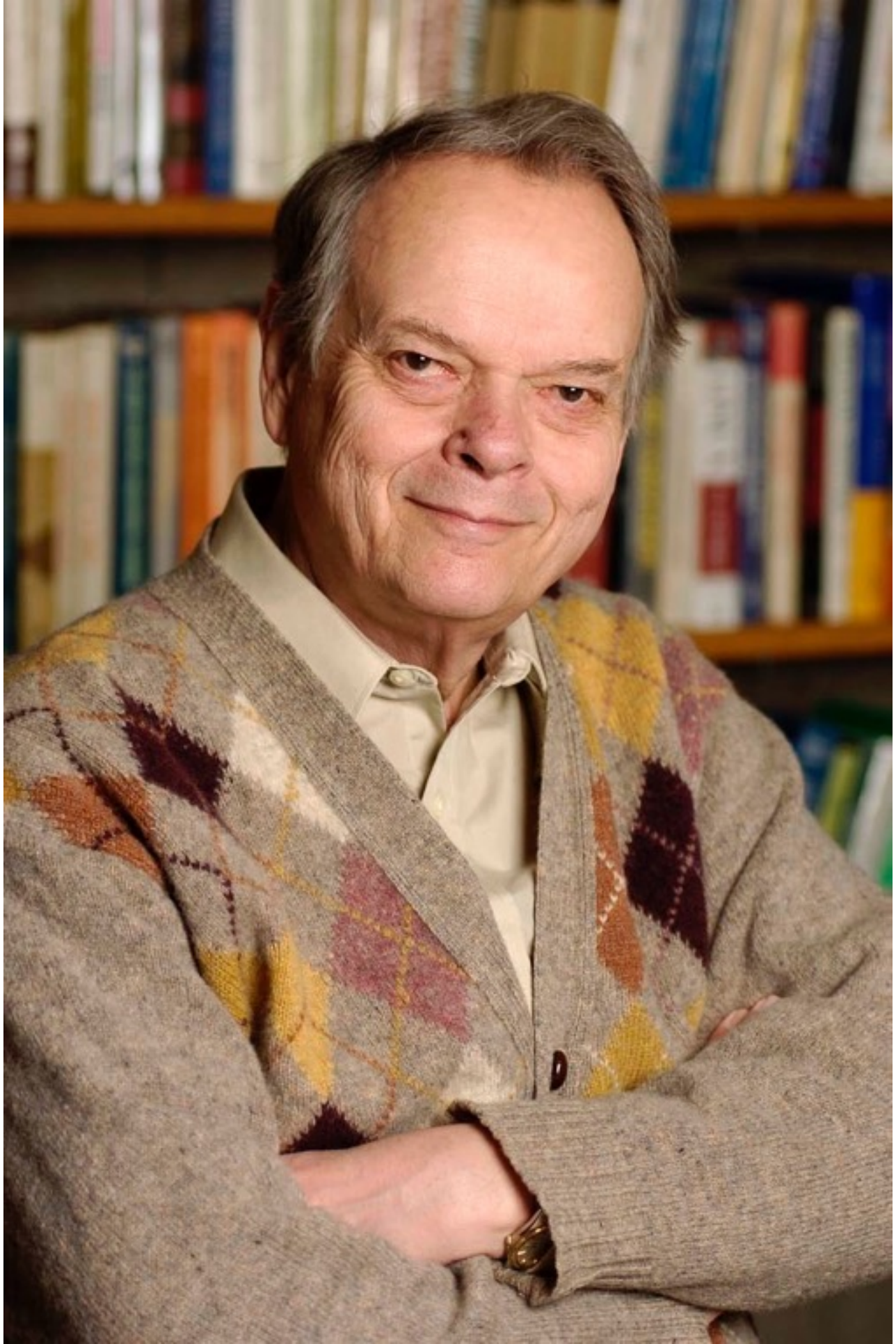
John Reynolds, Definitional Interpreters for Higher-Order Programming Languages

Philip Wadler
University of Edinburgh
Papers We Love, 7 June 2016




HOPE

Some other Papers I Love

John McCarthy, Toward a Mathematical Science of Computation, *IFIP Congress*, pages 21–28, 1962.

Peter Landin, The Next 700 Programming Languages, *CACM*, 9(3):157–166, March 1966.

Gordon Plotkin, Call-by-name, Call-by-value, and the Lambda Calculus, *TCS* 1:125–159, 1975.

John McCarthy presents Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *CACM* 3(4): 184—195, April 1960.

Philip Wadler explains
‘Propositions as Types’,
CACM, 58(12):75—84,
December 2015.

The Papers

John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, in *Proceedings of the ACM Annual Conference*, Volume 2, pages 717—740, August 1972.

John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, in *Higher-Order and Symbolic Computation*, 11(4):363—397, 1998.

John Reynolds, Definitional Interpreters Revisited, in *Higher-Order and Symbolic Computation*, 11(4):355—361, 1998.

John Reynolds, The Discoveries of Continuations, in *Lisp and Symbolic Computation*, 6:233—248, 1993.

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Reynolds/Definitional

72-164

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus call-by-name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Key Words and Phrases: programming language, language definition, interpreter, lambda calculus, applicative language, higher-order function, closure, order of application, continuation, LISP, GEDANKEN, PAL, SECD machine, J-operator, reference.
CR Categories: 4.20, 5.24, 4.13

†Work supported by Rome Air Force Development Center Contract No. 30602-72-C-0281 and ARPA Contract No. DAHC04-72-C-0003.

INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have tried to make the presentation accessible to readers who are unfamiliar with this previous work.

(Of course, interpretation can provide an implementation as well as a definition, but there are large practical differences between these usages. Definitional interpreters often achieve clarity by sacrificing all semblence of efficiency.)

We begin by noting some salient characteristics of programming languages themselves. The features of these languages can be divided usefully into two categories: *applicative* features, such as expression evaluation and the definition and application of functions, and *imperative* features, such as statement sequencing, labels, jumps, assignment, and procedural side-effects. Most user-oriented languages provide features in both categories. Although machine languages are usually completely imperative, there are few "higher-level" languages in this category. (IPL/V might be an example.) On the other hand, there is at least one well-known example of a purely applicative language: LISP. (i.e., the language defined in McCarthy's original paper.[1] Most LISP implementations provide an extended language including imperative features.) There are also several more recent, rather theoretical languages (ISWIM[2], PAL[3] and GEDANKEN[4]) which have been designed

Proc ACM 25th Nat. Conf 1972

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Reynolds/Definitional

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus

INTRODUCTION

An important and fr
method of defining a p
is to give an interpre
which is written in a
better understood lang
call these two languag
and *defining* languages
In this paper, we will
classify several varie
interpreters, and show
derived from one anoth
constructive methods.
approach to "construct
is original, the paper
attempt to review and

Higher-Order and Symbolic Computation, 11, 363–397 (1998)

Definitional Interpreters for Higher-Order Programming Languages*

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Keywords: programming language, language definition, interpreter, lambda calculus, applicative language

Higher-Order and Symbolic Computation, 11, 355–361 (1998)

Definitional Interpreters Revisited

JOHN C. REYNOLDS john.reynolds@cs.cmu.edu
School of Computer Science, Carnegie Mellon University

Abstract. To introduce the republication of "Definitional Interpreters for Higher-Order Programming Languages", the author recounts the circumstances of its creation, clarifies several obscurities, corrects a few mistakes, and briefly summarizes some more recent developments.

Keywords: operational semantics, denotational semantics, interpreter, lambda calculus, applicative language, functional language, metacircularity, higher-order function, defunctionalization, closure, call by value, call by name, continuation, continuation-passing-style transformation, LISP, ISWIM, PAL, Scheme, SECD machine, J-operator, escape, assignment.

In late 1971, Jean Sammet and Burt Leavenworth asked Art Evans and me to give a tutorial session on "the application of the lambda calculus to programming languages" at the 25th Anniversary ACM National Conference to be held the following summer. I had recently returned from a sabbatical at Queen Mary College in London, where I had immersed my-

LISP AND SYMBOLIC COMPUTATION: An International Journal, 6, 233–248, 1993

The Discoveries of Continuations

JOHN C. REYNOLDS (*John.Reynolds@cs.cmu.edu*)
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Keywords: Semantics, Continuation, Continuation-Passing Style

Abstract. We give a brief account of the discoveries of continuations and related concepts by A. van Wijngaarden, A. W. Mazurkiewicz, F. L. Morris, C. P. Wadsworth, J. H. Morris, M. J. Fischer, and S. K. Abdali.

In the early history of continuations, basic concepts were independently discovered an extraordinary number of times. This was due less to poor communication among computer scientists than to the rich variety of settings in which continuations were found useful: They underlie a method of program transformation (into continuation-passing style), a style of definitional interpreter (defining one language by an interpreter written in another language), and a style of denotational semantics (in the sense of

Definitional Interpreters for Higher-Order Programming Languages

Order-of application dependence	Use of higher-order functions: yes	no
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine Vienna definition

Order-of application dependence	Use of higher-order functions: yes	no
yes	I direct interpreter for GEDANKEN	II McCarthy's definition of LISP
no	IV Morris-Wadsworth method	III SECD machine Vienna definition

Next, we introduce a form of expression (due to Landin[7]) which is analogous to the block in ALGOL. If $x_1, \ldots, x_n$ are variables, and $r_1, \ldots, r_n$ and r_b are expressions, then

$$\underline{\text{let}}\ x_1 = r_1\ \underline{\text{and}}\ \ldots\ \underline{\text{and}}\ x_n = r_n\ \underline{\text{in}}\ r_b.$$

is a *let expression*, whose *declared variables* are $x_1, \ldots, x_n$, whose *declaring expressions* are $r_1, \ldots, r_n$, and whose *body* is r_b. (We will call each pair $x_i = r_i$ a *declaration*.) The evaluation of a let expression in an environment e begins with the evaluation of its declaring expressions r_i in the same environment. Then the value of the let expression is obtained by evaluating its body r_b in the environment which is the extension of e which binds each declared variable x_i to the value of the corresponding declaring expression r_i.

them.

Regardless of the specific nature of the data, there are three ways to introduce basic operations and tests into our applicative language:

(1) We may introduce constants denoting the basic functions (whose application will perform the basic operations and tests).

(2) We may introduce *predefined variables* denoting the basic functions. These variables differ from constants in that the programmer can redefine them with his own declarations. They are specified by introducing an *initial environment*, to be used for the evaluation of the entire program, which binds the predefined variables to their functional values.

(3) We may introduce special expressions whose evaluation will perform the basic operations and tests. Since this approach is used in most programming languages (and in mathematical notation), we will frequently use the common forms of arithmetic and boolean expressions without explanation.

THE DEFINED LANGUAGE

Although our defining language will use all of the features described in the previous section, along with appropriate basic operations and tests, the defined language will be considerably more limited, in order to avoid complications which would be out of place in an introductory paper. Specifically:

(1) Functions will be limited to a single argument. Thus all applicative expressions will have a single operand, and all lambda expression will have a single formal

ABSTRACT SYNTAX

We now turn our attention to the defining language. To permit the writing of interpreters, the values used in the defining language must include expressions of the defined language. At first sight, this suggests that we should use character strings as values denoting expressions, but this approach would enmesh us in questions of grammar and parsing which are beyond the scope of this paper. (An excellent review of these matters is contained in reference 17.)

Instead, we use the approach of *abstract syntax*, originally suggested by McCarthy(18). In this approach, it is assumed that programs are "really" abstract, hierarchically structured data objects, and that the character strings that one actually reads into the computer are simply representations of these abstract objects (in the same sense that digit strings are representations of integers). Thus the problems of grammar and parsing can be set aside as "input editing". (Of course, this does not eliminate these problems, but it separates them clearly from semantic considerations. See, for example, Wozencraft and Evans.(25))

We are left with two closely related problems: How to define sets of abstract expressions (and other structured data to be used by the interpreters), and how to define the basic functions for constructing, analyzing, and classifying these objects. Both problems are solved by introducing three forms of *abstract syntax equation*. (A more elaborate defined language would require a more complex treatment of abstract syntax, as given in Reference 13, for example.) Within these equations, upper case letter strings denote sets, and lower case letter

values of the defining language whose members *represent* the values of the defined language. However, since the variety of values provided in the defining language is richer than in the defined language, we have been able to represent each defined-language value by the same defining-language value. In our later interpreters this situation will change, and it will become more evident that VAL is a set of value representations.

Finally, we must define the set ENV of environments. Since the purpose of an environment is to specify the value which is bound to each variable, the simplest approach is to assume that an environment is a function from variables to values, i.e.,

ENV = VAR → VAL.

Within the various interpreters which we will present, each variable will range over some set defined by abstract syntax equations. For clarity, we will use different variables for different sets, as summarized in the following table:

Variable	Range	Variable	Range
r	EXP	e e'	ENV
x z	VAR	c c'	CONT
ℓ	LAMBDA	m m' m"	MEM
a b	VAL	rf	REF
f	FUNVAL	n	INTEGER

(The sets CONT, MEM, and REF will be defined later.)

A META-CIRCULAR INTERPRETER

Our first interpreter is a straightforward transcription of the informal language definition we have already given. Its central component is a function *eval* which produces the value of an expression *r* in a environment *e*:

```
eval = λ(r, e).                                                    I.1
    (const?(r) → evcon(r),                                         I.2
    var?(r) → e(r),                                                I.3
    appl?(r) → (eval(opr(r), e))(eval(opnd(r),e)),                 I.4
    lambda?(r) → evlambda(r, e),                                   I.5
    cond?(r) → if eval(prem(r), e)                                 I.6
        then eval(conc(r), e) else eval(altr(r),e),                I.7
    letrec?(r) → letrec e' =                                       I.8
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)    I.9
        in eval(body(r), e'))                                      I.10
evlambda = λ(ℓ, e). λa. eval(body(ℓ), ext(fp(ℓ), a, e))            I.11
ext = λ(z, a, e). λx. if x = z then a else e(x)                    I.12
```

The subsidiary function *evlambda* produces the value of a lambda expression ℓ in

I

A meta-circular interpreter

Types: Syntax

```
EXP = CONST ∪ VAR ∪ APPL
                LAMBDA ∪ COND ∪ LETREC
APPL = [opr: EXP, opnd: EXP]
LAMBDA = [fp: VAR, body: EXP]
COND = [prem: EXP, conc: EXP,
                altr: EXP]
LETREC = [dvar: VAR, dexp: LAMBDA,
                body: EXP]
```

Types: Semantics

· VAL = INTEGER ∪ BOOLEAN ∪ FUNVAL

FUNVAL = VAL → VAL

A META-CIRCULAR INTERPRETER

Our first interpreter is a straightforward transcription of the informal language definition we have already given. Its central component is a function *eval* which produces the value of an expression *r* in a environment *e*:

```
eval = λ(r, e).                                                    I.1
    (const?(r) → evcon(r),                                         I.2
    var?(r) → e(r),                                                I.3
    appl?(r) → (eval(opr(r), e))(eval(opnd(r),e)),                 I.4
    lambda?(r) → evlambda(r, e),                                   I.5
    cond?(r) → if eval(prem(r), e)                                 I.6
        then eval(conc(r), e) else eval(altr(r),e),                I.7
    letrec?(r) → letrec e' =                                       I.8
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)    I.9
        in eval(body(r), e'))                                      I.10
evlambda = λ(ℓ, e). λa. eval(body(ℓ), ext(fp(ℓ), a, e))            I.11
ext = λ(z, a, e). λx. if x = z then a else e(x)                    I.12
```

```
interpret = λr. eval(r, initenv)                                   I.13
initenv = λx. ( x = "succ" → λa. succ(a),                          I.14
    x = "equal" → λa. λb. equal(a, b))                             I.15
```

(1) The meta-circular interpreter does not shed much light on the nature of higher-order functions. For this purpose, we would prefer an interpreter of a higher-order defined language which was written in a first-order defining language.

(2) Changing the order of application used in the defining language induces a similar change in the defined language. To see this, suppose that *eval* is applied to an application expression $r_0(r_1)$ of the defined language. Then the result of *eval* will be obtained by evaluating the application expression (line I.4)

$$(\text{eval}(r_0, e))(\text{eval}(r_1, e))$$

in the defining language. If call-by-value is used in the defining language, then $\text{eval}(r_1, e)$ will be evaluated before the functional value of $\text{eval}(r_0, e)$ is applied. But evaluating $\text{eval}(r_1, e)$ interprets the evaluation of r_1, and applying the value of $\text{eval}(r_0, e)$ interprets the application of the value of r_0. Thus in terms of the defined language, r_1 will be evaluated before the value of r_0 is applied, i.e., call-by-value will be used in the defined language.

(3) Suppose we wish to extend the defined language by introducing the imperative features of labels and jumps (including jumps out of blocks). As far as is known, it is impossible to extend the meta-circular definition straightforwardly to accommodate these features (without introducing similar features into the defining language).

In the next section we will develop transformations of the meta-circular interpreter which will meet the first two of these objections. Then we will find that the transformation designed to meet the second objection also meets the third.

II

Defunctionalisation

Location	Global Variables	New Record Equation
I.11	ℓ e	CLOSR = [lam: LAMBDA, en: ENV]
I.14	none	SC = []
I.15 (outer)	none	EQ1 = []
I.15 (inner)	a	EQ2 = [arg1: VAL]

Our remaining task is to replace each of the four lambda expressions by appropriate record-creation operations, and to insert expressions in the branches of *apply* which will interpret the corresponding records. The lambda expression in line I.11 must be replaced by an expression which creates a CLOSR-record containing the value of the global variables ℓ and e:

```
evlambda = λ(ℓ, e). mk-closr(ℓ, e)                                    I.11'
```

Now *apply(f, a)* must produce the result of applying the function represented by *f* to the argument *a*. When *f* is a CLOSR-record, this result may be obtained by evaluating the body of the eliminated lambda-expression:

```
eval(body(ℓ), ext(fp(ℓ), a, e))
```

in an appropriate environment. This environment must bind the formal parameter of the lambda expression to the value of *a* and must bind the global variables of the lambda expression to the same value as the environment in which the CLOSR-record was created. Since the latter values are stored in the fields of *f*, we have:

```
apply = λ(f, a).
    (closr?(f) → let a = a and ℓ = lam(f) and e = en(f)
        in eval(body(ℓ), ext(fp(ℓ), a, e)),
    ... )
```

```
FUNVAL = CLOSR ∪ SC ∪ EQ1 ∪ EQ2
CLOSR = [lam: LAMBDA, en: ENV]
SC = []
EQ1 = []
EQ2 = [arg1: VAL]
ENV = INIT ∪ SIMP ∪ REC
INIT = []
SIMP = [bvar: VAR, bval: VAL, old: ENV]
REC = [letx: LETREC, old: ENV]
```

```
interpret = λr. eval(r, mk-init())                                    II.1
eval = λ(r, e).                                                       II.2
    (const?(r) → evcon(r),                                            II.3
    var?(r) → get(e, r),                                              II.4
    appl?(r) → apply(eval(opr(r), e), eval(opnd(r), e)),              II.5
    lambda?(r) → mk-closr(r, e),                                      II.6
    cond?(r) → if eval(prem(r), e)                                    II.7
        then eval(conc(r), e) else eval(altr(r), e),                  II.8
    letrec?(r) → eval(body(r), mk-rec(r, e)))                         II.9
apply = λ(f, a).                                                      II.10
    (closr?(f) →                                                      II.11
        eval(body(lam(f)), mk-simp(fp(lam(f)), a, en(f))),            II.12
    sc?(f) → succ(a),                                                 II.13
    eq1?(f) → mk-eq2(a),                                              II.14
    eq2?(f) → equal(arg1(f), a))                                      II.15
get = λ(e, x).                                                        II.16
    (init?(e) → (x = "succ" → mk-sc(), x = "equal" → mk-eq1()),       II.17
    simp?(e) → if x = bvar(e) then bval(e) else get(old(e), x),       II.18
    rec?(e) → if x = dvar(letx(e))                                    II.19
        then mk-closr(dexp(letx(e)), e) else get(old(e), x))          II.20
```

```
                eval [e; a] = [
    atom [e] → assoc [e; a];
    atom [car [e]] → [
    eq [car [e]; QUOTE] → cadr [e];
    eq [car [e]; ATOM] → atom [eval [cadr [e]; a]];
    eq [car [e]; EQ] → [eval [cadr [e]; a] = eval [caddr [e]; a]];
    eq [car [e]; COND] → evcon [cdr [e]; a];
    eq [car [e]; CAR] → car [eval [cadr [e]; a]];
    eq [car [e]; CDR] → cdr [eval [cadr [e]; a]];
    eq [car [e]; CONS] → cons [eval [cadr [e]; a]; eval [caddr [e];
                a]]; T → eval [cons [assoc [car [e]; a];
                                        evlis [cdr [e]; a]]; a]];
    eq [caar [e]; LABEL] → eval [cons [caddar [e]; cdr [e]];
                                        cons [list [cadar [e]; car [e]; a]];
    eq [caar [e]; LAMBDA] → eval [caddar [e];
                                        append [pair [cadar [e]; evlis [cdr [e]; a]; a]]
```

III

Continuations

Next, suppose that we can divide the functions which may be applied by our program into *serious* functions, whose application may sometimes run on forever, and *trivial* functions, whose application will always terminate.

As can be seen with a little thought, the condition implies that whenever some function calls a serious function, the calling function must return the same result as the called function, without performing any further computation. But any function which calls a serious function must be serious itself. Thus by induction, as soon as any serious function returns a result, every function must immediately return the same result, which must therefore be the final result of the entire program.

Nevertheless, there is a method for transforming an arbitrary program into one which meets our apparently restrictive condition. The underlying idea has appeared in a variety of contexts,(19,20,21) but its application to definitional interpreters is due to Morris and Wadsworth.(15) Basically, one replaces each serious function f_{old} (except the main program) by a new serious function f_{new} which accepts an additional argument called a *continuation*. The continuation will be a function itself, and f_{new} is expected to compute the same result as *fold*, apply the continuation to this result, and then return the result of the continuation, i.e.,

$$f_{new}(x_1, \ldots, x_n, c) = c(f_{old}(x_1, \ldots, x_n)) \quad .$$

additional "degree of freedom" which

```
CONT = FIN ∪ EVOPN ∪ APFUN ∪ BRANCH
FIN = []
EVOPN = [ap: APPL, en: ENV, next: CONT]
APFUN = [fun: VAL, next: CONT]
BRANCH = [cn: COND, en: ENV, next: CONT]
FUNVAL, ENV, etc. = same as in Interpreter II.
```

```
interpret = λr. eval(r, mk-init(), mk-fin())
eval = λ(r, e, c).
    (const?(r) → cont(c, evcon(r)),
    var?(r) → cont(c, get(e, r)),
    appl?(r) → eval(opr(r), e, mk-evopn(r, e, c)),
    lambda?(r) → cont(c, mk-closr(r, e)),
    cond?(r) → eval(prem(r), e, mk-branch(r, e, c)),
    letrec?(r) → eval(body(r), mk-rec(r, e), c))
apply = λ(f, a, c).
    (closr?(f) →
        eval(body(lam(f)), mk-simp(fp(lam(f)), a, en(f)), c),
    sc?(f) → cont(c, succ(a)),
    eq1?(f) → cont(c, mk-eq2(a)),
    eq2?(f) → cont(c, equal(arg1(f), a)))
cont = λ(c, a).
    (fin?(c) → a,
    evopn?(c) → let f = a and r = ap(c) and e = en(c) and c = next(c)
        in eval(opnd(r), e, mk-apfun(f, c)),
    apfun?(c) → let f = fun(c) and c = next(c) in apply(f, a, c),
    branch?(c) → let b = a and r = cn(c) and e = en(c) and c = next(c)
        in if b then eval(conc(r), e, c) else eval(altr(r), e, c))
get = same as in Interpreter II.
```

IV
Continuations with higher-order functions (Continuation-Passing Style)

$$VAL = INTEGER \cup BOOLEAN \cup FUNVAL$$

$$FUNVAL = VAL, CONT \to VAL$$

$$ENV = VAR \to VAL$$

$$CONT = VAL \to VAL$$

```
interpret = λr. eval(r, initenv, λa. a)
eval = λ(r, e, c).
    (const?(r) → c(evcon(r)),
    var?(r) → c(e(r)),
    appl?(r) → eval(opr(r), e, λf. eval(opnd(r), e, λa. f(a, c))),
    lambda?(r) → c(evlambda(r, e)),
    cond?(r) → eval(prem(r), e,
        λb. if b then eval(conc(r), e, c) else eval(altr(r), e, c)),
    letrec?(r) → letrec e' =
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)
        in eval(body(r), e', c))
evlambda = λ(ℓ, e) . λ(a, c). eval(body(ℓ), ext(fp(ℓ), a, e), c)
ext = λ(z, a, e). λx. if x = z then a else e(x)
initenv = λx. (x = "succ" → λ(a, c). c(succ(a)),
    x = "equal" → λ(a, c). c( λ(b, c'). c'(equal(a, b))))
```

V

Escape expressions (Scheme's call/cc)

If (in the defined language) x is a variable and r is an expression, then

escape x in r

is an *escape expression*, whose *escape variable* is x and whose *body* is r. The evaluation of an escape expression in an environment e proceeds as follows:

(1) The body r is evaluated in the environment which is the extension of e which binds x to a function called the *escape function*.

(2) If the escape function is never applied during the evaluation of r, then the value of r becomes the value of the escape expression.

(3) If the escape function is applied to an argument a, then the evaluation of the body r is aborted, and a immediately becomes the value of the escape expression.

Essentially, an escape function is a kind of label, and its application is a kind of jump. The greater generality lies in the ability to pass arguments while jumping.

(Landin's J-operator can be defined in terms of the escape expression by regarding *let* $g = J\ \lambda x.\ r_1$ *in* r_0 as an abbreviation for *escape* h *in let* $g = \lambda x.\ h(r_1)$ *in* r_0, where h is a new variable not occurring in r_0 or r_1. Conversely, one can regard *escape* g *in* r as an abbreviation for *let* $g = J\ \lambda x.\ x$ *in* r.)

To handle escape expressions, we begin by

escape function.

Since *eval* is a serious function, its result, which is obtained by applying the continuation *c* to the value of the escape expression, must be the final result of the entire program being interpreted. This means that *c* itself must be a function which will accept the value of the escape expression and carry out the interpretation of the remainder of the program. But the member of FUNVAL representing the escape function is also serious, and must therefore also produce the final result of the entire program. Thus to abort the evaluation of the body and treat the argument *a* as the value of the escape expression, it is only necessary for the escape function to ignore its own continuation *c'*, and to apply the higher-level continuation *c* to *a*. Thus we have:

```
eval = λ(r, e, c). ( ...
    escp?(r) → eval(body(r), ext(escv(r), λ(a, c'). c(a), e), c) )
```

VI

Assignment

```
interpret = λr. eval(r, initenv, initmem, λ(m, a). a)
eval = λ(r, e, m, c).
     (const?(r) → c(m, evcon(r)),
     var?(r) → c(m, e(r)),
     appl?(r) → eval(opr(r), e, m,
        λ(m', f). eval(opnd(r), e, m', λ(m", a). f(a, m", c))),
     lambda?(r) → c(m, evlambda(r, e)),
     cond?(r) → eval(prem(r), e, m, λ(m', b). if b
        then eval(conc(r), e, m', c) else eval(altr(r), e, m', c)),
    letrec?(r) → letrec e' =
        λx. if x = dvar(r) then evlambda(dexp(r), e') else e(x)
        in eval(body(r), e', m, c),
    escp?(r) → eval(body(r),
        ext(escv(r), λ(a, m', c'). c(m', a), e), m, c))
evlambda = λ(ℓ, e). λ(a, m, c). eval(body(ℓ), ext(fp(ℓ), a, e), m, c)
ext = λ(z, a, e). λx. if x = z then a else e(x)
initenv = λx. (x = "succ" → λ(a, m, c). c(m, succ(a)),
    x = "equal" → λ(a, m, c). c(m, λ(b, m', c').c'(m', equal(a, b))))
```