

## *The Next 700 Formal Language Descriptions*

LOCKWOOD MORRIS

lockwood@top.cis.syr.edu

The fact that most functions are constants defined by the programmer, and not variables . . . indicates a richness of the system which we do not know how to exploit very well.

*Lisp 1.5 Programmer's Manual, p. 18*

Most formal descriptions of programming languages are very like computer programs—I suspect not entirely because they are mostly written by ex-programmers. My impression is that several formal systems for language description attempt unsuccessfully to be completely unlike programming languages.

I do not wish to disparage the search for an escape from programming, but I want to suggest that while waiting for it to succeed we should apply some of our experience with programming languages to the problem of their formal description.

A program which is to actually run we want to be easy to write and easy to read (which to a first approximation is to say short), and so we write it in the most concise, or “powerful” if you like, language we can think of, subject to the constraint that there should be a system which runs programs in that language.

Similarly, we want a formal description to be easy to write and easy to read. The constraint is vaguer; roughly it is that we do not feel a formal description accomplishes much unless it is written in a language which was better understood beforehand than the language being described. What I want to suggest is that the art of formal description can build on its own successes, just as compilers for new programming languages need no longer always be written in machine code. Specifically, formal description has done best to date on applicative languages, or let me say the applicative language, variously known as “applicative expressions”, “pure Lisp”, and “applicative subset of . . .”. This happens to be a very concise language, and I propose we should use it to describe languages with non-applicative features.

The applicative language is especially suited to be a vehicle of formal description in that its domain of discourse includes functions. Now if we want to assign meanings to programs in a literal sense, we probably want meanings to be functions of some sort: e.g. a program for computing factorials should mean the factorial function. A formal description written in the applicative language can be so arranged that it operates on a program to yield its meaning, rather than operating on a program and its data to yield a final result. (Here, and for the rest of this paragraph, I am for simplicity assuming that formal descriptions simply *are* programs, although in fact, I have no quarrel with anyone who succeeds in reading them as non-programs.) That is to say, descriptions can be compiler-like rather than interpreter-like, in the sense of always producing a result, in a number of steps usually proportional to the size of their argument.

In what follows, I shall give first Landin's original mechanical evaluator for applicative expressions (which really is mechanical, and is only written in the applicative language by a sort of coincidence), and then three more descriptions of the applicative language, which are admittedly circular and useless, but which serve to show the progression from interpreter-like to compiler-like descriptions. I shall then give a compiler-like description of a (very schematized) language which might have assignments, and finally of one with jumps as well.

Some remarks on the notation used in what follows:  $h$ ,  $t$ ,  $:$ ,  $null$ ,  $nil$ ,  $*$ ,  $map$  are meant to stand for any set of functions which obey the laws for CAR, CDR, CONS, NULL, NIL, APPEND, MAPCAR in Lisp. The symbol  $\circ$  is used for functional composition, i.e.  $f \circ g(X) = f(g(X))$ . If  $M$ ,  $N$  are expressions, then as a more readable substitute for  $(\lambda x . M)(N)$  I may write:

$$x \text{ is } N; M$$

Similarly, instead of  $(\lambda x . M)(Y(\lambda x . N))$  I shall write

$$x \text{ isr } N; M$$

Y being the minimal-fixed-point finding function.

For making recursive definitions at the top level, however, I shall, out of habit, just write equations between functional expressions; e.g.

$$f(x, y) = \dots f \dots$$

instead of

$$f \text{ isr } \lambda(x, y) . \dots f \dots$$

I ask the reader's patience with the numerous undefined names which occur in the following formulae, and application of some sympathetic guesswork in piecing together the abstract syntaxes of the languages involved, and in inferring the types of the various objects. I have explained only what I consider the tricky bits, which is probably less than I should have.

Here, as an example of the old ways, is the SECD machine, adapted, via Wegner, from Landin's "Mechanical Evaluation of Expressions".

```

eval( $\pi$ ) = iterator(nil, primitive environment,  $\pi$ , nil)
iterator( $S, E, C, D$ ) = if null( $C$ )  $\wedge$  null( $D$ ) then h( $S$ )
                        else iterator(transform( $S, E, C, D$ ))
transform( $S, E, C, D$ ) =
  if null( $C$ ) then ( $S', E', C', D'$ ) is  $D$ ; (h( $S$ ) :  $S', E', C', D'$ )
  else if identifier(h  $C$ ) then (val( $E, h C$ ) :  $S, E, t C, D$ )
  else if  $\lambda$ exp(h  $C$ )
    then (constructclosure( $E, bv(h C), body(h C)$ ) :  $S, E, t C, D$ )
  else if h  $C$  = ap  $\wedge$  closure(h  $S$ )
    then (nil, (bvpert(h  $S$ ) : h( $t S$ )) : Epart(h  $S$ ), bodypart(h  $S$ ),
            (t( $t S$ ),  $E, t C, D$ ))
  else if h  $C$  = ap  $\wedge$   $\neg$ closure(h  $S$ ) then (h  $S$  (h( $t S$ )) : t( $t S$ ),  $E, t C, D$ )
  else if combination(h  $C$ )
    then ( $S, E, rand(h C)$  : rator(h  $C$ ) : ap : t  $C, D$ )
val( $E, i$ ) = if  $i$  = h( $E$ ) then t(h  $E$ ) else val(t  $E, i$ )

```

Here is a shorter equivalent.

```

eval'( $\pi$ ) = interpret( $\pi$ , primitive environment)
interpret( $\pi, E$ ) = if identifier( $\pi$ ) then val( $E, \pi$ )
                  else if  $\lambda$ exp( $\pi$ ) then  $\lambda a . interpret(body(\pi), (bv(\pi) : a) : E)$ 
                  else if combination( $\pi$ )
                    then interpret(rator( $\pi$ ),  $E$ )(interpret(rand( $\pi$ ),  $E$ ))

```

Three points before we go on:

1. *Transform* clarifies an important property of the variant of the applicative language I am using (if writing programs is what I am doing): application is done “by value”; that is, operands and operators are evaluated before application starts. The remarks I am going to make about the number of steps taken in “running” a description on a program would be nonsense if this were not understood.
2. In “Mechanical Evaluation of Expressions”, *transform* appears alone, without *eval* or *iterator*, and is said to be the transition function of a machine, so that even less of the power of the applicative language is being taken as already understood than appears here.
3. A function practically identical to *interpret* appears in “Mechanical Evaluation”, i.e., Landin does know how to program concisely in his own language.

Here is a version of the same thing which differs rather trivially from the preceding one, but which looks rather more like a compiler in that it makes it obvious that we only need to take the program to bits once.

```

eval''(\pi) = compile(\pi)(primitive environment)
compile(\pi) = if identifier(\pi) then \lambda E . val(E, \pi)
               else if \lambda exp(\pi) then f is compile(body(\pi));
                    \lambda E . \lambda a . f((bv(\pi) : a) : E)
               else if combination(\pi) then f is compile(rator(\pi));
                    g is compile(rand(\pi));
                    \lambda E . f(E)(g(E))

```

*Eval''* makes a questionable improvement over *eval'*, but it is a step towards doing something much more compiler-like: namely unzipping the environment into a list of identifiers—call it the static environment—present only at compile time, and a list of values—call it the dynamic environment—present only at run time. It will now become obvious that each occurrence of an identifier in the program only has to be looked up once in the symbol table.

```

eval'''(\pi) = compile'(\pi, primitive static environment)
                (primitive dynamic environment)
compile'(\pi, se) = if identifier(\pi) then finder(\pi, se)
                   else if \lambda exp(\pi) then f is compile'(body(\pi), bv(\pi) : se);
                        \lambda e . \lambda a . f(a : e)
                   else if combination(\pi) then f is compile'(rator(\pi), se);
                        g is compile'(rand(\pi), se);
                        \lambda e . f(e)(g(e))

finder(i, se) = if i = h se then h else finder(i, t se) \circ t

```

Now that we have a small bag of tricks, let us apply it to a somewhat more interesting language, say one with a state, or memory, in which the functions of the primitive environment may have side-effects. This is not really very hard; all we have to do is allow every evaluation which might entail invoking a primitive function to yield a new state as well as a value, and then make sure that states are passed around in such a way that they all descend in one hereditary line from the initial state.

```

execute(\pi) = (\lambda(x, \xi) . x)(compile(\pi, p.s.e)(p.d.e.)(initial state))
compile(\pi, se) = if identifier(\pi) then v is finder(\pi, se);
                  \lambda e . \lambda \xi . (v(e), \xi)
                  else if \lambda exp(\pi) then f is compile(body(\pi), bv(\pi) : se);
                        \lambda e . \lambda \xi . (\lambda(a, \eta) . f(a : e)(\eta), \xi)
                  else if combination(\pi) then f is compile(rator(\pi), se);
                        g is compile(rand(\pi), se);
                        \lambda e . \lambda \xi . ((\phi, \eta) is f(e)(\xi); \phi(g(e)(\eta)))

```

Note that the above description says precisely nothing about what sort of objects states may be; in particular we have no idea whether the primitive environment contains anything resembling an assignment operator and a contents function. Surprisingly enough, however, we are in a position to make the distinction that Reynolds [*CACM*, May, 1970] has pointed out between PAL-like languages, which bind identifiers only to references, and Algol 68-like languages, which bind to values in general. Namely, *compil* above binds to values in general; what goes into an environment is always the same as what comes back out. But we could write *compil'*, in which the line about getting something out of a dynamic environment, instead of

$$\lambda e . \lambda \xi . (v(e), \xi)$$

reads

$$\lambda e . \lambda \xi . (v(e)(\xi), \xi)$$

which says that anything put into the environment must be a function dependent on the state for its value. Actually, this is not terribly like binding to references; it is more like binding to the load halves of load-update pairs.

We could even write a most peculiar *compil''*, for a language in which the state was liable to being changed by merely accessing a bound value. For this, we would write in the same place

$$\lambda e . \lambda \xi . v(e)(\xi) \quad (\text{or “}v\text{” by itself, if we wanted to be deceptive})$$

where the bound function is expected to yield both halves of a (value, state) pair.

If you have believed me this far, we now seem to know what to do about languages which might have operations similar to assignment. Labels seem to be the next thing to try and cope with; the next and final example of a formal description schema will be about a language which might be Reynolds's GEDANKEN if enough more were said about it.

The problem with labels is that the evaluation of any subexpression may result in going to one, in which case the computation which might have been planned on to complete the evaluation of the main expression will have to be forgotten about. This means that the function compiled for the subexpression should be passed as an argument something which says what more is waiting to be done, so that the subexpression can decide whether to do it or not. This something has been variously called a “generalized program counter” and—misleadingly, I think—a “dump”. I shall take the simplistic approach here of having it be a function which does exactly what is required; namely, given the value and state resulting from the subexpression as arguments (assuming of course the subexpression hasn't gone anywhere) will obey the entire remainder of the program, all the way up to the top level, and come out with its final result.

GEDANKEN-like languages will need some more complicated abstract syntax than the simple applicative expressions we have been getting along with so far. Trivial additions are constants (introduced not because they are really different from variables, but because there might be infinitely many of them, and we would

like to keep the primitive environment finite) and conditional expressions. More important are blocks, about whose syntax I will say a Landinesque sentence:

A *block* has a *reclnames*, which is a list of identifiers  
 (of the simultaneously recursive functions declared in it)  
 and a *reclxeps*, which is a list of  $\lambda$ exps  
 (which are to be bound to corresponding reclnames)  
 and a *body*, which is a blockbody  
 where a *blockbody* is an expression (denoting the final value of the block)  
 or a *compoundbody*, and has a *firststatement*, which is an expression  
 and a *remainder*, which is a blockbody  
 or a *labelledbody*, and has a *label*, which is an identifier  
 and a *body*, which is a blockbody.

A block also has a *labellist* (of the labels appearing in it), but we may as well compute this:

$$\begin{aligned} \text{labellist}(b) = & \text{if } \text{labelledbody}(b) \text{ then } \text{label}(b) : \text{labellist}(\text{body}(b)) \\ & \text{else if } \text{compoundbody}(b) \text{ then } \text{labellist}(\text{remainder}(b)) \\ & \text{else } \text{nil} \end{aligned}$$

To make the description which follows a little easier to understand, here is a key to some of the variables used in it, with the types of the objects to which they get bound:

variable	type of binding		
	informal name	symbolic name	symbolic expression
$x, y, z$	compilations		$E \times \Xi \times P \rightarrow V$
$e, ne$	environments	$E$	$V$ -list
$\xi$	states	$\Xi$	
$\rho$	dumps	$P$	$V \times \Xi \rightarrow V$
$a, p$	values	$V$	$F + L + \text{primitive values}$
$f$	function values	$F$	$V \times \Xi \times P \rightarrow V$
$l$	label values	$L$	$\Xi \rightarrow V$

```

obey(program) = meaning(program)
                (primitive dynamic environment, initial state,  $\lambda(a, \xi) . a$ )
meaning(program) = compile(program, primitive static environment)
compile( $\pi, se$ ) = if constant( $\pi$ ) then  $\lambda(e, \xi, \rho) . \rho(\text{denotation}(\pi), \xi)$ 
  else if identifier( $\pi$ ) then ( $q$  is finder( $\pi, se$ );  $\lambda(e, \xi, \rho) . \rho(q(e), \xi)$ )
  else if  $\lambda exp$ ( $\pi$ ) then ( $y$  is compile(body( $\pi$ ),  $bv(\pi) : se$ );
     $\lambda(e, \xi, \rho) . \rho(\lambda(a, \xi, \rho) . y(a : e, \xi, \rho), \xi)$ )
  else if combination( $\pi$ )
  then ( $y$  is compile(rator( $\pi$ ),  $se$ );
     $z$  is compile(rand( $\pi$ ),  $se$ );
     $\lambda(e, \xi, \rho) . y(e, \xi, \lambda(f, \xi) . z(e, \xi, \lambda(a, \xi) . f(a, \xi, \rho)))$ )
  else if conditional( $\pi$ )
  then ( $x$  is compile(premiss( $\pi$ ),  $se$ );
     $y$  is compile(conclusion( $\pi$ ),  $se$ );
     $z$  is compile(alternative( $\pi$ ),  $se$ );
     $\lambda(e, \xi, \rho) . x(e, \xi, \lambda(p, \xi) . \text{if } p \text{ then } y(e, \xi, \rho)$ 
      else  $z(e, \xi, \rho))$ )
  else if block( $\pi$ )
  then ( $nse$  is labellist(body( $\pi$ )) * renames( $\pi$ ) *  $se$ ;
     $ss$  is compilebody(body( $\pi$ ),  $nse$ );
     $rr$  is map(rec $\lambda exps$ ( $\pi$ ),  $\lambda \pi . \text{compile}(\pi, nse)$ );
     $\lambda(e, \xi, \rho) . (ne \text{ isr } \text{map}(t(ss), \lambda x . \lambda \xi . x(ne, \xi, \rho))$ 
      *  $\text{map}(rr, \lambda x . x(ne, \xi, \lambda(f, \xi) . f))$ 
      *  $e$ ;
     $h(ss)(ne, \xi, \rho))$ )

```

We still need to define compilebody, which goes through a blockbody and compiles for each label a “composition” (but done inside-out, using dumps) of all the statements between the label and the end of the block, and returns a list of all these, with the “composition” for the entire body stuck on at the front:

```

compilebody( $b, se$ ) =
  if labelledbody( $b$ ) then ( $xx$  is compilebody(body( $b$ ));
    ( $h \ xx$ ) :  $xx$ )
  else if compoundbody( $b$ )
  then ( $xx$  is compilebody(remainder( $b$ ));
     $y$  is  $h \ xx$ ;
     $z$  is compile(firststatement( $b$ ),  $se$ );
    ( $\lambda(e, \xi, \rho) . z(e, \xi, \lambda(a, \xi) . y(e, \xi, \rho))$ ) :  $t \ xx$ )
  else compile( $b, se$ ) :  $nil$ 

```

We should also say how label values are to be used, namely that something in the primitive static environment, say “**goto**”, should correspond in the primitive dynamic environment to

$$\lambda(l, \xi, \rho) . I(\xi)$$

University of Essex  
November 1970

### References

1. Landin, P. J. The mechanical evaluation of expressions. *Computer Journal*, 6 (January 1964) 308–320.
2. Landin, P. J. The next 700 programming languages. *Comm. ACM*, 9 (January 1966) 157–166.
3. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge (1962, reprinted February 1965).
4. Reynolds, J. C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM*, 13, 5 (May 1970) 308–319.
5. Wegner, P. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill (1968).