An Introduction to Landin's "A Generalization of Jumps and Labels"

HAYO THIELECKE   ht@dcs.qmw.ac.uk

QMW, *University of London*

Abstract. This note introduces Peter Landin's 1965 technical report "A Generalization of Jumps and Labels", which is reprinted in this volume. Its aim is to make that historic paper more accessible to the reader and to help reading it in context. To this end, we explain Landin's control operator **J** in more contemporary terms, and we recall Burge's solution to a technical problem in Landin's original account.

1. Introduction

In the mid-60's, Peter Landin was investigating functional programming with "Applicative Expressions", which he used to provide an account of ALGOL 60 [10]. To explicate **goto**, he introduced the operator **J** (for jump), in effect extending functional programming with control. This general control operator was powerful enough to have independent interest quite beyond the application to ALGOL: Landin documented the new paradigm in a series of technical reports [11, 12, 13]. The most significant of these reports is "A Generalization of Jumps and Labels" (reprinted in this volume on pages 9–27), as it presents and investigates the **J**-operator in its own right.

The generalization is a radical one: it introduces *first-class control* into programming languages for the first time. Control is first-class in that the generalized labels given by the **J**-operator can be passed as arguments to and returned from procedures. In that regard, **J** is recognized as a precursor of **call/cc** in Scheme [3, 4, 16, 22].

To the contemporary reader, the **J**-operator could be understood as giving the programmer access to continuations — despite the fact that the general concept of continuation was not yet discovered in 1965. (For a history of continuations, see Reynolds's historical survey [19].) Continuations appear, albeit implicitly, within the states of the SECD-machine. The **J**-operator makes the implicit control information contained in the dump component of the SECD-machine available to the programmer as a procedure. Several special cases of continuations are discussed, in particular the result continuation (return address, as it were) of a procedure, which Landin even calls *natural continuation*, as well as (downward) failure continuations, called *failure actions*, in an application of **J** to backtracking.

Because of the underlying SECD states, the word "state" is frequently used by Landin as synonymous to the modern "continuation". In this usage, it is close to the systems programmer's concept of state (as in: "the state of a suspended process is saved for later resumption"); it should not be confused with the current usage of "state" in semantics, in the sense of something to be changed by assignment. As emphasised in Landin's introduction, control is independent of assignment.

In the remainder of this note, we discuss the relationship of **J** to **call/cc** and **return**, comment on its semantics in terms of SECD machine, and conclude with some general remarks of a historical nature. This is intended to help read the paper in context, especially subsequent developments in the area of control operators.

2.   The J operator

The **J**-operator is typically used in an idiom like this:

$$f = \lambda x.$$
$$\quad \textbf{let } g_1 = \lambda y.N_1$$
$$\quad \textbf{and } g_2 = \textbf{J}\,(\lambda z.N_2)$$
$$\quad M$$

Here a procedure f contains some local procedures g_1 and g_2. The **let**-expression is Landin's unsugared way of writing procedures declared within some other procedure. (In Landin's notation, scope in expressed by indentation. Here the scope of g_1 and g_2 is M.) For comparison, first consider what happens when the local procedure g_1 is called: after the call to g_1 is completed, control returns to the point where g_1 was called, so that evaluation proceeds somewhere in M. Now consider g_2: the λ-abstraction $\lambda z.N_2$ is modified by an application of the **J**-operator to give g_2 a non-standard control behaviour. More precisely, when g_2 is called, it does not return to where it was called, but to where f was called. In particular, calling g_2 affords a means for a fast exit from the body of f, i.e., M. When reading Landin's paper, the reader may like to consult the parser on page 15 for an example of the use of **J**.

2.1.   **J** *as a generalized* **return**

One may motivate **J** by considering its common idiom $\textbf{J}I$, where $I = \lambda x.x$ is the identity combinator, as a generalized return function [1]. Although not usually described in terms of continuations, the **return** function is perhaps the most widely known example of a jump with arguments, i.e., continuation invocation. It is common practice to jump out of some nested structure with the overall result of the current function, even in quite trivial programs, e.g.:

```
int fact(int n)
{
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

Note that when functions may be nested, as in the Gnu dialect of C, the **return** refers to the nearest enclosing function. Idealising this, we may regard **return** as syntactic sugar for $\textbf{J}I$. But the crucial difference between a return function defined in terms of **J** and the return statement of C is that the former is a first-class function. One may pass it as an argument to

a different procedure, as in $g(\mathbf{J}I)$, giving g a means for an exit from its caller. Consider for example $f = (\lambda x. \ldots g(\mathbf{return}) \ldots)$ where $g = (\lambda r. \ldots r(a) \ldots)$. f hands off the ability to return from f to g. When g then calls its argument, the call of f that called g returns with a. It is perhaps worth emphasising that the binding is static: when g calls a **return** function defined in f, this causes f, and not g, to return. Conversely, one may retrieve **J** from first-class **return** by postcomposition [14]:

$$\mathbf{J}\, f = \mathbf{return} \circ f$$

2.2. **J** *compared to* **call/cc**

J and **call/cc** differ in their view as to where continuations are introduced. The latter, in keeping with the overall style of Scheme, is expression-oriented; the former slightly more procedure-oriented in that a special significance is attached to λ-abstractions among expressions. With **call/cc**, every expression has a continuation, which one can seize by wrapping the expression into a **call/cc** i.e., by writing $\mathbf{call/cc}(\lambda k.M)$. With **J**, the only places where continuations are introduced are λ-abstractions. **J** gives access to a continuation, but it is not the current one. Rather it is the result continuation of the statically enclosing λ. (Hence one may have to introduce additional λ-abstractions in order to seize the right continuation.)

So **J** subordinates continuations to procedures, in that continuations can only be introduced as the result continuation of some procedure, and continuations can only be used (invoked) indirectly by being attached to some procedure. (This subordination of some more general programming language construct to procedures is perhaps not without parallel. For instance, whereas in C, blocks with local variable declarations can appear anywhere, in Pascal, blocks have to be procedure bodies.)

A comparison between the **J**-operator and the by now more familiar **call/cc** was first made by Reynolds. Reynolds's paper on definitional interpreters [18] introduces the **escape**-construct, which is the binder variant of **call/cc**, i.e.,

$$\mathbf{escape}\, k \,\mathbf{in}\, M = \mathbf{call/cc}(\lambda k.M) \quad \text{and} \quad \mathbf{call/cc} = \lambda m.\mathbf{escape}\, k \,\mathbf{in}\, mk$$

Reynolds shows **J** and **escape** to be interdefinable:

$$\mathbf{let}\, g = \mathbf{J}(\lambda x.R_1) \,\mathbf{in}\, R_0$$
$$= \mathbf{escape}\, h \,\mathbf{in}\, \mathbf{let}\, g = \lambda x.h(R_1) \,\mathbf{in}\, R_0$$

Notice how **J** postcomposes a continuation h to its argument. Conversely,

$$\mathbf{escape}\, g \,\mathbf{in}\, R$$
$$= \mathbf{let}\, g = \mathbf{J}(\lambda x.x) \,\mathbf{in}\, R$$

Strictly speaking, these equivalences do not hold generally, but only immediately inside a λ-abstraction. The second can be adapted to a general equivalence by inserting a dummy λ-abstraction to make **J** grab the *current* continuation:

$$\mathbf{escape}\, g \,\mathbf{in}\, R$$
$$= (\lambda().\mathbf{escape}\, g \,\mathbf{in}\, R)\,()$$
$$= (\lambda().\mathbf{let}\, g = \mathbf{J}(\lambda x.x) \,\mathbf{in}\, R)\,()$$

For **call/cc**, such a λ is (fortuitously) already there to bind the argument:

$$\mathbf{call}/\mathbf{cc} = \lambda f.f(\mathbf{J}(\lambda x.x))$$

The reverse direction, i.e., defining **J** in terms of Scheme's **call/cc**, was studied by Felleisen [3]: he defines a syntactic embedding of **J** into Scheme, essentially as follows:

$$\begin{aligned} x^* &= x \\ (MN)^* &= M^* N^* \\ (\lambda x.M)^* &= \lambda x.\mathbf{call}/\mathbf{cc}(\lambda k.M^*[\mathbf{J} \mapsto \lambda f.k \circ f]) \end{aligned}$$

A crucial point is that this embedding does not map λ directly to λ.

Less formally, and in terms of Scheme-inspired nomenclature, it is worth emphasising that **J** is not of the form "**call-with-foo**"; rather it could be phrased as **compose-with-current-dump**. In particular, in an idiom like $\mathbf{J}(\lambda x.M)$, the x does not receive the current dump; instead, the function $\lambda x.M$ is postcomposed with the (continuation represented by the) current dump. In the same spirit, **return** could be dubbed **call-current-dump-with**. (**call-with-current-dump** would also be possible, but somewhat artificial.)

3.   The Transition-Rule of the extended SECD-machine

J is given an operational semantics by extending the SECD machine from "The mechanical evaluation of expressions" [9]. (Introductions to the basic SECD-machine without **J** can also be found in many functional programming textbooks [5, 7, 17].) Landin's original definition (on page 16), however, is somewhat incomplete, as pointed out by Felleisen [3]. These omissions were tacitly corrected by Burge in his textbook [1], which uses the SECD-machine, including **J**, as a foundation. For historical accuracy, the editors of this special issue chose to reprint the paper as it was. Hence we recall Burge's corrected definition in this note: see Figure 1 for a definition of Landin's function *Transform* with Burge's corrections.

The notation for the *Transform* function of the SECD machine follows that of Landin's other papers, e.g., [9]. It is largely self-explanatory or standard (like the McCarthy conditional $\rightarrow$). Colon ":" stands for list construction (**cons**), u is the unit list function; h and t are the head and tail functions for lists, respectively.

Compared to the original SECD-machine, there are three new transitions. There is one new command, **J**, that may appear in the control string; and there are two new kinds of values, stateappenders and program closures, that can be applied to an argument much like an ordinary function closure. (The stateappenders are Burge's addition. Landin only used program closures.) Thus the new arms of the conditional are

$$\begin{aligned} X = \mathbf{J} &\rightarrow \ldots \\ stateappender\, f &\rightarrow \ldots \\ progclosure\, f &\rightarrow \ldots \end{aligned}$$

A reader looking for an small example could consider evaluating an instance of the "unnatural exit" idiom $\mathbf{J}\,(\lambda x.x)$.

$$(v : S, E, \mathbf{J}(\lambda x.x) : ap : C, D)$$

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

Figure 1. The SECD transition function after Landin and Burge

is transformed into

$$(v : S', E', C', D'),$$

where $(S', E', C', D') = D$, just as if v had been returned by natural exit from a procedure call, that is, starting from $(v : S, E, (), D)$.

For more details on the semantics of **J**, see Burge [1], and Felleisen [3], also Felleisen and Friedman [4]. Reynolds [18] explains how SECD states can be regarded as "defunctionalized" continuations; see the textbook [6] for more details on defunctionalising continuation-passing interpreters.

4.   Conclusions

The **J**-operator was ahead of its time in that it not only predated **call/cc** by a decade, but even the dynamic non-local exits of MacLisp [20]. Moreover, unlike **J**, dynamic exits do not give an answer to Landin's question what it means for a function to return a label as its result. Consider a Lisp function returning a "label" (corresponding to a catch tag in Lisp):

```
(defun label-as-result ()
  (catch 'tag 'tag))
```

When called, this literally returns **tag**. One could argue that returning a catch tag is begging the question what is denoted by it. In the SECD formalism, an analogous function could be written as $\lambda().\mathbf{J}(\lambda x.x)$. This function returns a program closure. So, within the SECD framework, Landin could give an answer to what is the "thing denoted by a label". This answer was earlier, albeit much less abstract, than that by denotational semantics, i.e., continuations [21].

Apart from the historical significance of **J**, we could also point to some conceptual issues. **J** differs in two independent ways from **call/cc**:

1. **call/cc** seizes the current continuation, while **J** seizes (the continuation represented by) the current dump;

2. **call/cc** gives direct access to the continuation, while **J** requires a function to which the continuation is to be postcomposed.

Of these two differences, it is the first that makes the comparison between **J** and **call/cc** difficult. More significantly, it is also the cause for a certain pitfall in equational reasoning with terms that may contain **J**. Landin refers to this as an "undesirable technical feature" of **J** (page 26). The second point is whether postcomposition of a continuation to a function should be a special operation. Burge claims as an advantage of **J** that a "program closure has a function and a state", i.e., continuation, rather than "only a state" [1, pp. 86–87]. Similarly, Landin [14] argues for this feature of **J** in terms of implementation.

In the modern typed setting [2, 15] one could make this second point quite abstractly. Suppose we have both functions $f : A \to B$ and continuations $k : \neg A$, and that all we know about continuations is that they can be fed arguments of the appropriate type. Arguably the most fundamental fact to note is that for each function $f : A \to B$ one has another function, call it $\neg f$, that takes a continuation as its argument and postcomposes it to f, hence $\neg f : \neg B \to \neg A$. This makes $\neg$ an operation with the following type:

$$\neg : (A \to B) \to (\neg B \to \neg A)$$

Hofmann [8] notes that such an operation (which is definable in terms of **call/cc**) may be useful to reason with equational axioms, such as the "naturality" of **call/cc**. Thus one could argue that, quite apart from the "undesirable feature" of **J**, Landin and Burge's emphasis on postcomposition has been corroborated.

Acknowledgments

References

1. William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

2. Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proc. ACM Symp. Principles of Programming Languages*, pages 163–173, January 1991.

3. Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

4. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986.

5. Anthony Field and Peter Harrison. *Functional Programming*. Addison-Wesley, 1988.

6. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

7. Martin C. Henson. *Elements of Functional Languages*. Blackwell Scientific Publications, Oxford, 1987.

8. Martin Hofmann. Sound and complete axiomatizations of call-by-value control operators. *Math. Struct. in Comp. Science*, 1994.

9. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

10. Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, February and March 1965.

11. Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965. To be reprinted in Lisp and Symbolic Computation.

12. Peter J. Landin. Getting rid of labels. Technical report, UNIVAC Systems Programming Research, July 1965.

13. Peter J. Landin. Programming without imperatives—an example. Technical report, UNIVAC Systems Programming Research, March 1965.

14. Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In O. Danvy, editor, *ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes Series, 1997.

15. AT&T Bell Laboratories. *Standard ML of New Jersey — Base Environment*, 0.93 edition, February 1993.

16. Jonathan Rees and William Clinger (editors). Revised[4] report on the algorithmic language Scheme. *ACM Lisp Pointers IV*, July-September 1991.

17. Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.

18. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, August 1972.

19. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993.

20. Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. In Richard L. Wexelblat, editor, *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 231–270, New York, NY, USA, April 1993. ACM Press.

21. Christopher Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

22. Mitchell Wand. Continuation-based multiprocessing. *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.