Higher-Order and Symbolic Computation, 11, 125–143 (1998)

A Generalization of Jumps and Labels

PETER J. LANDIN peterl@dcs.qmw.ac.uk
QMW, *University of London* [1]

Abstract. This paper describes a new language feature that is a hybrid of labels and procedures. It is closely related to jumping out of a functional subroutine, and includes conventional labels and jumping as a special, but probably not most useful, case. It is independent of assignment, i.e., it can be added to a "purely-functional" ("non-imperative") system (such as LISP without pseudo-functions or program feature). Experiments in purely functional programming suggest that its main use will be in success/failure situations, and failure actions. This innovation is incorporated in the projected experimental system, ISWIM.

Keywords:

> "Explaining to programmers the logical structure of programming languages is like a cat explaining to a fish what it feels like to be wet"—Gorn.

Introduction

This paper provides constructive answers to three questions about the logical structure of programming languages. Each arises from an observation about the roles of labels and jumping in current languages.

First, there are languages that have neither assignment nor jumping; e.g., "pure LISP", i.e., LISP without pseudo-functions or program feature. (For evidence that such a language may nevertheless be powerful, see [7], especially the Appendix, and [6].) Also, it is obvious that a language may have assignment but not jumping; e.g., ALGOL 60 deprived of **goto**, labels, **label**, and **switch**'s; or LISP with program feature but without labels and `GOTO`. Is the converse true? That is to say, can a language have no assignment and yet have jumping, or rather have some analogy to conventional jumping that yields conventional jumping as a special case when assignment is added?

There is an elementary remark that may not be out of place here. It is impossible to introduce assignment into pure LISP merely by a definition. The proof of this is that no mere definition can interfere with the interchangeability of "x" and "**cons**(**car**(x),**cdr**(x))". So there is a firm distinction between having and not having assignment.

Now for the second observation and the second question. Some operations can usefully be present in a language even though there are no atomic symbols that are appropriate operands for them. E.g., operations on truth-values, in particular ALGOL 60's conditional expressions, and '$\vee$', etc. would be useful even without Boolean identifiers, or the symbols '**true**' and '**false**'. Is this true of jumping? i.e., is there some operation that is a component of jumping but can meaningfully be used without any feature like user-coined labels; and that yields conventional jumping when user-coined labels are added?

Third, there is an analogy between labels and the identifiers of parameterless, non-functional procedures. In ALGOL 60 this analogy is so close that the symbol **goto** is

actually redundant; a processor could safely skip its occurrences. Can this analogy be reversed to provide things like labels, but bearing the same relation to "parameterful" and/or functional procedures that labels bear to parameterless, non-functional ones?

An affirmative answer to each of these questions is provided by describing a new language feature that is independent of assignment and of user-coined identifiers, and yields label-like analogies to all kinds of procedures. It also provides an answer to the question: What does a label denote? More precisely it conditionally gives meaning to phrases like "the value, or denotation of a label", and lays down limits to how far such talk can be pushed.

This new language feature generalizes jumps and labels as they occur in current languages. It seems possible that the familiar special case is not the most useful one.

Programmers are usually wary of assertions about programming languages in general, rather than about some particular language—justifiably since such assertions are usually vague. They are with equal justification suspicious when an assertion about a specific language is alleged to have wider linguistic significance. The present discussion avoids both these strictures. It is precise without being special to a single language. It achieves this by using the language model described in [3], one of whose purposes was to facilitate precise statements about languages in general.

Thus the illustrative references above to LISP and ALGOL 60 imply no special commitment to those languages. They are included as a gesture to readers unfamiliar with the AE/SECD model. Such a reader might well turn first to the first section of the Appendix, which describes the innovation informally, presenting it in the guise of an extension to ALGOL 60.

An extension to applicative expressions

An earlier paper [3] described a formal system whose only structural rules are functional application and functional abstraction, and showed how it mirrored many familiar features of current programming languages. The expressions of this system were called "applicative expressions" (AEs). In the present paper and a companion [10] dealing with assignment, the AE-system is developed so as to mirror *imperative* features of programming languages. For example, the extended system can be considered as a generalization of ALGOL 60. It was briefly described in a previous paper [4] which used it as a tool for formalizing ALGOL 60's semantics.

When describing AEs [3], their structure was given by means of a "structure definition", and their semantics by a definition of a function *val* that produces the "value of", or "thing denoted by" an AE. This was followed up by describing an abstract machine, the "SECD-machine" (Stack, Environment, Control, Dump), that interprets AEs "correctly", i.e., it produces the value of a given AE. The machine was itself characterized by the definition of another function *Transform*, its step-by-step transition rule. When we add jumping, we find by contrast that the meaning of an expression is difficult to characterize without resorting directly to the SECD-machine. So the revised semantics will be defined in terms of the behavior of the SECD-machine, developed to allow for the new facility.

We introduce a new kind of intermediate result, the "program-closure" that can be considered as the "value", or "thing denoted by", a label. A program-closure is characterized by

its *body*-part, which is a function,
and its *dump* part, which is an SECD-state.

We now explain how program-closures are produced and how they are used.

There is an operation **J** that transforms a function f into the program-closure whose body is f and whose dump is the current dump. Formally, this transformation is

$$\lambda(\mathbf{J}{:}f{:}S, E, ap{:}C, D).\\ \quad (consprogramclosure(f, D){:}S, E, C, D)$$

Each time a program-closure L comes to be applied, the first thing to happen is that it is replaced by L's body, and the current state (or more accurately that part of it which is about to be dumped if the function is a closure) is replaced by the SECD-state that constitutes L's dump. Then the attempt to apply function to argument is recommenced. Formally, this transformation is

$$\lambda(L{:}x{:}S, E, ap{:}C, D).\\ \quad (body\, L{:}x{:}S', E', ap{:}C', D')\\ \quad \textbf{where } (S', E', C', D') = dump\, L$$

Unlike the functions in the original SECD-system, **J** cannot be characterized in terms of its effect on the abstract objects that are "denoted by" AEs. It depends essentially on the SECD-mechanism. The proof of this is that it defies the standard substitution rules. For example, compare these two AEs.

$$\begin{array}{l}\textbf{let } f(x, y) =\\ \quad \textbf{let } E = \mathbf{J}\lambda(u, v, w).(u - v)/w\\ \quad \textbf{let } g(z) = \ldots E(a^2, b^2, c^2) \ldots\\ \quad \ldots g(h^2) \ldots\\ \ldots f(d^2, e) \ldots\end{array} \qquad \begin{array}{l}\textbf{let } f(x, y) =\\ \quad \textbf{let } g(z) =\\ \quad\quad \textbf{let } E = \mathbf{J}\lambda(u, v, w).(u - v)/w\\ \quad\quad \ldots E(a^2, b^2, c^2) \ldots\\ \quad \ldots g(h^2) \ldots\\ \ldots f(d^2, e) \ldots\end{array}$$

Here we assume that E is not called outside g's definition, and note that E's definition has no free variables. So, but for the **J**, their equivalence would follow by the standard substitution rules.

As a corollary this counterexample also shows that **J** is a genuinely new facility in the sense that it cannot be introduced into, say, pure LISP by a definition.

The **J** operator was first introduced in a slightly less general form in [4] where it played an important role in the logical analysis of ALGOL 60 (and also by implication of other languages containing both labels and procedure structure). It was briefly related to another innovation called there "program-point declarations". This relation is developed in the appendix.

The **J** operator was used in [6], where it played an important role in an experimental application of LISP-like programming methods to a conventional lengthy numerical calculation that included error tests and error actions. This use of **J** is discussed in the next section but one.

An extension to the where-notation

Applicative expressions are abstract objects in the sense of being characterized independently of specific written representations. For example,

$$(\lambda u.u(u+1))(a+b) \qquad \begin{array}{l} u(u+1) \\ \textbf{where } u = a+b \end{array}$$

are two ways of writing the same AE. The **where**-notation can be considered as a more palatable way of writing the expressions of λ-calculi.

The **where** notation can be extended to cater legibly for the more useful occurrences of **J**. We introduce the following additional piece of "syntactic sugar" for writing AEs (It is additional in the sense of being defined here in terms of other syntactic sugar, namely "definitions".):—

pp*definition* *definee* = **J (***definiens***)**

e.g.,

pp$\boldsymbol{L = \lambda x. \ldots}$ $L = \mathbf{J}\ \lambda x. \ldots$

Hence

let pp $L = \lambda x. \ldots$ $(\lambda L. \ldots)(\mathbf{J}\ \lambda x. \ldots)$

The symbol pp may be read "program-point".

Thus, corresponding to the sugared notation for $\boldsymbol{\lambda}$-expressions:

$$\boldsymbol{f = \lambda(u,v).uv(u+v)} \qquad \boldsymbol{f(u,v) = uv(u+v)}$$

we use a sugared notation for pps:

$$\boldsymbol{f = \mathbf{J}\ \lambda(u,v).uv(u+v)} \qquad \boldsymbol{pp\ f(u,v) = uv(u+v)}$$

The pp-notation suffices for all occurrences of **J** as an operator. For,

$$\boldsymbol{\ldots(\mathbf{J}\ F)\ldots}$$

is equivalent to

$$\boldsymbol{(\lambda L.\ \ldots L \ldots)(\mathbf{J}\ F)}$$

i.e., to

$$\begin{array}{l} \textbf{let pp } \boldsymbol{L = F} \\ \boldsymbol{\ldots L \ldots} \end{array}$$

It is worth stressing that we are not concerned here with this or any other specific notation for program-points, but with the notion of program-points *abstracted from* written representations. That is to say we are presenting for consideration, a new feature for program languages, but we are doing so in such a way that it can be discussed and judged without regard to a particular way of writing it.

We are also suggesting particular ways of writing it, partly to lend concreteness to our proposal, but mainly to facilitate the presentation. Someone assessing the proposals might well accept the abstract feature while rejecting our choice of concrete representations, just as he might object to the term "program-point" without objecting to the concept. In either case, he might go on to seek more acceptable alternatives This is the virtue of the technique of "abstract structure" as an approach to language design.

Implementation

It is intended that the currently projected ISWIM system should include **J**, dressed up in the guise of 'pp'. However, in the absence of any experience the present section is no more than a note on the general implications that the above specification of **J** has for the run-time set-up.

In the appendix we show how program-points can be eliminated in terms of jumps and labels, in particular in terms of jumps out of procedures. This correctly suggests that the implementation of **J** involves a similar mechanism to that required for jumping out of an ALGOL 60 procedure.

This is not compounded by the greater generality of AEs over ALGOL 60. The function-producing feature of AEs amounts roughly to the possibility of calling a procedure after leaving the block in which it was declared. This makes a LIFO storage allocation scheme inadequate unless supported by another non-LIFO source of storage together with a rule for choosing which to use. The situation for pps is the same as for functions. For example, as a special case, given an entirely non-LIFO system, such as that used by LISP, **J** presents no storage allocation problems.

Such rules are outside the present topic. We may however, observe that, when **J** is added, the function-producing feature amounts roughly to the possibility of jumping to a label after leaving the block in which it was "introduced". This situation is similar to that arising when label assignments or procedure assignments are introduced into FORTRAN or ALGOL 60. There has recently been a lot of discussion of such facilities. In PL/1 and EULER it is possible to write a program that may lead to such apparent defiance of scopes, but no meaning is attached to it.

The system of AEs leads naturally to a meaning for such a program, and experiments suggest that this is an extremely useful programming tool. Among other things, it provides a rather elegant **own**-like facility (see [5]).

Use

My experiments in non-procedural programming (partially reported in [6,7]) suggest that its weak point lies in the success/failure situations and the diagnostic or remedial action required on failure. It seems to be in this area that labels and jumping can be adequately matched only by using program-points. It also seems to be this area that gains most from the greater generality that program-points offer over conventional labels and jumping; and that suffers least from their deficiency in being unapproachable in-line.

Thus in the NPL program (a lengthy thermodynamics calculation) that serves as a text for [6], my transliteration into functional notation used functions whose arguments include

failure actions. The question whether a failure action is a function or a pp does not affect the textual appearance of calls for it. Moreover, the transliterated program contains no occurrence of **J** (nor of a pp-definition), since the failure actions are a parameter of the entire NPL program.

Nevertheless, it is crucial to its correct operation that they are pps and not functions. (For the benefit of a reader who refers to [6] it should be observed that the failure situations are classified there into "hard errors" and "soft errors". Only the hard errors prematurely terminate the calculation that detects them; i.e., the soft errors do not involve **J**.)

The original NPL program allows for diagnostic action by virtue of the many "external" variables that would be accessible to any failure routine that might be provided. It is to be presumed that these are enough for any required discrimination among possible causes of failure, or reporting of discredited results. One major feature of non-imperative programming is the lack of any facility like resetting external variables. The entire outcome of executing any routine is contained in its "result". This statement holds even if the execution discredits itself.

Thus, the use of pps with parameters and results (rather than merely the parameterless, resultless sort) is vital if the new technique is to hold its own. On a subjective rating it does better than that. The arguments for and against functional as opposed to procedural programming are equally applicable in its support. Notably, the question whether a variable contributes to the diagnostic or remedial action is no longer indicated by the sequence of executing various steps, but by the program's parenthetical structure (more accurately, by what we call its "applicative structure").

I now give a quite different application, to programs for scanning a text to recognize phrase structure

Let us call a "recognizer" a function that operates on two arguments.

1. a failure action, which may be either a function or a program closure,

2. a source text

and produces, if successful, two things:

1. the result of successful recognition

2. a residual tail of the source text, derived from it by removing the recognized stem.

If unsuccessful, then the recognizer produces whatever results from applying the failure action to the source text.

If $\boldsymbol{f}$ and $\boldsymbol{g}$ are two such recognizers, then we are interested in certain related recognizers, corresponding to cartesian concatenation and to class union (these being the two operations that are basic to BNF, and are indicated there by juxtaposition and '|' respectively).

Each of the following functions operates on two recognizers and produces a recognizer:

$$
\begin{aligned}
concat'\ (\boldsymbol{f},\boldsymbol{g})(\boldsymbol{E},\boldsymbol{s}) = {} & \textbf{let } \boldsymbol{E}' = \mathbf{J}\ \boldsymbol{E} \\
& \textbf{let } \boldsymbol{b}', \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{E}', \boldsymbol{s}) \\
& \textbf{let } \boldsymbol{b}'', \boldsymbol{s}'' = \boldsymbol{g}(\boldsymbol{E}'', \boldsymbol{s}') \\
& \qquad \textbf{where } \boldsymbol{E}''(\boldsymbol{s}') = \boldsymbol{E}'(\boldsymbol{s}) \\
& construct(\boldsymbol{b}', \boldsymbol{b}''), \boldsymbol{s}''
\end{aligned}
$$

$$union'\ (\boldsymbol{f},\boldsymbol{g})(\boldsymbol{E},\boldsymbol{s}) = \boldsymbol{f}(\boldsymbol{E}',\boldsymbol{s})$$
$$\textbf{where } \boldsymbol{E}'(\boldsymbol{s}) = \boldsymbol{g}(\boldsymbol{E},\boldsymbol{s})$$

The former definition uses a function *construct* for combining the results of two successful recognitions.

The technique used here is in contrast with both the following:

1. The conventional procedural method using exit jumps
2. The non-procedural method using functions that produce a success/failure indication as part of their result (see e.g. Burge [1]).

The transition-rule of the extended SECD-machine

We now present the step-by-step transition-rule for the SECD-machine extended to allow for **J**. This is to be compared with the earlier definition of *Transform* given in [3]. The changes are precisely the additional arms for the cases "*progclosure* $\boldsymbol{f}$" and "$\boldsymbol{f}$ = **J**".

$$\begin{array}{l}
Transform\ [\boldsymbol{S},\boldsymbol{E},\boldsymbol{C},\boldsymbol{D}] = \\
\quad null\ \boldsymbol{C} \rightarrow [\boldsymbol{h}\ \boldsymbol{S}{:}\boldsymbol{S}',\boldsymbol{E}',\boldsymbol{C}',\boldsymbol{D}'] \\
\qquad\qquad \textbf{where } [\boldsymbol{S}',\boldsymbol{E}',\boldsymbol{C}',\boldsymbol{D}'] = \boldsymbol{D} \\
\quad \textbf{else} \rightarrow \\
\quad \textbf{let } \boldsymbol{X} = \boldsymbol{h}\ \boldsymbol{C} \\
\quad identifier\ \boldsymbol{X} \rightarrow [(location\ \boldsymbol{E}\ \boldsymbol{X})\boldsymbol{E} : \boldsymbol{S},\boldsymbol{E},\boldsymbol{t}\ \boldsymbol{C},\boldsymbol{D}] \\
\quad \boldsymbol{\lambda} exp\ \boldsymbol{X} \rightarrow [consclosure((\boldsymbol{E}, bv\ \boldsymbol{X}), u(body\ \boldsymbol{X})) : \boldsymbol{S},\boldsymbol{E},\boldsymbol{t}\ \boldsymbol{C},\boldsymbol{D}] \\
\quad \boldsymbol{X} = ap \rightarrow \\
\qquad \textbf{let } \boldsymbol{f}{:}\boldsymbol{x}{:}\boldsymbol{S}' = \boldsymbol{S} \\
\qquad closure\ \boldsymbol{f} \rightarrow \\
\qquad\quad \textbf{let } consclosure((\boldsymbol{E}',\boldsymbol{J}),\boldsymbol{C}') = \boldsymbol{f} \\
\qquad\quad [(), consenv(assoc(\boldsymbol{J},\boldsymbol{x}),\boldsymbol{E}'),\boldsymbol{C}',[\boldsymbol{S}',\boldsymbol{E},\boldsymbol{t}\ \boldsymbol{C},\boldsymbol{D}]] \\
\qquad progclosure\ \boldsymbol{f} \rightarrow \\
\qquad\quad \textbf{let } consprogclosure(\boldsymbol{f}',\boldsymbol{D}') = \boldsymbol{f} \\
\qquad\quad [\boldsymbol{f}'{:}\boldsymbol{x}{:}\boldsymbol{S}'',\boldsymbol{E}'',ap{:}\boldsymbol{C}'',\boldsymbol{D}''] \\
\qquad\quad \textbf{where } \boldsymbol{S}'',\boldsymbol{E}'',\boldsymbol{C}'',\boldsymbol{D}'' = \boldsymbol{D}' \\
\qquad \textbf{else} \rightarrow [\{\boldsymbol{f} = \mathbf{J} \rightarrow consprogclosure(\boldsymbol{x},\boldsymbol{D});\ \boldsymbol{f}\ \boldsymbol{x}\} : \boldsymbol{S},\boldsymbol{E},\boldsymbol{t}\ \boldsymbol{C},\boldsymbol{D}] \\
\quad \textbf{else} \rightarrow \textbf{let } combine(\boldsymbol{F},\boldsymbol{Z}) = \boldsymbol{X} \\
\qquad\quad [\boldsymbol{S},\boldsymbol{E},\boldsymbol{Z}{:}\boldsymbol{F}{:}ap{:}\boldsymbol{t}\ \boldsymbol{C},\boldsymbol{D}]
\end{array}$$

Conclusion

The introduction of imperative features into a "purely functional" (or "denotative", or "referentially transparent") system, has precedents, namely the "program feature" of LISP, and an analogous feature of Gilmore's abstract machine [2]. However, in these schemes, labels differed from other identifiers in that no meaning was attached to any but the simplest

use of them, namely as the destination of jumps at scope level zero. This is a special case of what ALGOL 60 allows. The IAE system includes identifiers that play the role of labels, without structural restrictions on their use. For example, a meaning is given to functions whose result is denoted by a label, and to label assignments. The labels of ALGOL 60, and a fortiori of LISP and Gilmore are included as a special case.

We have separated certain features of programming languages that usually appear interdependent. We have shown that jumping and labels are not essentially connected with strings of imperatives and in particular, with assignment. Second, that jumping is not essentially connected with labels. In performing this piece of logical analysis we have provided a precisely limited sense in which the "value of a label" has meaning. Also, we have discovered a new language feature, not present in current programming languages, that promises to clarify and simplify a notoriously untidy area of programming—that concerned with success/failure situations, and the actions needed on failure.

Thus, IAEs offer a blending of labels into functional notation, in which all the machinery of functional notation plays a natural part.

The result is a generalization of jumping, of which the familiar special case may prove to be the least important for working programmers.

Appendix on the relation between labels and J

The following sections are a detailed discussion of the relationship between **J** and conventional jumping as manifest in ALGOL 60. We first informally describe a possible extension to ALGOL 60, namely "program-point declarations". Then we relate these on the one hand to existing ALGOL 60 facilities and on the other hand to **J**.

(This appendix is a more detailed and slightly amended version of the subsection on "Labels and Jumps" in [5]. That in turn added details to a similarly entitled subsection of [4].)

An analysis of ALGOL 60's labels

The following discussion of jumps springs from the observation that the symbol '**goto**' in ALGOL 60 is redundant, and could be ignored by a processor. That is to say, there is a considerable similarity between labels and the identifiers of parameterless non-type procedures. It is possible to use the same "calling mechanism" for both, leaving any

differences to be made by the thing that is "called". Thus there is a natural meaning to be given to a program that, at different times, substitutes labels and procedures for the same formal, e.g.

procedure P;
 if q **then goto** M;
 . . .

 L: . . .
 . . . $f(P)$ . . . $f(L)$ . . .

In view of this feature of ALGOL 60, it is to be expected that some people have sought to "explain" labels as a sort of heterodoxly written declaration. That is to say it has been shown how the work of labels can be performed instead by identifiers introduced in block-heads, using either conventional procedure declarations or declarations of a new kind. Both approaches bring out the similarity between labels and the identifiers of parameterless non-type procedure.

The former approach is developed by van Wijngaarden [9], and requires that

- Exit from a procedure is always by **goto**; i.e natural exiting is eliminated. That is to say the device only yields a valid treatment of procedure exits at the cost of abandoning the facility for closed subroutines that is embodied in ALGOL 60's procedure. This leads to a blurring of the distinction between exiting and temporarily delegating control to another procedure. It thus fails to reflect the possibility of information disposal, i.e., of storage recovery.
  If the purpose of the analysis is semantic specification, not cheap running, then this device is not invalidated by the fact that it involves accumulating a pile of "resumption-points", one for every executed jump, that are never taken up. However, it is less satisfactory if we are hoping for a model that illuminates run-time behavior as well as language.

- Type-procedures are replaced by non-type ones. This involves eliminating nested calls by decomposing them into strings of statements. It therefore presents something of a challenge to anyone who is interested in using fewer, larger statements, rather than more and smaller.

To appreciate how these complications arise, consider the following:

```
begin real x;
        S1;
        S2;
    L1: S3;
        S4;
L2: L3: S5;
        S6;
    L4: S7;
        S8
end; L5 : ...
```

```
begin real x;
  procedure L1;
    begin S3; S4; L2 end;
  procedure L2; L3;
  procedure L3;
    begin S5; S6; L4 end;
  procedure L4;
    begin S7; S8; L5 end;
  S1;
  S2;
  L1
end; L5 : ...
```

where L_1, L_2 and L_3 occur in the S_i's (as well as explicitly in the right-hand version).

The intuitive difference between calling a procedure and jumping to a label is that the former makes provision for resumption of the sequence containing the call. So the execution of the right-hand version will accumulate one such potential resumption-point for each jump to L_i in the left-hand version. However, its effect on all local and global quantities will be the same as that of the left-hand version.

It is essential to this example that the block under treatment is a statement of a super-block, and not, for example, a procedure body. For only then can procedure L_4 have an unnatural exit, i.e., an exit by '**goto**' rather than by natural continuation. (The presence of the label L_5 is not a serious restriction. If absent, a label could be concocted.) If the block in question were a procedure body, the final segment would be exited naturally, and would be followed incorrectly by a resumption of the sequence that textually follows the call for L_4.

There are two ways out of this difficulty. One is to complicate the transformation of the original ALGOL 60 by inserting ALGOL 60 statements that make entry and exit explicit, i.e., abandoning the facility for closed sub-routines that is provided by ALGOL's procedures. This is the approach of van Wijngaarden referred to above. Of the two objections to it, the second emerges when the analysis is extended to deal with *functional* (i.e., "type") procedures.

Accordingly, we are prompted to compromise in the matter of logical economy, and introduce the new format called "program-point declarations" as a substitute for labelled segments of program. The execution of a program-point is the same as for the corresponding procedure, *except that* in the event of natural exit, not only is the program-point body exited, but also the block in whose head it is declared.

This section has given an informal and incomplete account of program-points by describing the original intention behind the concept. Their detailed relation to ALGOL 60's labels will be taken up in the section after next. The next section elaborates the concept as an *extension* to ALGOL 60 rather than as an alternative to labels. In this form it gives the readers familiar with ALGOL 60 an introduction to **J**.

An extension to ALGOL 60

The preceding section discussed the analogy between labels and parameterless non-type procedures. Our present point of departure is the idea of developing this analogy in reverse, i.e., we seek things like labels that are analogous to type-procedures with parameters. Accordingly, consider adding to ALGOL 60 a new kind of declaration, "program-point declarations", syntactically identical to procedure declarations except that the initial symbol '**programpoint**' replaces '**procedure**'. Thus the usual variants are allowed, i.e., with/without parameters, type/non-type. Furthermore, program-point identifiers may be used according to the same rules as procedure identifiers, giving rise to "program-point statements", "program-point designators", and to parameters of type **programpoint**.

We abbreviate the symbol **programpoint** to pp', to avoid prejudging its relation to the symbol pp introduced previously as a piece of syntactic sugar for certain occurrences of '**J**'. This relation is defined precisely in a later section, but meanwhile the imprecision will be emphasized by using 'pp'-declaration', 'pp'-statement', etc.

Consider for example

```
begin real procedure f(x, y);
      begin real pp' Error (u, v, w);
                  ...

            ...
            ... Error(a², b², c²) ...
            ...
      end
      ...
      ... f(d², e²) ...
      ...
end
```

Executing the call for f may or may not give rise to executing the call for *Error*. If it does then it will be prematurely terminated on exit from *Error*, and the result of *Error* will be delivered as the result of f. This holds regardless of the context of the call for *Error*, e.g. even if it is within inner blocks and procedure declarations.

On exit from an ALGOL 60 procedure, control is returned to a point determined by where the call for the procedure is written. By contrast, on exit from a pp' control is returned to a point determined by where the *declaration* of the pp' is written. To emphasize this fact compare the following (which are closely analogous to a pair of AEs given in the earlier section "An Extension to Applicative Expressions"):

```
begin
  real proc f(x, y);
  begin
    real pp' E(u, v, w);
      E = (u − v)/w;
    real proc g(z);
      ...
      ... E(a², b², c²) ...
      ...
    ...
    ... g(h²) ...
    ...
  end
  ...
  ... f(d², e²) ...
  ...
end
```

```
begin
  real proc f(x, y);
    begin
    real proc g(z);
      begin
        real pp' E(u, v, w);
        E = (u − v)/w;
        ...
        ... E(a², b², c²) ...
        ...
    ...
    ... g(h²) ...
    ...
  end
  ...
  ... f(d², e²) ...
  ...
end
```

Assume E is not called outside g's declaration, and note that E's declaration uses no non-locals. Had E been a procedure these two properties would have been enough to ensure the equivalence of the above pair. E could have been written either local to g, or less local, without any effect on the outcome. But because it is a pp', they are not equivalent. The left-hand case is merely the previous example with more detail filled in. A call for E prematurely terminates f (and a fortiori g). In the right hand case, the position of E's declaration makes it an emergency exit from g only, not from f.

More generally, compare the effect of executing a call for a pp' (either a pp'-statement or a pp'-designator) with that of the corresponding procedure call. They are identical up until the exit. Thus except for certain implicit suggestions about what happens at exit, section 4.7 of ALGOL 60 report, on "procedure statements" carries over intact. If the exit is "unnatural" (i.e., by a **goto**-statement, or by a call for a program-point non-local to the program-point being executed), then the fact of having called a **programpoint** rather than a **procedure** is totally without outcome.

If on the other hand the exit is "natural" (i.e., by running off the end of the body) then this precipitates exit (of the unnatural kind) from the block in whose head the pp' was declared; and a fortiori it also precipitates unnatural exit from every block or procedure entered since. Thus, in particular, if a pp' is declared in the head of a procedure-body, then any call for it precipitates an unnatural exit from the procedure, and so a natural exit from the procedure itself. If the procedure happens to be a *type*-procedure, then its result is the result of the pp' (which is therefore constrained to be a type-pp' of compatible type).

The above account is incomplete. It leaves unsettled various points concerning procedure bodies that are not blocks, pp's declared within **for**-statements, nested activations of procedure, and others. Our purpose here is not to specify precisely an addition to ALGOL 60, but to trade on the readers' familiarity with ALGOL 60 as a means of explaining program-points. In the following sections, the connection between labels, program-points, and AEs

is described more precisely. This provides an instance of how AEs can be used in language definition

Eliminating labels in terms of program-points

The discussion of the preceding section but one will now be resumed. Using program-point declarations, the example given there can be satisfactorily rendered as follows:

```
begin real x;
          S1;
          S2;
      L1: S3;
          S4;
   L2: L3: S5;
          S6;
      L4: S7;
          S8
end
```

```
begin real x;
  programpoint L1;
    begin S3; S4; L2 end;
  programpoint L2; L3;
  programpoint L3;
    begin S5; S6; L4 end;
  programpoint L4;
    begin S7; S8 end;
  S1;
  S2;
  L1
end
```

Strictly speaking, the use of '**programpoint**' rather than '**procedure**' only affects the outcome in the case of L_4. However, the above treatment has the superficial merit of uniformity, and two more important ones

- It caters also for natural exits from the middle of a procedure (e.g. ALGOL 58's **return**, or PL/1's `RETURN`)

- It does not jeopardize modelling the run-time difference between calling a closed routine and jumping, i.e., between making provision for resumption and making no such provision.

The considerations that led to program-points were only the first of two difficulties that arise in exploiting the similarity between labels and parameterless non-type procedures. We now come to the other.

In the above examples, a block (or procedure body) is considered as a listing of labelled and unlabelled items, but the treatment of labels inside such items is not explained. Thus, the above example was crucially a block; had it been a compound statement, the program-point declarations would have had a larger scope. For example, consider (only to reject) the following:

```
begin real x;
   S1;
   if p then goto L1
        else begin S2;
                  L1: S3;
                      S4;
                  L2: S5;
                      S6;
             end;
   S7;
   S8
end
```

```
begin real x;
   programpoint L1;
      begin S3; S4; L2 end;
   programpoint L2;
      begin S5; S6 end;
   S1;
   if p then L1
        else begin S2; L1 end;
   S7;
   S8
end
```

Now suppose S_1 contains a jump to L_2 and suppose S_5, S_6 contain no jumps. Then the exit from L_2 will cause exit from the block; it will *not* lead to S_7. This error can be avoided by a preliminary transformation of the ALGOL 60 that removes instances of the critical situation, by adding a jump (possibly to a newly added label) at the end of every compound statement. Thus the above example becomes:

```
begin real x;
   S1;
   if p then goto L1
        else begin S2;
                  L1: S3;
                      S4;
                  L2: S5;
                      S6;
                      goto M
             end;
M  S7;
   S8
end
```

```
begin real x;
   programpoint L1;
      begin S3; S4; L2 end;
   programpoint L2;
      begin S5; S6; M end;
   programpoint M;
      begin S7; S8 end;
   S1;
   if p then L1
        else begin S2; L1 end;
   M
end
```

As can be seen, the re-arrangement consists, roughly speaking, of ensuring that the labelled statement and its successors are not inside a conditional statement, and hence that jumps into the middle of a conditional statement do not occur.

A jump into the middle of a compound statement can be removed by an analogous re-arrangement. (It is unlikely to occur except in the context of a jump into a conditional statement, as in the example above.)

Since a jump into a block or **for**-statement is not ALGOL 60, these rearrangements suffice to transform any block into one amenable to the treatment illustrated above. However, it should be observed that the presence of labels inside conditional statements does not mean that transcription into program-points *necessarily* involves this re-arrangement. This precaution is obligatory only for any statement that can be entered unnaturally and exit naturally, e.g. in its most primitive form:

if p **then goto** L
  else L: ;

if p **then goto** L
  else begin L: **goto** M **end**; M:

There are various ways in which the criterion for applying this transformation can be made coarser and easier to specify. For instance, the transformation can be applied to *every* compound or arm of a conditional that contains a label and does not end in a jump. Or to every compound or arm. Or we can add jumps and labels to ensure that every non-jump is followed by a jump.

Eliminating pps in terms of labels

This section reverses the discussion of the preceding one. It is clear that a parameterless non-type pp'-declaration can easily be eliminated by introducing a label. The only possible complication is not serious and arises from ALGOL 60's lack of the ALGOL 58 **return** facility.

begin pp' $f(x)$;
  begin
  ...
  end
  ...
  ... $f(a)$ ...
  ...
end

begin proc $f(x)$;
  begin
  ...
  goto L
  end
  ...
  ... $f(a)$ ...
  ...
L: **end**

The elimination of functional pp's is less direct, thus proving that they are a substantial addition to the language.

In fact, they must first be replaced by non-functional pps, using rules precisely analogous to those employed in [9] to eliminate functional procedures. (This involves decomposing any statement containing a call for a functional pp.) This complication is of course closely related to the runtime behavior needed when a functional procedure is exited unnaturally.

It follows from the above that any merits pps have above labels are likely to be more pronounced in the functional case. This is confirmed by my experiments.

It will be observed that every label introduced by the above rule immediately precedes a block-**end**; and every **goto**-statement it introduces immediately preceeds a procedure-body-**end**. These are very limited uses of labels and **goto**. This suggests that a programmer who attempts to use pps *instead of* labels is likely to arrive at a very different structure for his program. Some clue to the direction of their influence is provided by the above-observed greater complication of eliminating *functional* pps. It suggests that pps are likely to be a greater gift to a programmer with a bias towards using program-structure rather than explicit sequencing, i.e., towards "functional" rather than "imperative" modes of expression.

It also suggests that there are situations in which pps make program-structure a more attractive alternative to explicit sequencing than it would otherwise be. Hence, their influence is in the same general direction as that advocated in my "Getting Rid of Labels" [7].

The most striking instance of this is that they lack the asymmetrical approach of labels—i.e., the possibility of approaching a label both naturally (without naming it) and by jumping. A pp, like a procedure, can only be approached by explicitly naming it (or more precisely by an explicit expression that is equivalent to its name—e.g., ALGOL 60's "designational expressions").

The Relation between J and program-point declarations

This section establishes the precise correspondence between the "program-point declaration" extension to ALGOL 60, and the **J** extension to AEs. Thus, since the symbol pp was merely a more palatable way of writing certain occurrences of '**J**', the relation between pp and pp' is also established.

Hitherto, the "semantics" of program-point declarations have been described informally following the custom of the ALGOL 60 report. So this section, in fact, constitutes a formal definition of this language feature, settling a number of small questions that have so far been left unspecified.

Our point of departure is the straightforward relation between declarations and supporting definitions in AEs, e.g.,

begin
  real a;
  proc $f(x)$;
  ϕ_1;
  ϕ_2
end
	let $a = \mathbf{o.o}$
	let rec $f(x) = \phi_1$
	ϕ_2
		i.e.,
		$(\lambda a.(\lambda f.\phi_2)$
		$(\mathbf{Y}\lambda f x.\phi_1))$
		$(\mathbf{o.o})$

Also

begin
  real a;
  proc $f(x)$;
  ϕ_1;
  proc $g(x)$;
  ϕ_2;
  ϕ_3
end
	let $a = \mathbf{o.o}$
	let rec $(f(x) = \phi_1$
	and rec $g(x) = \phi_2)$
	ϕ_3
		i.e.,
		$(\lambda a.$
		$(\lambda(f,g).\phi_3)$
		$(\mathbf{Y}\lambda(f,g).$
		$(\lambda x.\phi_1, \lambda x.\phi_2))$
		$(\mathbf{o.o})$

Analogously we might expect

begin
  real a;
  pp $f(x)$;
  ϕ_1
  ϕ_2
end
	let $a = \mathbf{o.o}$
	let rec pp $f(x) = \phi_1$;
	ϕ_2
		i.e.,
		$(\lambda a.(\lambda f.\phi_2)$
		$(\mathbf{Y}\lambda f \mathbf{J} \lambda x.\phi_1))$
		$(\mathbf{o.o})$

and (incorrectly)

begin		i.e.,
real a;	**let** $a = \mathbf{o.o}$	
pp $f(x)$;	**let rec**	$(\boldsymbol{\lambda a.}$
ϕ_1;	(**pp** $f(x) = \phi_1$	$(\boldsymbol{\lambda}(f, g).\phi_3)$
pp $g(x)$;	**and pp** $g(x) = \phi_2$)	$(\mathbf{Y}\boldsymbol{\lambda}(f, g).$
ϕ_2;	ϕ_3	$(\mathbf{J}\ \boldsymbol{\lambda x.}\phi_1, \mathbf{J}\ \boldsymbol{\lambda x.}\phi_2))$
ϕ_3		$(\mathbf{o.o})$
end		

This last falls down on account of what seems an undesirable technical feature of **J**. A recursive definition implies an extra $\boldsymbol{\lambda}$-level, and the application of **J** must occur outside this level not inside it. (The former case, when there is just one pp, squeaks through, as the reader may prove by using the transition-rule for **J**.)

Instead we must use

begin	**let** $a = \mathbf{o.o}$	i.e.,
real a;	**let rec**	$(\boldsymbol{\lambda a.}$
pp $f(x)$;	$(f(x) = \phi_1$	$(\boldsymbol{\lambda}(f, g).\phi_3)$
ϕ_1;	**and** $g(x) = \phi_2)$	$(\mathbf{Y}\boldsymbol{\lambda}(f, g).(\mathbf{J}f, \mathbf{J}g))$
pp $g(x)$;	**let** $f = \mathbf{J}f$	$(\mathbf{Y}\boldsymbol{\lambda}(f, g).$
ϕ_2;	**and** $g = \mathbf{J}g$	$(\boldsymbol{\lambda x.}\phi_1, \boldsymbol{\lambda x.}\phi_2))$
ϕ_3	ϕ_3	$(\mathbf{o.o})$
end		

Whether or not this is evidence of some defect in **J** is a judgement that should await further experiment.[2]

Apart from this complication the relation between program-points and supporting definitions is the same as the corresponding treatment of other sorts of declarations. In fact, the procedures, switches, and pps of each block must be grouped in a single recursive definition.

None of the above examples include a *parameterless* procedure or pp. In rendering them as applicative expressions, the call must be made explicit, e.g., by using a null argument-list. So a '**goto**' (like a call for a parameterless procedure, and formal parameter called by name) gives rise to an empty bracket-pair, e.g.,

goto L	$L\ (\ )$
goto if p **then** L **else** $S\ \mathbf{3}$	**if** $(p)(L, S(\mathbf{3}))(\ )$

The general treatment can be relaxed in various special cases. E.g., if the first statement of a block is labeled, then we have

begin	**let** $a = \mathbf{o.o}$	i.e.,
real a;	**let rec**	$(\boldsymbol{\lambda} a.$
pp $f(x)$;	$(f(x) = \phi_1$	$\quad 1st$
$\quad \phi_1$;	**and** $g(x) = \phi_2)$	$\quad ((\boldsymbol{\lambda}(f,g).(\mathbf{J}f, \mathbf{J}g))$
pp $g(x)$;	**let** $f = \mathbf{J}f$	$\quad (\mathbf{Y}\boldsymbol{\lambda}(f,g).$
$\quad \phi_2$;	**and** $g = \mathbf{J}g$	$\quad\quad (\boldsymbol{\lambda} x.\phi_1, \boldsymbol{\lambda} x.\phi_2)))$
goto f	f	$(\mathbf{o.o})$
end		

Again, if there are no backward jumps, then the recursion need not include the labels. So a non-recursive, simultaneous definition suffices (and avoids the above mentioned complication concerning the $\boldsymbol{\lambda}$-level of the application of **J**.)

Conclusion

There is a chain of relations between

1. ALGOL 60's labels and jumps
2. The "program-point declaration" extension to ALGOL 60
3. The "pp-definition" extension to **where**-notation,
4. **J**
5. The SECD-machine

This appendix has completed the chain by establishing the first two links. There are two reasons for doing this. First, it provides an introduction to **J** for someone unfamiliar with the AE/SECD model. Second, it shows how the model can be used in specifying a language, in this case ALGOL 60 and an extension of it.

Acknowledgments

This report could not have been written without many discussions with W. H. Burge.

Notes

1. This work was carried out while the author was at Univac Systems Programming Research and originally appeared as a Univac technical report dated August 29, 1965.

2. Another example of **J**'s defiance of the usual substitution rules is the non-equivalence of

L = **J**L	L, M, N = *map* **J**(L, M, N)
and M = **J**M	**where rec** *map*$(f)\ X = $ *null* $X \rightarrow (\ )$
and N = **J**N	$f(h\,X)$: *map* $f\ (t\,X)$

The right hand version defines L, M, and N as pps that would, if called, lead back into the definiens of *map*. This has no conceivable use. The equivalence is restored by writing:

$$L, M, N = \mathit{map}(\mathbf{B}(\mathbf{J}I))(L, M, N)$$

where $\mathbf{B}$ is the function $\mathbf{B}(f)(g)(x) = f(g(x))$

References

1. Burge, W.H., "The Evaluation, Classification and Interpretation of expressions," Proceedings of the 19th National ACM Conference, 1964.
2. Gilmore, P.C., "An abstract computer with a LISP-like machine language without a label operator," In *Computer Programming and Formal Systems*, ed. Braffort, P., and Hirschberg, D., North Holland Publishing Co., Amsterdam, 1963.
3. Landin, P.J., "The Mechanical evaluation of expressions," Comp. J. *6*, pp. 308–320.
4. Landin, P.J., "A Correspondence between ALGOL 60 and Church's Lambda-notation," Comm. ACM *8*, 89–101, 158–165, 1965.
5. Landin, P.J., "A formal description of ALGOL 60." Presented at IFIP Working Conf., Baden, Sept. 1964.
6. Landin, P.J., "Programming without lmperatives—an Example," UNIVAC S.P. Research Report (March, 1965)
7. Landin, P.J., "Getting Rid of Labels," UNIVAC S.P. Research Report (July, 1965)
8. McCarthy, J., "Towards a mathematical science of computation." IFIP Munich Conference, 1962, North Holland, Amsterdam, 1963.
9. Van Wijngaarden, A., "Recursive definition of syntax and semantics." Presented at IFIP Working Conf., Baden, Sept. 1964.
10. Landin, P.J., "An Analysis of Assignment in Programming Languages," UNIVAC S.P. Research Report (September, 1965)