Definitional interpreters for higher-order programming languages

Krzysztof Wróbel

Institute of Computer Science
University of Wrocław

strzkrzysiek@gmail.com

11 March, 2015

Motivation

- In 1972, that is at the time of Reynolds' research, there was no or very few methods to express the semantics of a given language.
- Most of the languages were usually defined by interpreters written in a programming language based on lambda calculus, that was hopefully better understood.
- One can see a problem that by writing such an interpreter some of the features of the defining language could be implicitely incorporated by the defined language (e.g. strategy of evaluation).
- The idea is to make somehow the defined language independent of the nature of the defining language.

The roadmap

- A simple applicative language
- Description of the defined language
- First and simple meta-cyclic interpreter
- Introducing CPS and defunctionalization
- A try to get rid of higher-order functions
- Independence from the strategy of evaluation – continuations
- A glimpse at some imperative features

Terminology

The defining language

The language our interpreters are written in.

The defined language

The language defined by those interpreters.

The defining language – variables and constants

Variables

Set of symbols that is evaluated to some value specified in the given environment – a mapping between variables and values.

Constants

We will not specify the set of constants precisely, but it should contain at least integers and Boolean true and false. Their evaluation gives the same value regardless of the environment.

The defining language – syntax

Lambda abstraction

$\lambda(r_1, \ldots, r_n).r_{body}$

Application

$r_f(r_1, \ldots, r_n)$

Simple conditional expression

if r_p **then** r_c **else** r_a

Multiple conditional expression

$(r_{p1} \to r_{c1}, \ldots, r_{pn} \to r_{cn})$ is equivalent to
if r_{p1} **then** r_{c1} **else** $\ldots$ **if** r_{pn} **then** r_{cn} **else error**

The defining language – syntax

Let expression

let $x_1 = r_1$ **and** $\ldots$ **and** $x_n = r_n$ **in** r_b

Recursive let expression

letrec $x_1 = r_1$ **and** $\ldots$ **and** $x_n = r_n$ **in** r_b

The defined language

- Functions will be limited to a single argument. Thus all applicative expressions will have a single operand, and all lambda expressions will have a single formal parameter.
- Only call by value will be used.
- Only simple conditional expressions will be used.
- Nonrecursive let expressions will be excluded.
- All recursive let expressions will contain a single declaration and their declaring expressions can be only in a form of a lambda expression.
- Values will be integers, boolean, and functions (actually closures).
- Basic operations will be *succ* (returns the successor of an integer n) and *equal* (tests integer equality).

Abstract syntax of the defined language

Since this is beyond the scope of this talk, we won't be bothering about lexing and parsing the program. Instead, we will consider a program to be already in a form of a *abstract syntax tree*.

The nodes in that tree will be represented as records together with some adequate accessors, constructors, and classifiers. Consider a set S_0 of all records of the same "type". We will write:

$$S_0 = [a_1 : S_1, \ldots, a_n : S_n]$$

where fields of these records are elements of respective set S_i and a_i denotes an accessor to the ith field. Moreover, implicitly we declare here a constructor $mk\text{-}s_0$ of n arguments and a classifier s_0? that test whether its argument belongs to S_0.

Abstract syntax of the defined language

Right now we are ready to define the data structures that will be used by the interpreter.

$\text{EXP} = \text{CONST} \cup \text{VAR} \cup \text{APPL} \cup \text{LAMBDA} \cup \text{COND} \cup \text{LETREC}$

$\text{APPL} = [\textit{opr}\colon \text{EXP}, \textit{opnd}\colon \text{EXP}]$
$\text{LAMBDA} = [\textit{fp}\colon \text{VAR}, \textit{body}\colon \text{EXP}]$
$\text{COND} = [\textit{prem} : \text{EXP}, \textit{conc} : \text{EXP}, \textit{altr} : \text{EXP}]$
$\text{LETREC} = [\textit{dvar}\colon \text{VAR}, \textit{dexp}\colon \text{LAMBDA}, \textit{body}\colon \text{EXP}]$

$\text{VAL} = \text{INTEGER} \cup \text{BOOLEAN} \cup \text{FUNVAL}$
$\text{FUNVAL} = \text{VAL} \rightarrow \text{VAL}$

$\text{ENV} = \text{VAR} \rightarrow \text{VAL}$

Finally... a meta-circular interpreter

$$
\begin{aligned}
&eval = \lambda(r, e).\\
&\quad (const?(r) \rightarrow evcon(r),\\
&\quad var?(r) \rightarrow e(r),\\
&\quad appl?(r) \rightarrow (eval(opr(r), e))(eval(opnd(r), e)),\\
&\quad lambda?(r) \rightarrow evlambda(r, e),\\
&\quad cond?(r) \rightarrow \textbf{if } eval(prem(r), e)\\
&\qquad\quad \textbf{then } eval(conc(r), e) \textbf{ else } eval(altr(r), e),\\
&\quad letrec?(r) \rightarrow \textbf{letrec } e' =\\
&\qquad\qquad \underline{\lambda x.\ \textbf{if } x = dvar(r) \textbf{ then } evlambda(dexp(r), e') \textbf{ else } e(x)}\\
&\qquad\quad \textbf{in } eval(body(r), e'))
\end{aligned}
$$

Finally... a meta-circular interpreter

$$evlambda = \lambda(l, e).\underline{\lambda a.eval(body(l), ext(fp(l), a, e))}$$

$$ext = \lambda(z, a, e).\lambda x.\ \textbf{if}\ x = z\ \textbf{then}\ a\ \textbf{else}\ e(x)$$

$$interpret = \lambda r.eval(r, initenv)$$

$$initenv = \lambda x.(x = \text{"succ"} \rightarrow \underline{\lambda a.succ(a)},$$
$$x = \text{"equal"} \rightarrow \underline{\lambda a.\underline{\lambda b.equal(a, b)}})$$

Before we proceed... CPS

On the next slides we will use two techniques: converting to CPS and defunctionalization. It would be good to begin with some simple examples. We will start with converting a simple factorial function to CPS.

Direct style

$fact = \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * fact(n-1)$

Continuation-passing style

$fact\text{-}c = \lambda(n, c).\ \textbf{if}\ n = 0\ \textbf{then}\ c(1)\ \textbf{else}\ fact\text{-}c(n-1, \lambda m.c(n*m))$
$fact = \lambda n.fact\text{-}c(n, \lambda x.x)$

Before we proceed... CPS

Another example of converting a program to CPS. We will write a function that multiplies all elements in a given list. In order to do that we have to extend our language with functions $empty?(l)$, $head(l)$ and $tail(l)$.

Direct style

$$mult = \lambda l.\ \textbf{if}\ empty?(l)\ \textbf{then}\ 1\ \textbf{else}\ head(l) * mult(tail(l))$$

Continuation-passing style

$$\begin{aligned}
mult\text{-}c = \lambda(l, c).\ &\textbf{if}\ empty?(l)\ \textbf{then}\ c(1) \\
&\textbf{else if}\ equal(head(l), 0)\ \textbf{then}\ 0 \\
&\textbf{else}\ mult\text{-}c(tail(l), \lambda m.c(head(l) * m))
\end{aligned}$$

$$mult = \lambda n.mult\text{-}c(n, \lambda x.x)$$

Before we proceed... Defunctionalization

The aim of defunctionalization is to get rid of use of higher-order features of our language. This means that we don't want any function either to be an argument of another function or to be returned by a function. Recall the example with factorial function.

$$fact\text{-}c = \lambda(n, c).\ \textbf{if}\ n = 0\ \textbf{then}\ c(1)\ \textbf{else}\ fact\text{-}c(n-1, \underline{\lambda m.c(n * m)})$$

$$fact = \lambda n.fact\text{-}c(n, \underline{\lambda x.x})$$

Two underlined lambdas are here passed as an argument of a function. Obviously, we initially we wanted them to be functions since they are continuations. But maybe there is a way to represent them.
Indeed, there is.

Before we proceed... Defunctionalization

Closure

Evaluation of a lambda expression which binds all occurences of free variables to their values in a given environment.

We will represent that two lambda expressions as records which contain values of their global variables at the time of definition. Let's make use of already mentioned *record equations*.

MULT = [*arg*: INTEGER, *next*: CONT]	will represent	$\lambda m.n * c(m)$
INIT = []	will represent	$\lambda x.x$

CONT = MULT ∪ INIT

Before we proceed... Defunctionalization

Since right now our continuations are records, we cannot simply apply them to an integer. We will make the following transformation:

$c(n) \rightarrow cont(n, c)$

Now we can define the final transformed version of the factorial function.

$$
\begin{aligned}
\mathit{fact\text{-}c} &= \lambda(n, c).\ \textbf{if}\ n = 0\ \textbf{then}\ cont(1, c) \\
&\qquad\qquad\quad \textbf{else}\ \mathit{fact\text{-}c}(n - 1, \mathit{mk\text{-}mult}(n, c)) \\
\mathit{fact} &= \lambda n.\mathit{fact\text{-}c}(n, \mathit{mk\text{-}init}())
\end{aligned}
$$

$$
\begin{aligned}
cont = \lambda(a, c).(&\mathit{init}? \rightarrow a, \\
&\mathit{mult}? \rightarrow cont(arg(c) * a, next(c)))
\end{aligned}
$$

Defunctionalizing the meta-cyclic interpreter

Recall the meta-cyclic interpreter. Some of the structures used in it were represented with functions. These were:

- functional values / closures (FUNVAL),
- environment (ENV).

We will try to defunctionalize them to records.

- $(eval(opr(r), e))(eval(opnd(r), e)) \rightarrow apply(eval(opr(r), r), eval(opnd(r), e))$
- $e(r) \rightarrow get(e, r)$

Let's start with the set FUNVAL.

Defunctionalizing FUNVAL

In the code of the interpreter elements of the set FUNVAL were underlined with solid line. There were four of them and for each we define a seperate record equation.

$\lambda a.eval(body(l), ext(fp(l), a, e))$	$\mathrm{CLOSR} = [lam : \mathrm{LAMBDA}, en : \mathrm{ENV}]$
$\lambda a.succ(a)$	$\mathrm{SC} = []$
$\lambda a.\lambda b.equal(a, b)$	$\mathrm{EQ1} = []$
$\lambda b.equal(a, b)$	$\mathrm{EQ2} = [arg1 : \mathrm{VAL}]$

$$\mathrm{FUNVAL} = \mathrm{CLOSR} \cup \mathrm{SC} \cup \mathrm{EQ1} \cup \mathrm{EQ2}$$

Defunctionalizing FUNVAL

$$evlambda = \lambda(l, e).mk\text{-}closr(l, e)$$

$$initenv = \underline{\lambda x.(x = "\mathrm{succ}" \rightarrow mk\text{-}sc(),}$$
$$\underline{x = "\mathrm{equal}" \rightarrow mk\text{-}eq1())}$$

$$\begin{aligned}
apply = \lambda(f, a).\\
&(closr?(f) \rightarrow \textbf{let } l = lam(f) \textbf{ and } e = en(f)\\
&\quad \textbf{in } eval(body(l), ext(fp(l), a, e)),\\
&sc?(f) \rightarrow succ(a),\\
&eq1?(f) \rightarrow mk\text{-}eq2(a),\\
&eq2?(f) \rightarrow \textbf{let } b = a \textbf{ and } a = arg1(f) \textbf{ in } equal(a, b))
\end{aligned}$$

Defunctionalizing ENV

Similary, in the interpreter elements of set ENV were underlined with dashed line. There were three of them and again, we have three record equations.

An initial environment:
$\text{INIT} = []$

A simple extension of an environment:
$\text{SIMP} = [\mathit{bvar} : \text{VAR}, \mathit{bval} : \text{VAL}, \mathit{old} : \text{ENV}]$

A letrec extension of an environment:
$\text{REC} = [\mathit{letx} : \text{LETREC}, \mathit{old} : \text{ENV}, \sout{\mathit{new} : \text{ENV}}]$

$\text{ENV} = \text{INIT} \cup \text{SIMP} \cup \text{REC}$

Defunctionalizing ENV

Replacement of the three environment-producing lambda expression gives:

$\mathit{letrec?}(r) \to$ **let~~rec~~** $e' = \mathit{mk\text{-}rec}(r, e) \ldots$

$\mathit{ext} = \lambda(z, a, e).\mathit{mk\text{-}simpl}(z, a, e)$

$\mathit{initenv} = \mathit{mk\text{-}init}()$

Defunctionalizing ENV

...and the environment producing function is:

$$
\begin{aligned}
&get = \lambda(e, x).\\
&\quad (init?(e) \rightarrow (x = \text{"succ"} \rightarrow mk\text{-}sc(), x = \text{"equal"} \rightarrow mk\text{-}eq1()),\\
&\quad simp?(e) \rightarrow \ \textbf{let } z = bvar(e) \textbf{ and } a = bval(e) \textbf{ and } e = old(e)\\
&\qquad\quad \textbf{in if } x = z \textbf{ then } a \textbf{ else } get(e, x),\\
&\quad rec?(e) \rightarrow \ \textbf{let } r = letx(e) \textbf{ and } e = old(e) \textbf{ and } e' = e\\
&\qquad\quad \textbf{in if } x = dvar(r) \textbf{ then } evlambda(dexp(r), e') \textbf{ else } get(e, x))
\end{aligned}
$$

Finally... the second interpreter

$$
\begin{aligned}
&\mathit{interpret} = \lambda r.\mathit{eval}(r, \mathit{mk\text{-}init}()) \\
&\mathit{eval} = \lambda(r, e). \\
&\quad (\mathit{const?}(r) \rightarrow \mathit{evcon}(r), \\
&\quad \mathit{var?}(r) \rightarrow \mathit{get}(e, r), \\
&\quad \mathit{appl?}(r) \rightarrow \mathit{apply}(\mathit{eval}(\mathit{opr}(r), r), \mathit{eval}(\mathit{opnd}(r), e)), \\
&\quad \mathit{lambda?}(r) \rightarrow \mathit{mk\text{-}closr}(r, e), \\
&\quad \mathit{cond?}(r) \rightarrow \textbf{if } \mathit{eval}(\mathit{prem}(r), e) \\
&\quad\quad \textbf{then } \mathit{eval}(\mathit{conc}(r), e) \textbf{ else } \mathit{eval}(\mathit{altr}(r), e), \\
&\quad \mathit{letrec?}(r) \rightarrow \mathit{eval}(\mathit{body}(r), \mathit{mk\text{-}rec}(r, e)))
\end{aligned}
$$

Finally... the second interpreter

$$
\begin{aligned}
&apply = \lambda(f, a).\\
&\quad (closr?(f) \rightarrow\\
&\quad\quad eval(body(lam(f)), ext(fp(lam(f)), a, en(f))),\\
&\quad sc?(f) \rightarrow succ(a),\\
&\quad eq1?(f) \rightarrow mk\text{-}eq2(a),\\
&\quad eq2?(f) \rightarrow equal(arg1(f), a))\\
&get = \lambda(e, x).\\
&\quad (init?(e) \rightarrow (x = {''}\mathrm{succ}{''} \rightarrow mk\text{-}sc(), x = {''}\mathrm{equal}{''} \rightarrow mk\text{-}eq1()),\\
&\quad simp?(e) \rightarrow \textbf{ if } x = bvar(e) \textbf{ then } bval(e) \textbf{ else } get(old(e), x),\\
&\quad rec?(e) \rightarrow \textbf{ if } x = dvar(letx(e))\\
&\quad\quad \textbf{then } mk\text{-}closr(dexp(letx(e)), e) \textbf{ else } get(old(e), x))
\end{aligned}
$$

Non-terminating expressions and evaluation-strategy dependence

Consider an example where *exp* is non-termination and *f* terminates, and doesn't need the value of expression *exp*. Then the answer for a question whether the following expression terminates depends on the strategy of evaluation of the defining language.

$apply(eval(opr(r), e), eval(opnd(r), e))$

However, we wanted our defined language to incorporate call-by-value strategy and because of this, in our interpreter that expression should never terminate.

Continuations

To deal with this problem we will introduce continuations

CONT = VAL $\to$ VAL,

and change functions *interpret*, *eval* and *apply* to have the following form:

$$interpret = \lambda r.eval(r, mk\text{-}init(), \lambda a.a)$$
$$eval = \lambda(r, e, c)\ldots.$$
$$apply = \lambda(f, a, c)\ldots.$$

We will think our further actions to perform will be embdeded into those continuations. This will allow us to have the control of order of execution.

Continuations

For all of the trivial functions (i.e. definitely terminating) we will simply pass the result of their application to the current continuation.

$$
\begin{aligned}
& eval = \lambda(r, e, c). \\
& \quad (const?(r) \rightarrow c(evcon(r)), \\
& \quad var?(r) \rightarrow c(get(e, r)), \\
& \qquad \vdots \\
& \quad lambda?(r) \rightarrow c(mk\text{-}closr(r, e)), \ldots) \\
& apply = \lambda(f, a, c).(\ldots, \\
& \quad sc?(f) \rightarrow c(succ(a)), \\
& \quad eq1?(f) \rightarrow c(mk\text{-}eq2(a)), \\
& \quad eq2?(f) \rightarrow c(equal(arg1(f), a)))
\end{aligned}
$$

Continuations

In the following instructions we would like to pass the current continuation (i.e. actions we have to perform later) as an argument of *eval*.

$$letrec?(r) \rightarrow (eval(body(r), mk\text{-}rec(r, e), c))$$

$$\vdots$$

$$(closr?(f) \rightarrow$$
$$eval(body(lam(f)), mk\text{-}simp(fp(lam(f)), a, en(f)), c)$$

Continuations

We are left with two statements where are would like to force the order and strategy of evaluation(left-to-right and call-by-value).

$$appl?(r) \to eval(opr(r), e, \lambda f.eval(opnd(r), e, \lambda a.apply(f, a, c)))$$

$$cond?(r) \to eval(prem(r), e,$$
$$\lambda b.\ \textbf{if}\ b\ \textbf{then}\ eval(conc(r), e, c)\ \textbf{else}\ eval(altr(r), e, c))$$

The almost ready third interpreter

$$
\begin{aligned}
&interpret = \lambda r.eval(r, mk\text{-}init(), \underline{\lambda a.a})\\
&eval = \lambda(r, e, c).\\
&\quad (const?(r) \to c(evcon(r)),\\
&\quad var?(r) \to c(get(e, r)),\\
&\quad appl?(r) \to eval(opr(r), e, \underline{\lambda f.eval(opnd(r), e, \underline{\lambda a.apply(f, a, c)})}),\\
&\quad lambda?(r) \to c(mk\text{-}closr(r, e)),\\
&\quad cond?(r) \to eval(prem(r), e,\\
&\qquad \underline{\lambda b.\ \textbf{if}\ b\ \textbf{then}\ eval(conc(r), e, c)\ \textbf{else}\ eval(altr(r), e, c))}\\
&\quad letrec?(r) \to eval(body(r), mk\text{-}rec(r, e), c))
\end{aligned}
$$

The almost ready third interpreter

$$
\begin{aligned}
&apply = \lambda(f, a, c).\\
&\quad (closr?(f) \rightarrow\\
&\qquad eval(body(lam(f)), mk\text{-}simp(fp(lam(f)), a, en(f)), c),\\
&\quad sc?(f) \rightarrow c(succ(a)),\\
&\quad eq1?(f) \rightarrow c(mk\text{-}eq2(a)),\\
&\quad eq2?(f) \rightarrow c(equal(arg1(f), a)))\\
&get = \lambda(e, x).\\
&\quad (init?(e) \rightarrow (x = {}''\mathrm{succ}'' \rightarrow mk\text{-}sc(), x = {}''\mathrm{equal}'' \rightarrow mk\text{-}eq1()),\\
&\quad simp?(e) \rightarrow \ \mathbf{if}\ x = bvar(e)\ \mathbf{then}\ bval(e)\ \mathbf{else}\ get(old(e), x),\\
&\quad rec?(e) \rightarrow \ \mathbf{if}\ x = dvar(letx(e))\\
&\qquad \mathbf{then}\ mk\text{-}closr(dexp(letx(e)), e)\ \mathbf{else}\ get(old(e), x))
\end{aligned}
$$

Why almost?

By converting our interpreter to CPS, we have once again introduced higher-order functions. Since the conversion to first-order language is pretty mechanical, we will just write down the new record equations and then the fully ready, defunctionalized, strategy-of-evaluation independent interpreter.

The initial continuation – identity
$\text{FIN} = []$
Evaluate-operand continuation
$\text{EVOPN} = [ap : \text{APPL}, en : \text{ENV}, next : \text{CONT}]$
Apply-function continuation
$\text{APFUN} = [fun : \text{VAL}, next : \text{CONT}]$
Branch continuation
$\text{BRANCH} = [cn : \text{COND}, en : \text{ENV}, next : \text{CONT}]$

$\text{CONT} = \text{FIN} \cup \text{EVOPN} \cup \text{APFUN} \cup \text{BRANCH}$

Finally... the third interpreter

$$
\begin{aligned}
&interpret = \lambda r.eval(r, mk\text{-}init(), mk\text{-}fin())\\
&eval = \lambda(r, e, c).\\
&\quad (const?(r) \to cont(c, evcon(r)),\\
&\quad var?(r) \to cont(c, get(e, r)),\\
&\quad appl?(r) \to eval(opr(r), e, mk\text{-}evopn(r, e, c)),\\
&\quad lambda?(r) \to cont(c, mk\text{-}closr(r, e)),\\
&\quad cond?(r) \to eval(prem(r), e, mk\text{-}branch(r, e, c)),\\
&\quad letrec?(r) \to eval(body(r), mk\text{-}rec(r, e), c))
\end{aligned}
$$

Finally... the third interpreter

$$
\begin{aligned}
apply = \lambda(f, a, c).&\\
&(closr?(f) \rightarrow\\
&\qquad eval(body(lam(f)), mk\text{-}simp(fp(lam(f)), a, en(f)), c),\\
&sc?(f) \rightarrow cont(c, succ(a)),\\
&eq1?(f) \rightarrow cont(c, mk\text{-}eq2(a)),\\
&eq2?(f) \rightarrow cont(c, equal(arg1(f), a)))\\
get = \lambda(e, x).&\\
&(init?(e) \rightarrow (x = \text{"succ"} \rightarrow mk\text{-}sc(), x = \text{"equal"} \rightarrow mk\text{-}eq1()),\\
&simp?(e) \rightarrow \ \textbf{if } x = bvar(e) \textbf{ then } bval(e) \textbf{ else } get(old(e), x),\\
&rec?(e) \rightarrow \ \textbf{if } x = dvar(letx(e))\\
&\qquad \textbf{then } mk\text{-}closr(dexp(letx(e)), e) \textbf{ else } get(old(e), x))
\end{aligned}
$$

Finally... the third interpreter

$$
\begin{aligned}
&\mathit{cont} = \lambda(c, a).\\
&\quad (\mathit{fin?}(c) \to a,\\
&\quad \mathit{evopn?}(c) \to \textbf{let } f = a \textbf{ and } r = \mathit{ap}(c) \textbf{ and}\\
&\qquad\qquad\qquad\qquad e = \mathit{en}(c) \textbf{ and } c = \mathit{next}(c)\\
&\qquad\quad \textbf{in } \mathit{eval}(\mathit{opnd}(r), e, \mathit{mk\text{-}apfun}(f, c)),\\
&\quad \mathit{apfun?}(c) \to \textbf{let } f = \mathit{fun}(c) \textbf{ and } c = \mathit{next}(c) \textbf{ in } \mathit{apply}(f, a, c),\\
&\quad \mathit{branch?}(c) \to \textbf{let } b = a \textbf{ and } r = \mathit{cn}(c) \textbf{ and}\\
&\qquad\qquad\qquad\qquad e = \mathit{en}(c) \textbf{ and } c = \mathit{next}(c)\\
&\qquad\quad \textbf{in if } b \textbf{ then } \mathit{eval}(\mathit{conc}(r), e, c) \textbf{ else } \mathit{eval}(\mathit{altr}(r), e, c))
\end{aligned}
$$

Escape expressions

We will introduce now an imperative control mechanism.

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

escape x **in** r

is an *escape expression*. The evaluation of it in an environment e proceeds as follows:

- The body r is evaluated in the environment that is the extension of e that binds x to a function called the *escape expression*.
- If the escape function is never applied during the evaluation of r, then the value of r becomes the value of the escape expression.
- 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 function.

Escape expressions

In order to extend our interpreters to handle escape expressions, we begin by extending the abstract syntaxt appropriately:

$\text{EXP} = \ldots \cup \text{ESCP}$
$\text{ESCP} = [\mathit{escv} : \text{VAR}, \mathit{body} : \text{EXP}]$

Since the escape variable is bound to a function, we must add to the set FUNVAL a new kind of record that represents escape functions:

$\text{FUNVAL} = \ldots \cup \text{ESCF}$
$\text{ESCF} = [\mathit{cn} : \text{CONT}]$

Escape expressions

These records are created in the new branch of *eval*:

$$eval = \lambda(r, e, c).(\ldots,$$
$$escp?(r) \rightarrow eval(body(r), mk\text{-}simp(escv(r), mk\text{-}escf(c), e), c))$$

and are interpreted by a new branch of *apply*:

$$apply = \lambda(f, a, c).(\ldots,$$
$$escf?(f) \rightarrow cont(cn(f), a))$$

References

John C. Reynolds (1998)
Definitional interpreters for higher-order programming languages.
Higher-Order and Symbolic Computation 11(4), 363 – 397.

The End