy because it was designed to be a *minimal* language rather than a *convenient* one. All lambda calculus "functions" must take exactly one "argument"; the only "data type" is lambda expressions; and the only "primitive operation" is variable substitution. While its utter simplicity makes lambda calculus ideal for logicians, it is too primitive for use by programmers. The meta-language we use is a programming language called SCHEME {Note Schemepaper} which is based on lambda calculus.

SCHEME is a dialect of LISP. [McCarthy 62] It is an expression-oriented, applicative order, interpreter-based language which allows one to manipulate programs as data.

. . .

1.1. Simple Recursion

One of the easiest ways to produce a looping control structure is to use a recursive function, one which calls itself to perform a subcomputation. For example, the familiar factorial function may be written recursively in ALGOL:

integer procedure *fact*(n); **value** n; **integer** n;
  fact := **if** n=0 **then** 1 **else** n**fact*(n-1);

The invocation *fact*(n) computes the product of the integers from 1 to n using the identity $n! = n(n\text{-}1)!$ (n>0). If n is zero, 1 is returned; otherwise *fact* calls itself recursively to compute (n-1)!, then multiplies the result by n and returns it.

This same function may be written in SCHEME as follows:

```
(DEFINE FACT
   (LAMBDA (N) (IF (= N 0) 1
                   (* N (FACT (- N 1))))))
```

SCHEME does not require an assignment to the "variable" *fact* to return a value as ALGOL does. The `IF` primitive is the ALGOL **if-then-else** rendered in LISP syntax. Note that the arithmetic primitives are prefix operators in SCHEME.

. . .

2.2. The GO TO Statement

A more general imperative structure is the compound statement with labels and GO TOs. Consider the following code fragment due to Jacopini, taken from Knuth: [Knuth 74]

. . .

4.3. Fast Call-By-Name

Call-by-need does not fully capture the essence of call-by-name. If a side effect occurs between two references of a parameter, the parameter will yield the same value if passed call-by-need, but may yield different values if passed call-by-name. {Note Jensensdevice} For example:

```
begin
   real dx;
   real procedure integral(lower, upper, exp, var)
      value lower, upper;
      real lower, upper, exp, var;
      begin
         real sum;
         sum := 0;
         for var := lower + (dx/2) step dx until upper do
               sum := sum + exp;
         integral := sum;
      end;
   dx := .001;
   print(4 * integral(0, 1, dx/(x^2 + 1), x));
end
```

prints an approximation to *pi* by calculating

$$4\int_0^1 \frac{dx}{x^2+1}$$

which is four times the arctangent of 1. It depends on the call-by-name parameter *exp* changing value when the variable *var* is changed. This example in fact brings out *two* problems. First, call-by-need does not allow the value of a parameter to change when a variable used in the argument expression is modified. Second, the example presses the issue of assignment to call-by-name parameters.

The first problem can be fixed by modifying the call-by-need mechanism to notice side effects and re-evaluate the parameter if its value might have changed. Instead of `NEED-THUNK`, we use the following function:

```
(DEFINE MEMO-THUNK
        (LAMBDA (THUNK)
            ((LAMBDA (VALUE SAVED-COUNT)
                 (LAMBDA ()
                     (IF (= SAVED-COUNT (GLOBAL-SIDE-EFFECT-COUNT))
                         VALUE
                         (BLOCK (ASET 'SAVED-COUNT
                                      (GLOBAL-SIDE-EFFECT-COUNT))
                                (ASET 'VALUE (THUNK))
                                VALUE))))
             NIL
             -1))))
```

Conclusions

We have expressed a number of programming constructs in terms of a simple applicative language, SCHEME, based on lambda calculus. It is not surprising that this is possible, since SCHEME is universal. What is surprising is that the translation is so ***natural***. Most of the translations are syntactically local. The translated program is recognizably equivalent to the original, because the global structure is preserved. The translation process does not increase the size of the program very much.

Landin [Landin 65] and Reynolds [Reynolds 72] have used similar techniques to model programming constructs.

. . .

{Note Gotophobia}

. . .

Notes

{Alonzowins} ^

The lambda calculus was originally developed by Alonzo Church as a formal axiomatic system of logic. [Church 41] Happily, it may be re-interpreted in several interesting ways as a model for computation.

. . .

{Gotophobia} ^

The great GO TO controversy was started by Dijkstra in 1968 . . .
Knuth presents an extensive history of the GO TO controversy [Knuth 74]

. . .

{Jensensdevice} ^

The technique of repeatedly modifying a variable passed call-by-name in order to produce side effects on another call-by-name parameter is commonly known as Jensen's device, particularly in the case where the call-by-name parameters are j and $a[j]$. We cannot find any reference to Jensen or who he was, and offer a reward for any information leading to the identification, arrest, and conviction of said Jensen.
{{see [Naur 60] :-}}

. . .

{Landinknewthis} ^

In [Landin 65] Landin uses this same technique to model call-by-name. However, he modelled assignment to call-by-name parameters in a way much different from the one we use later: he uses L-values rather than an extra assignment thunk.

. . .

{Schemepaper} ^

SCHEME is fully described in [Sussman 75], which contains a complete reference manual as well as a fully documented implementation of the language in MacLISP [Moon 74].

Bibliography

. . .

{{ [Knuth 74] **1, 2**
Knuth, Donald E. *Structured Programming with go to Statements*. ACM Computing Surveys 6, 4 (December 1974) pp. 261–301. }}

[Landin 65] **1, 2, 3, 4, 5, 6, 7**
Landin, Peter J. *A Correspondence between ALGOL 60 and Church's Lambda-Notation*. CACM 8, 2-3 (February and March 1965).
. . .

[McCarthy 62] ^
McCarthy, John, et al. *LISP 1.5 Programmer's Manual*. The MIT Press (Cambridge, 1962).
{{see also McCarthy et al *LISP 1.5 Programmer's Manual* [Second edition] The MIT Press (Cambridge, 1965).}}
. . .

{{ [Naur 60] ^
Naur, Peter *An example of a function designator changing the value of a formal variable*. ALGOL Bulletin, Issue 10, pp. 12–13. }}
. . .

[Reynolds 72] **1, 2, 3, 4, 5, 6**
Reynolds, John C. *Definitional Interpreters for Higher Order Programming Languages*. ACM Conference Proceedings 1972.
. . .

[Sussman 75] **1, 2, 3, 4**
Sussman, Gerald Jay, and Steele, Guy L. Jr. *SCHEME: An Interpreter for Extended Lambda Calculus*. AI Lab Memo 349. MIT (Cambridge, December 1975).
{{Transcriptions: HTML, PDF}}
{{Republished with notes as *Scheme: A Interpreter for Extended Lambda Calculus*. Higher-Order and Symbolic Computation 11(4):405-439 (December 1998).
see also: Sussman, G.J., Steele, G.L. *The First Report on Scheme Revisited*. Higher-Order and Symbolic Computation 11, 399–404 (1998). }}
. . .