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

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

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

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

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:

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

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:

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

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:

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

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 (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

$$
\begin{array}{rcll}
\langle exp \rangle & ::= & \langle const \rangle \mid & \\
 & & \langle var \rangle \mid & \\
 & & \textbf{(lambda (}\langle var \rangle\textbf{)}\ \langle exp \rangle\textbf{)} \mid & \\
 & & (\langle exp \rangle\ \langle exp \rangle) \mid & \\
 & & \textbf{(define}\ \langle var \rangle\ \langle exp \rangle\textbf{)} \mid & \\
 & & \langle syntactic\text{-}extension \rangle & \text{(see text)} \\
 & & & \\
\langle const \rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots &
\end{array}
$$

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:

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

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 $\langle var \rangle$ to the value of $\langle expression \rangle$. 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

$$(\mathbf{let}\ ((\langle var \rangle\ \langle exp \rangle))\ \langle body \rangle)$$

abbreviates

$$((\mathbf{lambda}\ (\langle var \rangle)\ \langle body \rangle)\langle exp \rangle)$$

and is added by entering

$$(\mathbf{extend\text{-}syntax}\,()\ [(\mathbf{let}\ ((\langle var \rangle\ \langle exp \rangle))\ \langle body \rangle)\ \ ((\mathbf{lambda}\ (\langle var \rangle)\ \langle body \rangle)\langle exp \rangle)]).$$

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

$$(\mathbf{extend\text{-}syntax}\,()\ [(\mathbf{or}\ \langle exp1 \rangle\ \langle exp2 \rangle)\ \ (\mathbf{let}\ (v\ \langle exp1 \rangle)\ (\mathbf{if}\ v\ v\ \langle exp2 \rangle))])$$

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

$$(\mathbf{extend\text{-}syntax}\,(v)\ [(\mathbf{or}\ \langle exp1 \rangle\ \langle exp2 \rangle)\ \ (\mathbf{let}\ (v\ \langle exp1 \rangle)\ (\mathbf{if}\ v\ v\ \langle exp2 \rangle))])$$

v would have been accessible from within $\langle exp2 \rangle$ and would have overridden any other meaning of v in $\langle exp2 \rangle$.

[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:

$$\begin{array}{l}\textbf{(extend-syntax}\ (\ldots)\\ \quad \textbf{[(Lambda}\ \textbf{(}\langle \mathit{var}\rangle\textbf{)}\ \langle \mathit{body}\rangle\textbf{)}\\ \quad\ \textbf{(lambda}\ \textbf{(}\langle \mathit{var}\rangle\textbf{)}\\ \qquad\qquad \ldots\ \textbf{(let}\ \textbf{(}\mathrm{J}\ \ldots\textbf{)}\ \langle \mathit{body}\rangle\textbf{)))]}\textbf{)}.\end{array}$$

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

$$\textbf{(lambda}\ (f)\ \underline{\textbf{(lambda}\ (a)\ \ldots\textbf{)}}\textbf{)}$$

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:

```
(extend-syntax (J)
  [(Lambda (⟨var⟩) ⟨body⟩)
   (lambda (⟨var⟩)
      (call/cc
         (lambda (j)
            (let (J (lambda (f) (lambda (a) (j(fa))))) ⟨body⟩))))]).
```

Whereas the variable j must be transparent because it only represents the raw continuation, the new definition of J must be accessible from within $\langle body \rangle$. Therefore we put J into the variable-name part of **extend-syntax**. This, however, leads to a subtle mistake in our definition. The binding of the variable J shadows the binding of the formal parameter $\langle var \rangle$. If $\langle var \rangle$ stands for the variable name J, the argument will be inaccessible from within the abstraction body. To provide the programmer with the possibility of using J as a formal parameter, we must interchange the binding order. This is done by introducing a transparent, intermediate variable y:

```
(extend-syntax (J)
  [(Lambda (⟨var⟩) ⟨body⟩)
   (lambda (y)
      (call/cc
         (lambda (j)
            (let (J (lambda (f) (lambda (a) (j(fa)))))
              (let ((⟨var⟩ y) ⟨body⟩)))))]).
```

Lambda-abstractions together with the global definition of J and the rest of Scheme form a new language, say Scheme-J,[3] which realizes Landin's SECD-J-language. One point worth validating is that the redefinition of J within **Lambda** uses the correct continuation. For this purpose we exploit the informal rules from Section 3. Suppose we are at a point in the evaluation of a program where a **Lambda**-abstraction is applied in some textual context ...[ ]...:

$$\ldots((\mathbf{Lambda}\ (x)\ Body)Arg)\ldots.$$

[3] In Scheme 84 a programmer can redefine the core syntax and, hence, Scheme 84 could serve as the concretization of the SECD-J-language.

In expanded form this reads as

```
... ((lambda (y)
        (call/cc
          (lambda (j)
            (let (J (lambda (f) (lambda (a) (j(fa)))))
              (let (x y) Body)))))Arg)... .
```

According to the evaluation rules we continue with

```
...(call/cc
      (lambda (j)
        (let (J (lambda (f) (lambda (a) (j(fa)))))
          (let (x ValofArg) Body))))...,
```

where *ValofArg* is the *value* of the expression *Arg*, and finally

```
...(let (J (lambda (f) (lambda (a) (j(fa)))))
      (let (x ValofArg) Body))...,
```

where $j \approx \ldots[\ ]\ldots$.

This binding of j corresponds to the continuation of the original application, and therefore, when a program point created by J is invoked on a function F and an argument A, it will resume the program evaluation with

$$\ldots \mathit{ValofFA} \ldots,$$

if *ValofFA* is the result of invoking F on A. This precisely reflects the operational meaning of program points as described by the facts and equations in Section 2.

Remark. The preceding validation reveals that our solution is only partially correct. If a program invokes a large number of program points before it actually gets to resume the continuation j, it may run out of stack space where an SECD-program may have terminated. Technically speaking, the solution is not *tail-recursive*. It can be fixed by enforcing an invocation of the continuation *before* (fa) is evaluated or, equally, by delaying (fa) so that the continuation is truly invoked. The standard way of getting this kind of timing problem right is to *freeze* and *thaw* the appropriate actions. Freezing means to construct a nullary function; thawing is the converse: it means to apply a nullary function to no arguments. The optimizing version of **Lambda** is:

```
(extend-syntax (J)
  [(Lambda (⟨var⟩) ⟨body⟩)
   (lambda (y)
      ((call/cc
         (lambda (j)
            (let (J (lambda (f) (lambda (a) (j(lambda () (f a))))))
              (let ((⟨var⟩ y) ⟨body⟩)))))))]).
```

The definition of the initial binding for J must be modified in a similar way.
End Remark

A natural question arises at this point: Can the J-operator implement call/cc? The answer is straightforward. A **Lambda**-abstraction which realizes call/cc must take one argument and apply it to the continuation of the application. On the other hand, a program point always combines this continuation with some closure and, if the closure is the identity function, a program point *is* the desired continuation. Hence, we can define call/cc in the SECD-J-language by:[4]

```
(define call/cc
   (Lambda (f)
      (f(J(Lambda (x) x))))).
```

This definition shows that the two languages are directly equivalent. It is not necessary to write an interpreter in order to simulate one language in the other.

5. A denotational approach

In this section we present the J-operator in the standard denotational framework for a semantics of the SECD-language. Figure 4 recalls the usual definitions for the syntactic and semantic domains and equations [14]. We have omitted constants for the sake of simplicity.

Since J is a variable name with a predetermined denotation, the syntax of our language remains unchanged. The initial environment for $\mathcal{E}$ must contain a yet to be determined binding for J:

$$(\mathrm{J}, J_{\mathrm{init}}) \in \rho_{\mathrm{init}}.$$

[4] This is due to Reynolds who defined his **escape**-construct with J [13].

Figure 4: A Denotational Semantics of the SECD-language

Syntactic Domains:

$$\begin{aligned} x &\in \mathit{Var} && \text{(Variables)},\\ L, M, N &\in \Lambda && \text{(SECD-language)}. \end{aligned}$$

Syntax:

$$L ::= x \mid \lambda x.M \mid MN$$

Semantic Domains:

$$\begin{aligned} \rho \in \mathit{Env} &= \mathit{Var} \longrightarrow \mathit{Val} && \text{(Environments)},\\ m, n, v \in \mathit{Val} &= \mathit{Val} \longrightarrow \mathit{Cont} \longrightarrow \mathit{Val} && \text{(Values or Closures)},\\ \kappa, \kappa', \ldots \in \mathit{Cont} &= \mathit{Val} \longrightarrow \mathit{Val} && \text{(Continuations)}. \end{aligned}$$

Semantic Functions:

$$\mathcal{E} : \Lambda \longrightarrow \mathit{Env} \longrightarrow \mathit{Cont} \longrightarrow \mathit{Val},$$

$$\begin{aligned} \mathcal{E}[\![x]\!] &= \lambda\rho\kappa.\kappa(\rho[\![x]\!]),\\ \mathcal{E}[\![\lambda x.M]\!] &= \lambda\rho\kappa.\kappa(\lambda v\kappa'.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]\kappa'),\\ \mathcal{E}[\![MN]\!] &= \lambda\rho\kappa.\mathcal{E}[\![N]\!]\rho(\lambda n.\mathcal{E}[\![M]\!]\rho(\lambda m.mn\kappa)). \end{aligned}$$

In order to determine the denotation of J we need to construct domains for two new kind of values, namely, stateappenders and program points. The domain of values becomes:

$$\mathit{Val} = [\,\mathit{Val} \longrightarrow \mathit{Cont} \longrightarrow \mathit{Val}\,] + \mathit{SA} + \mathit{PP}.$$

Next we translate the facts of Section 2 into our denotational framework. Fact 1 says in the terms of the denotational semantics in Figure 4 that a dump corresponds to the continuation κ' which is passed to the closure. An initial dump is the initial continuation. From Fact 2 we know that the result of looking up J in the environment is a stateappender which, when invoked, immediately returns a program point, *i.e.*, it passes the program point to its continuation. Hence, the domain of stateappenders is defined by

$$SA = Val \longrightarrow Cont \longrightarrow PP.$$

Program points are also like closures, but they ignore their current dump and hence their continuation. Instead, they invoke their encoded applicable value on their argument and return the result value to the continuation κ', while ignoring their own. The domain of program points is therefore like the original domain of values and we have the following new definitions:

$$\begin{aligned} Val &= [\,Val \longrightarrow Cont \longrightarrow Val] + SA + PP, \\ SA &= Val \longrightarrow Cont \longrightarrow PP, \\ PP &= Val \longrightarrow Cont \longrightarrow Val. \end{aligned}$$

It is easy to see that these equations may be collapsed into the original equation for values and that we therefore do not need new domains.

We now know that the arity of J is $\lambda f \kappa_f. \ldots$ since the value of J is a stateappender. This stateappender returns a program point of type $\lambda a \kappa_a. \ldots$ which, if it is the initial one, runs the closure f on its argument a and then stops the machine. In denotational terms, initial program points throw away their continuation and instead use the initial continuation $\mathbf{I}$:

$$J_{\text{init}} = \lambda f \kappa_f . \kappa_f(\lambda a \kappa_a . (fa\mathbf{I})).$$

Furthermore, from Fact 3 we can deduce that J must be redefined with every closure application. This can be realized in at least two ways. The most direct transliteration would redefine the denotation of an application. The application would pass along the new value for J to a closure. But that would mean that we must also change the denotation of a closure. It would take the value for its argument, the new value for J, and the continuation of the application. Since this continuation is the only piece that J needs to form program points, we can also redefine J once the closure is called. In other words, we only need to modify the denotation of abstractions in order to give J the correct meaning within function bodies. We redefine the second semantic equation as:

$$\mathcal{E}[\![\lambda x.M]\!] = \lambda \rho \kappa . \kappa(\lambda v \kappa' . \mathcal{E}[\![M]\!]\rho[\mathrm{J} \leftarrow J_{\text{cl}}][x \leftarrow v]\kappa').$$

The binding of J and of the argument are serialized such that J may be used as a bound variable. The new denotation J_{cl} must be like the initial one, except that the program points which are produced use the application continuation:

$$J_{\text{cl}} = \lambda f \kappa_f . \kappa_f(\lambda a \kappa_a . (fa\kappa')).$$

In summary the two modifications to the semantics in Figure 4 are

$$(\mathrm{J}, (\lambda f \kappa_f . \kappa_f (\lambda a \kappa_a . f a \mathrm{I}))) \in \rho_{\mathrm{init}}$$

and

$$\mathcal{E}[\![\lambda x.M]\!] = \lambda \rho \kappa . \kappa (\lambda v \kappa' . \mathcal{E}[\![M]\!] \rho [\mathrm{J} \leftarrow \lambda f \kappa_f . \kappa_f (\lambda a \kappa_a . f a \kappa')][x \leftarrow v] \kappa').$$

The denotational characterization has much in common with the Scheme version of the preceding section. Both definitions provide an initial binding, both redefine the meaning of the abstraction mechanism in the language, and both use continuations as their major tool to describe the effect of J and program points.

Nevertheless we perceive a series of differences which underline some subtle points in the definition of J. Scheme's initial definition of J explicitly invokes the initial continuation and clarifies that the computation terminates with an invocation of an initial program point. Furthermore, by using the operator call/cc in order to obtain the proper continuation, the syntactic extension in Scheme makes unmistakenly clear which continuation is used by a program point, namely, the continuation of the closure application. The three explicit continuations in the denotational version are rather confusing. Lastly, the introduction of an auxiliary variable y in the syntactic extension for Scheme directs the reader's attention to the difficulty of binding J at the right time. This is not quite as simple as the serialization of the environment extensions, but the point becomes obvious. Although these differences are mainly due to the complicated nature of J, they also indicate a difference between Scheme as a defining language and denotational semantics. Whereas in the latter framework everything is explicit, definitions via Scheme concentrate on the essentials. Depending on the situation this may be an advantage or a disadvantage.

6. Conclusion

We have constructed a Scheme-embedding of the J-operator which essentially characterizes J with respect to the current continuation of a closure invocation. The inverse relation, namely, how to define a J-based facility which grabs the current continuation, has been known for a long time and is much simpler.

The asymmetry between the two embeddings indicates that the two operators are, although formally equivalent, in different complexity classes. This may also explain why J has generally been used in situations like

$$(\mathbf{Lambda}\,(x)\,(\mathbf{let}\,(\mathit{lab}\,(\mathrm{J}\,\mathit{fun}))\ldots)).$$

This expression is equivalent to the construction

$$(\textbf{lambda}\,(x)\,(\text{call/cc}\,(\textbf{lambda}\,(c)\,(\textbf{let}\,(\textit{lab}\,(\textbf{lambda}\,(x)\,(c(\textit{fun}\,x))))\ldots))))$$

and it is easy to see which continuation is contained in the program point *lab*. Furthermore, J and program points are closely entangled with the definition and meaning of closures and closure invocations. Indeed, J can only be understood by a redefinition of the abstraction mechanism. This unfortunately implies that equivalences like

$$((\textbf{Lambda}\,()\,\textit{Body})) = \textit{Body}$$

are no longer valid if *Body* contains references to J. The standard way of reasoning about abstractions and applications has almost become impossible.

In contrast, the call/cc-operation in Scheme only depends on the ubiquitous concept of the current continuation. Its meaning is independent of other language constructs and therefore, easier to comprehend. Because of this, it is also easy to model other non-functional control facilities with call/cc in a rather direct way. Scheme may thus serve as a natural target language for language definitions in the same style that Landin was aiming for with the SECD-J-language.

Acknowledgement. Dan Friedman originally directed our attention to the problem of embedding Landin's SECD-J-language in Scheme. Our note greatly benefited from Cameron Nagle's attempt to prove the impossibility of this task. Bruce Duba, Dan Friedman, and Chris Haynes provided useful comments on earlier drafts. The author gratefully acknowledges the support of an IBM Graduate Research Fellowship.

References

1. ABELSON, H., G.J. SUSSMAN. *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, 1985.

2. BURGE, W. *Recursive Programming Techniques*, Addison-Wesley, 1975.

3. CLINGER, W.D., D.P. FRIEDMAN, M. WAND. A scheme for a higher-level semantic algebra, in *Algebraic Methods in Semantics*, J. Reynolds, M.Nivat (Eds.), 1985, 237–250.

4. CLINGER, W., REES, J. (Eds.) The revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21**(11), 1986, to appear.

5. DYBVIG, R. K. *The Scheme Programming Language*, Prentice Hall, 1987, in press.

6. FELLEISEN, M., D.P. FRIEDMAN. Control operators, the SECD-machine, and the λ-calculus, *Formal Description of Programming Concepts III*, North-Holland, Amsterdam, 1986, to appear.

7. FISCHER, M.J. Lambda calculus schemata, *Proc. ACM Conference on Proving Assertions about Programs*, Las Cruces, *SIGPLAN Notices*, **7**(1), 1972, 104–109.

8. FRIEDMAN, D.P., C.T. HAYNES, E. KOHLBECKER, M. WAND. Scheme 84 Interim Reference Manual, Technical Report No. 153, Indiana Univeristy Computer Science Department, 1985.

9. KOHLBECKER, E. *Syntactic Extensions in the Programming Language Lisp*, Ph.D. dissertation, Indiana University Computer Science Department, 1986.

10. LANDIN, P.J. A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM,* **8**(2), 1965, 89–101; 158–165.

11. LANDIN, P.J. An abstract machine for designers of computing languages, *Proc. IFIP Congress*, 1965, 438–439.

12. PLOTKIN, G. D. Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science* **1**, 1975, 125–159.

13. REYNOLDS, J.C. Definitional interpreters for higher-order programming languages, *Proc. ACM Annual Conference*, 1972, 717–740.

14. STOY, J.E. *Denotational Semantics: The Scott-Stratchey Approach to Programming Languages*, The MIT Press, Cambridge, Massachusetts, 1981.

15. SUSSMAN G.J., G. STEELE. Scheme: An interpreter for extended lambda calculus, Memo 349, MIT AI-Lab, 1975.