A Final Scheme-Word on Landin's J-Operator

by

Matthias Felleisen

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 205

A Final Scheme-Word on Landin's J-Operator

by

Matthias Felleisen

November, 1986

This material is based on work supported in part by an IBM Fellowship Award to M. Felleisen, and by the National Science Foundation grant number DCR 85-01277.

A Final Scheme-Word on Landin's J-Operator

Matthias Felleisen

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405, USA

Abstract

Landin's J-operator was the first attempt to extend the λ-calculus with a non-functional control facility. We show in this note how the extended language can be embedded in Scheme. This finally clarifies the relationship between the J- and the call/cc-operator. Beyond the historical perspective the note simultaneously provides insight into the programming with continuations and the construction of language embeddings for the clarification of semantic issues.

J-operator programming with continuations embeddings

1. The λ-calculus and Landin's J-operator

Landin's J-operator [10, 11] was the first attempt to extend the λ-calculus with a non-functional control facility. It was added to model the semantics of Algol 60's labels and jumps. The concept of a continuation [7, 13] had not yet been invented and, in any case, Landin aimed for as direct a translation of Algol into the λ-calculus as possible.

In order to model labels and their interaction with Algol's block structure the J-operator must communicate with applications of λ-abstractions which serve as representations for blocks. Whenever a λ-abstraction is invoked, Landin's definitional SECD-machine remembers the current state so that it can perform the rest of a program's actions after the application is evaluated. A label is therefore the combination of this saved state with the remaining actions of a block. The formation of a labeled statement is accomplished by invoking J on a function which models the rest of the block. J grabs the currently saved state, combines it with its argument, and returns this function-like object: a program point. A **goto** is the invocation of a program point which reinstalls the saved state and runs its associated function.

A short example will clarify this particular use of J. Suppose we are given the block

$$\ldots \textbf{begin integer } x\textbf{; } \langle stmt1\rangle\textbf{; } L\textbf{: } \langle stmt2\rangle \textbf{ end} \ldots$$

where $\langle stmt1\rangle$ and $\langle stmt2\rangle$ are arbitrary statements, possibly containing **goto** statements to the label L. A Landinesque translation of this block becomes:

$$\ldots \textbf{let } x = 0 \textbf{ in let } L = \mathrm{J}\langle fstmt2\rangle \textbf{ in } \langle fstmt1\rangle \ldots$$

where $\langle fstmt\rangle$ stands for a functional representation of the respective statement. The last action of $\langle fstmt1\rangle$ would be a jump to L, *i.e.*, $L()$.

In contrast to ordinary Algol-labels program points are first-class objects in Landin's extended λ-calculus-language. They can be the result of an expression, bound to parameters, and stored in data structures. This property, in conjunction with the close ties to labels, made the J-operator a rather opaque concept and it did not become popular.

Ten years later Sussman and Steele [15] developed another λ-calculus-based programming language, called Scheme. Like Landin they added facilities to control the evaluation of programs in a non-functional fashion. Unlike Landin they could use the concept of a continuation and continuation-passing (cps) style programming [7, 13]. A continuation is a function-like abstraction of the rest of the program evaluation. In cps programming a program explicitly constructs, passes around, and invokes its current continuation. This explicit administration of the rest of the evaluation gives a program direct power over the evaluation process, but it makes the program complex and hard to understand. Scheme leaves the bookkeeping work to the underlying evaluation mechanism and instead provides a linguistic facility which gives a program access to the continuation when needed. This achieves the same effect as cps-programming and improves the comprehensibility.

Many non-functional control features of algorithmic languages can be directly transliterated into an equivalent Scheme construction [3]. Examples are function exits, backtrack points, and exception handlers. We refer to this concept as embedding or, less formally, as syntactic sugar [10]. The advantage of embeddings is twofold. First, an embedded expression is either manually or automatically replaced *in situ* by its Scheme counterpart without knowledge of the textual context. Embeddings thus avoid the overhead of an interpreter solution and are also much simpler than a compiler-like preprocessing which would result in a restructuring of the entire program. Second, it follows that the facilities of both languages, the

embedding and the embedded language, may freely interact if they respect each others semantic integrity.

Since Scheme's semantics is relatively clean and easy to understand, embeddings provide an alternative way to define the meaning of programming constructs. The J-operator language, however, has so far resisted attempts to describe it as syntactic sugar. Our goal in this note is to derive such a solution, validate its correctness, and thereby enhance our understanding of J and continuations.

The paper is self-contained with the exception of Section 5. In the next section we formalize Landin's SECD-language and machine and investigate the J-operator. Section 3 contains a subset of Scheme which suffices to express J. In Section 4 we develop an embedding of the SECD-J-language and argue its correctness. For the mathematically inclined reader we include Section 5 which presents the same result in the framework of denotational semantics. This part assumes some knowledge about denotational semantics and its use of continuations for the explanation of control facilities. Section 6 puts our work into perspective.

2. The SECD-language and machine

The pure SECD-machine is a definitional interpreter for the λ-calculus language. In order to distinguish the language from the traditional λ-calculus we refer to it as the SECD-language. It is defined in Figure 1. For some programming examples we recommend the text books of Burge [2] and of Abelson and Sussman [1].

The machine is a transition system in the sense of automata theory with an infinite set of states and a transition function. A machine state is a 4-tuple comprising a stack, an environment, a control string sequence, and a dump. A control string is an SECD-term or the unique symbol **ap**. An environment is a finite map from variables to values. Values are, for the moment, either constants or closures. Closures are machine representations of evaluated λ-abstractions; they are a pair containing a λ-abstraction and an environment which defines the meaning of the free variables in the abstraction. A stack is a sequence of values. Finally, a dump is either an empty dump or a machine state.

Conventions. The letters σ, ρ, γ, and δ stand for arbitrary stacks, environments, control strings, and dumps. The notation $\rho[x \leftarrow v]$ characterizes an environment which is like ρ except for the place x where it returns the value v. M, N and u, v range over arbitrary Λ-terms and values, respectively. We use square brackets $[\cdot, \cdot, \ldots]$ to denote sequences and angle brackets $\langle \cdot, \cdot, \ldots \rangle$ for tuples. ▪ represents the

Figure 1: The SECD-language

The improper symbols are λ, ., (, and). *Const* is a set of constants some of which may be functions; *Var* is a countable set of variables. The symbols $a, \ldots$ and $x, \ldots$ range over *Const* and *Var* as meta-variables but are also used as if they were elements of the sets. If f is a functional element in *Const* and a is a constant for which f is defined then $ap(f, a)$ denotes the result of the application of f to a. The *SECD-language* contains

- *constants:* a if $a \in Const$;
- *variables:* x if $x \in Var$;
- *abstractions:* $(\lambda x.M)$ if M is an SECD-term and $x \in Var$; x is called the *bound* variable in the expression M or the *formal parameter*, M is the *abstraction body* or just *body*;
- *applications:* (MN) if M,N are SECD-terms; M is sometimes called the *function part* or the *operator*, N is correspondingly called *argument part* or *operand.*

All variables that do not occur bound in an expression are called *unbound* or *free*.

empty sequence and the empty dump. **End of Conventions**

The transition function is displayed in Figure 2. The transition from one state to the next is dictated by the first component of the control string sequence. Constants and variables load associated values on the stack; abstractions produce closures as intermediate results. An application simply changes the control string sequence. This causes the machine to first evaluate the operand, then the operator, and finally to produce the result of the application. The symbol **ap** causes the machine to apply the top of the stack to the value in the second position. If the two are constants, the result of applying the function to the argument is left on the stack. A closure is invoked by running the function body in the closure environment extended by a binding of the formal parameter to the value of the argument. The current state is saved in the dump so that it can be reinstalled upon exhaustion of the control string sequence.

A program, that is, a term with no unbound variables, is evaluated on the

Figure 2: The SECD-transition function

$$\langle \sigma, \rho, [a, \gamma], \delta \rangle \longmapsto \langle [a, \sigma], \rho, \gamma, \delta \rangle$$
$$\langle \sigma, \rho, [x, \gamma], \delta \rangle \longmapsto \langle [\rho(x), \sigma], \rho, \gamma, \delta \rangle$$
$$\langle \sigma, \rho, [\lambda x.M, \gamma], \delta \rangle \longmapsto \langle [\langle \lambda x.M, \rho \rangle, \sigma], \rho, \gamma, \delta \rangle$$
$$\langle \sigma, \rho, [MN, \gamma], \delta \rangle \longmapsto \langle \sigma, \rho, [N, M, \mathbf{ap}, \gamma], \delta \rangle$$
$$\langle [\langle \lambda x.M, \rho_M \rangle, v, \sigma], \rho, [\mathbf{ap}, \gamma], \delta \rangle \longmapsto \langle \blacksquare, \rho_M[x \leftarrow v], M, \langle \sigma, \rho, \gamma, \delta \rangle \rangle$$
$$\langle [f, a, \sigma], \rho, [\mathbf{ap}, \gamma], \delta \rangle \longmapsto \langle [ap(f, a), \sigma], \rho, \gamma, \delta \rangle$$
$$\langle [v, \sigma], \rho, \blacksquare, \langle \sigma_0, \rho_0, \gamma_0, \delta_0 \rangle \rangle \longmapsto \langle [v, \sigma_0], \rho_0, \gamma_0, \delta_0 \rangle$$

SECD-machine by putting it in the control string position, and setting the stack, environment, and dump to the empty component, respectively. The machine terminates upon encountering a state with an empty control string and an empty dump. The value on the stack is the final result.

Since the machine evaluates both parts of an application before invoking a function on an argument, it is called an *applicative-order* machine. Plotkin [12] has shown that this strategy coincides with the standard reduction function of the λ-value-calculus and that this calculus is therefore the correct medium to reason about SECD-programs.

Each of the four state components serves a specific purpose during the evaluation. The control string generally decides about the next transition step. The environment defines the meaning of unbound variables in control strings. The stack corresponds to a sequence of intermediate results of the current closure. It is in principle the same stack that is commonly found in pocket calculator programs.

An empty dump corresponds to the stop instruction on a regular machine. When a closure is applied, a new dump is formed, which remembers the current state components. Indeed, this is the only time when a new dump is constructed. The dump components are reinstalled when the control string is exhausted. Since the SECD-machine is a deterministic transition system, the starting state of an evaluation sequence determines the unique result. Hence, a dump, which is a sequence of states, abstracts a transition sequence:

Fact 1. *An empty dump signals* stop*; a non-empty dump encodes the actions which*

must be performed after the invocation of a closure is completed.

Given this basic machinery, we can now proceed to discuss the J-operator.[1]

We introduce J into the framework of the SECD-machine as a variable with a predefined meaning [2, 10, 11]. For a complete definition of its meaning we have to extend the above system by three rules. First, we define what J means as a control string:

$$\langle \sigma, \rho, [\mathrm{J}, \gamma], \delta \rangle \longmapsto \langle [\langle \mathrm{J}, \delta \rangle, \sigma], \rho, \gamma, \delta \rangle \text{ if } \mathrm{J} \notin \mathit{Domain}(\rho) \qquad (J.con)$$

With the extra condition we preserve the possibility that some function uses J as a formal parameter. The tagged structure $\langle \mathrm{J}, \delta \rangle$ is called *stateappender* and must be considered as a new kind of value. Second, we specify what it means to invoke a stateappender:

$$\langle [\langle \mathrm{J}, \delta_J \rangle, v, \sigma], \rho, [\mathbf{ap}, \gamma], \delta \rangle \longmapsto \langle [\langle \mathrm{J}, v, \delta_J \rangle, \sigma], \rho, \gamma, \delta \rangle. \qquad (sa.ap)$$

The resulting object $\langle \mathrm{J}, v, \delta_J \rangle$ is called a *program point* and is yet another kind of value. Finally, we say how to evaluate a program point invocation:

$$\langle [\langle \mathrm{J}, u, \delta_J \rangle, v, \sigma], \rho, [\mathbf{ap}, \gamma], \delta \rangle \longmapsto \langle [u, v], \emptyset, [\mathbf{ap}], \delta_J \rangle. \qquad (pp.ap)$$

The three rules make use of two new kinds of values, both of which may be invoked on values. We refer to the union of functional constants, closures, stateappenders, and program points as *applicable values.*

From the formal definitions we can deduce the following

Fact 2.

a) J *evaluates to a stateappender which contains the* current *dump.*

b) *A program point is the combination of a state appender with an applicable value resulting from the invocation of the stateappender on this value.*

c) *The invocation of a program point throws away the current machine state, installs its dump in the dump position, and invokes the associated applicable value on its argument.*

Fact 1 and part a) of Fact 2 imply that J's meaning changes when a closure is invoked and a new dump is constructed. We can also reverse this perspective and state it from the point of view of an abstraction body:

[1] For the formalization of J we follow Burge's development [2, pp.81-87]. Landin's original account of the extended SECD-machine [11] is informal and, unfortunately, slightly inconsistent. While J is syntactically treated like a variable, its semantics is only defined for the operand position, *i.e.*, as if $(\mathrm{J}\, M)$ were a new syntactic form. Since this treatment is a special case of Burge's, we decided to investigate the latter.

Fact 3. *The evaluation of* J *yields a different stateappender in every function body and depends on where and when the respective closure is invoked.*

The characterization of J according to these facts suffices for our case study. In the next section we introduce the programming language Scheme. The above facts then enable us to define the SECD-J-language by a syntactic extension of Scheme.

3. Scheme

For the purpose of this paper we regard Scheme as a dialect of the pure SECD-language, *i.e.*, an applicative-order concretization of the λ-calculus. The syntax is displayed in Figure 3.

Figure 3: Concrete Syntax of Scheme

⟨*exp*⟩ ::= ⟨*const*⟩ |
⟨*var*⟩ |
(lambda (⟨*var*⟩**)** ⟨*exp*⟩**)** |
(⟨*exp*⟩ ⟨*exp*⟩**)** |
(define ⟨*var*⟩ ⟨*exp*⟩**)** |
⟨*syntactic-extension*⟩ (see text)

⟨*const*⟩ ::= 0 | 1 | 2 | ...

Beyond this core language Scheme contains a number of predefined operations one of which gives a program access to its current continuation. It is called *call-with-current-continuation* and is generally abbreviated call/cc. The application (call/cc M) grabs its continuation and then applies M to this continuation, that is, M is called with the current continuation. A continuation is a function-like representation of the rest of the program evaluation. When invoked on a value, it resumes the program as if the expression (call/cc M) had returned this value. Like constants and functions, continuations are first-class objects.

Reasoning about the behavior of continuations is difficult if based on a denotational or machine semantics. We have recently developed a syntactic theory which greatly facilitates reasoning with call/cc and continuations [6]. It yields the following two (informal) rules which, when added to the usual β-value- (substitution-) rule of the λ_v-calculus, allow a programmer to reason symbolically about the evaluation of programs in our subset of Scheme:

- the expression (call/cc M) is equivalent to an application of M to the textual evaluation context of the call/cc-expression: a continuation corresponds to the textual evaluation context
- when a context, *i.e.* a continuation, is invoked on a value, the current evaluation context is thrown away and the program continues with the continuation-context and the hole that was left behind by (call/cc M) is filled with the value.

For an application of these rules consider the expression

$$(((\text{call/cc}\,\mathbf{I})(\mathbf{K}\,\text{add1}))\,0),$$

where $\mathbf{I} = (\mathbf{lambda}\,(x)\,x)$ and $\mathbf{K} = (\mathbf{lambda}\,(y)\,(\mathbf{lambda}\,(x)\,y))$. The evaluation of this expression yields 1. This can be computed as follows:

$$\begin{aligned}
&(((\text{call/cc}\,\mathbf{I})(\mathbf{K}\,\text{add1}))\,0) && (1)\\
\rightarrow&(((\text{call/cc}\,\mathbf{I})(\mathbf{lambda}\,(x)\,\text{add1}))\,0) && (2)\\
\rightarrow&(((\mathbf{I}\,\gamma)(\mathbf{lambda}\,(x)\,\text{add1}))\,0)\ \text{where}\ \gamma \approx (([\ ](\mathbf{lambda}\,(x)\,\text{add1}))\,0) && (3)\\
\rightarrow&((\gamma\,(\mathbf{lambda}\,(x)\,\text{add1}))\,0) && (4)\\
\rightarrow&(((\mathbf{lambda}\,(x)\,\text{add1})(\mathbf{lambda}\,(x)\,\text{add1}))\,0) && (5)\\
\rightarrow&(\text{add1}\,0) \rightarrow 1. && (6)
\end{aligned}$$

Although the expression *per se* is rather pointless, its evaluation demonstrates how continuations are treated as first-class objects (lines 3 and 4) and how the grabing (lines 2 and 3) and invoking of continuations (lines 4 and 5) is easily performed with the above rules.

Full Scheme supports many other linguistic facilities such as the construction of arbitrary constants (**quote**), branching constructs (*e.g.*, **if**), side-effects (**set!**), modified forms of abstraction (multiple parameters, rest parameters), and definitions (**define**). Except for the last one, none are of any importance to our purpose. Definitions are expressions of the form

$$(\mathbf{define}\ \langle var\rangle\ \langle expression\rangle).$$

The result is undefined and the effect is that the initial environment contains a binding of ⟨*var*⟩ to the value of ⟨*expression*⟩. Definitions may be embedded in expressions[2] in which case they are closed in the current environment.

Many implementations of Scheme also contain a facility to define syntactic sugar. Especially convenient are the facilities found in Scheme 84 [8] and Chez Scheme [5]. For example, a syntactic variation of Landin's **let** such as

(let (⟨*var*⟩ ⟨*exp*⟩**)** ⟨*body*⟩**)**

abbreviates

((lambda (⟨*var*⟩**)** ⟨*body*⟩**)**⟨*exp*⟩**)**

and is added by entering

(extend-syntax () **[(let (**⟨*var*⟩ ⟨*exp*⟩**)** ⟨*body*⟩**) ((lambda (**⟨*var*⟩**)** ⟨*body*⟩**)**⟨*exp*⟩**)]).**

The effect is that the programmer has a new language available which is like the old one but has an additional syntactic form.

Every declaration of a syntactic extension (as used in this note) has two parts: a possibly empty list of variable names and a pair of pattern-expressions enclosed in brackets. The first pattern expression is called input pattern, the second one is the output pattern. When the syntax preprocessor discovers an actual instance of the input pattern in a program, it replaces this instance by an instance of the output pattern with the pattern variables appropriately instantiated. If the expansion creates a binding to a variable that is not in the actual input expression, this binding is transparent unless the variable name is included in the first argument of **extend-syntax**. Hence, in an **or**-expression specified by

(extend-syntax () **[(or** ⟨*exp*1⟩ ⟨*exp*2⟩**) (let (***v* ⟨*exp*1⟩**) (if** *v v* ⟨*exp*2⟩**)))])**

the variable *v* is inaccessible from within the subexpression. On the other hand, if we had entered the definition

(extend-syntax (*v***) [(or** ⟨*exp*1⟩ ⟨*exp*2⟩**) (let (***v* ⟨*exp*1⟩**) (if** *v v* ⟨*exp*2⟩**)))])**

v would have been accessible from within ⟨*exp*2⟩ and would have overridden any other meaning of *v* in ⟨*exp*2⟩.

---

[2] In dialects of Scheme which satisfy the current standard [4] this kind of expression must be modified so that the binding is established first and then the binding is side-effected to contain the proper value. Scheme 84 handles these definitions without rearrangements.

With the **extend-syntax** facility it is easy to embed other languages in Scheme. We refer the reader to Kohlbecker's dissertation [9] for more information on the capabilities of **extend-syntax** and a survey of embeddings based on this facility [9, pp.22–23]. In the next section we demonstrate how the SECD-J-language is embedded.

4. Embedding the SECD-J-language in Scheme

The design of a language embedding in Scheme generally proceeds in two steps. During the first step every (abstract) syntactic form of the embedded languaged is assigned a Scheme semantics via the definition of a syntactic extension. In the second step the semantic primitives are added to the initial environment. In many cases there is a trade-off between these two steps. Syntactic extensions may require modifications of semantic primitives and *vice versa*.

In the case at hand the syntax and semantics of the embedded core language is seemingly already subsumed by Scheme. The only semantic primitive is J. However, according to Fact 3, the evaluation of J yields a different value in different function bodies and invocations thereof. The most direct way to implement this is to define J in the initial environment and to rebind J for function bodies. The second part actually means that we need abstractions which are different from Scheme's **lambda**-abstractions. We accomplish this by defining a syntax for SECD-abstractions in Scheme:

(extend-syntax (...)
  [**(Lambda** (⟨*var*⟩) ⟨*body*⟩)
   (lambda (⟨*var*⟩)
     ... **(let** (J ...) ⟨*body*⟩))]).

Given these decisions we can turn our attention to the bindings of J.

The evaluation of J yields a stateappender according to Fact 2a). A stateappender acts like a closure: when invoked on a value, it immediately returns a program point. Since stateappenders and program points are applicable values, **lambda**-abstractions are valid representations for both. Hence, J should be bound to an expression like

(lambda (*f*) **(lambda** (*a*) ...))

where the underlined part represents the program point and f and a are the argument to the stateappender and program point, respectively. Fact 2c) finally requires that a program point invokes the argument to the stateappender on the argument to

the program point and reinstalls the dump. We can make the above approximation more precise:

$$\textbf{(lambda}\,(f)\,\textbf{(lambda}\,(a)\ldots(fa)\ldots)).$$

The ellipsis indicates the actions which implement the reinstallement of the dump.

Following Fact 1 an empty dump corresponds to a stop instruction. Invoking a program point which contains the empty dump means to stop the computation after having evaluated (fa). In Scheme "stopping the computation" means to invoke an initial continuation which returns the program to the read-eval-print loop of the interpreter system. From this we can deduce J's initial binding:

```
(call/cc
  (lambda (initk)
    (define J (lambda (f) (lambda (a) (initk(fa)))))))).
```

Clearly, *initk* stands for the empty textual context and, when invoked, simply returns its argument to the interpreter loop and stops.

A non-empty dump represents the rest of the computation of a closure invocation, *i.e.*, a dump is the continuation of a closure invocation. Hence the first action of a **Lambda**-body must be the grabbing of the current continuation. This is accomplished with a call/cc-application and thus the output pattern of **Lambda** starts out approximately like

$$\textbf{(lambda}\,(\langle var\rangle)\,(\text{call/cc}\,\textbf{(lambda}\,(j)\ldots))).$$

The only remaining question is what the new value of J is. J's arity stays the same, but, instead of invoking the initial continuation on (fa), J—following Fact 2—must now reinstall the current dump which means it must invoke j:

$$\textbf{(lambda}\,(f)\,\textbf{(lambda}\,(a)\,(j(fa)))).$$

Putting things together, we obtain the following syntactic extension for **Lambda** as a first approximation: