λ-calculus (part 2)

A small language

$$
\begin{aligned}
E \triangleq\ & \mathrm{Nat} \\
\mid\ & E\ \mathit{op}\ E \\
\mid\ & \mathrm{Var} \\
\mid\ & \textbf{let}\ \mathrm{Var} = E\ \textbf{in}\ E
\end{aligned}
$$

2 reduction rules ($n \in \mathrm{Nat}$)

$n_1\ \mathit{op}\ n_2 \longrightarrow \mathrm{compute}(n_1\ \mathit{op}\ n_2)$ i.e. $1+2 \longrightarrow 3$

$\textbf{let}\ x = n\ \textbf{in}\ E \longrightarrow E[x:=n]$

Substitution

$m[x:=n] \rightarrow m$

$(E_1 \; op \; E_2)[x:=n] \rightarrow E_1[x:=n] \; op \; E_2[x:=n]$

$x[x:=n] \rightarrow n$

$y[x:=n] \rightarrow y \qquad (x \neq y)$

$(\textbf{let } x=E_1 \textbf{ in } E_2)[x:=n] \rightarrow$

$\qquad \textbf{let } x=E_1[x:=n] \textbf{ in } E_2$

$(\textbf{let } y=E_1 \textbf{ in } E_2)[x:=n] \rightarrow \qquad (x \neq y)$

$\qquad \textbf{let } y=E_1[x:=n] \textbf{ in } E_2[x:=n]$

Context

$$
\begin{aligned}
C \triangleq\ & [] \\
\mid\ & C\ \mathit{op}\ E \\
\mid\ & E\ \mathit{op}\ C \\
\mid\ & \textbf{let}\ \mathrm{Var} = C\ \textbf{in}\ E \\
\mid\ & \textbf{let}\ \mathrm{Var} = E\ \textbf{in}\ C
\end{aligned}
$$

Call by value context

$$
\begin{aligned}
C \triangleq\ & [] \\
\mid\ & C\ \mathit{op}\ E \\
\mid\ & n\ \mathit{op}\ C \\
\mid\ & \textbf{let}\ \mathrm{Var} = C\ \textbf{in}\ E
\end{aligned}
$$

$$\frac{e_1 \longrightarrow e_2}{C[e_1] \longrightarrow C[e_2]}$$

$\longrightarrow$ is a function

$\longrightarrow^*$ is still a function

$\longrightarrow^*$ contain eval : (E $\mapsto$ Nat)

Evaluation

$[\![n]\!] \rightarrow n$

$[\![E_1 \; op \; E_2]\!] \rightarrow [\![E_1]\!] \; op \; [\![E_2]\!]$

$[\![x]\!] \rightarrow ?$

$[\![\textbf{let } x{=}E_1 \textbf{ in } E_2]\!] \rightarrow [\![E_2]\!]$ with $x{=}\; [\![E_1]\!]$

Evaluation with environment

$[\![n]\!]_\varepsilon \rightarrow n$

$[\![E_1 \; op \; E_2]\!]_\varepsilon \rightarrow [\![E_1]\!]_\varepsilon \; op \; [\![E_2]\!]_\varepsilon$

$[\![x]\!]_\varepsilon \rightarrow (\text{get } \varepsilon \; x)$

$[\![\textbf{let } x{=}E_1 \textbf{ in } E_2]\!]_\varepsilon \rightarrow$

$\quad [\![E_2]\!]_{\varepsilon'}$ with $\varepsilon' = (\text{extend } \varepsilon \; x \; [\![E_1]\!]_\varepsilon)$

Compilation to λ-calculus

$[\![0]\!] \rightarrow \mathbf{0}$

$[\![n]\!] \rightarrow (\mathbf{1+}\ [\![\text{'n-1'}]\!])$

$[\![E_1\ op\ E_2]\!] \rightarrow (op\ [\![E_1]\!]\ [\![E_2]\!])$

$[\![x]\!] \rightarrow x$

$[\![\mathbf{let}\ x{=}E_1\ \mathbf{in}\ E_2]\!] \rightarrow (\lambda x.[\![E_2]\!]\ [\![E_1]\!])$

denotational semantics

Compilation to λ-calculus with environment

$[\![0]\!] \rightarrow \lambda\varepsilon.\mathbf{0}$

$[\![n]\!] \rightarrow \lambda\varepsilon.(\mathbf{1+}\ ([\![\text{'n-1'}]\!]\ \varepsilon))$

$[\![E_1\ op\ E_2]\!] \rightarrow \lambda\varepsilon.(op\ ([\![E_1]\!]\ \varepsilon)\ ([\![E_2]\!]\ \varepsilon))$

$[\![x]\!] \rightarrow \lambda\varepsilon.(\text{get}\ \varepsilon\ x)$

$[\![\mathbf{let}\ x{=}E_1\ \mathbf{in}\ E_2]\!] \rightarrow$

$\lambda\varepsilon.([\![E_2]\!]\ (\text{extend}\ \varepsilon\ x\ ([\![E_1]\!]\ \varepsilon)))$

but 'get' need to use Var and eqVar

The beauty of de Bruijn indexes

$[\![0]\!]_\rho \rightarrow \lambda\varepsilon.\mathbf{0}$

$[\![n]\!]_\rho \rightarrow \lambda\varepsilon.(\mathbf{1+}\ ([\![\text{'n-1'}]\!]_\rho\ \varepsilon))$

$[\![E_1\ op\ E_2]\!]_\rho \rightarrow \lambda\varepsilon.(op\ ([\![E_1]\!]_\rho\ \varepsilon)\ ([\![E_2]\!]_\rho\ \varepsilon))$

$[\![x]\!]_\rho \rightarrow \lambda\varepsilon.(\mathbf{nth}\ \text{'dBI x }\rho\text{'}\ \varepsilon) \rightarrow (\mathbf{nth}\ \text{'dBI x }\rho\text{'})$

$[\![\mathbf{let}\ x=E_1\ \mathbf{in}\ E_2]\!]_\rho \rightarrow$

$\lambda\varepsilon.([\![E_2]\!]_{(x\bullet\rho)}\ (\text{cons}\ ([\![E_1]\!]_\rho\ \varepsilon)\ \varepsilon))$

$\text{dBI x } (x\bullet\rho) \rightarrow \mathbf{0}$

$\text{dBI x } (y\bullet\rho) \rightarrow (\mathbf{1+}\ (\text{dBI x }\rho))$

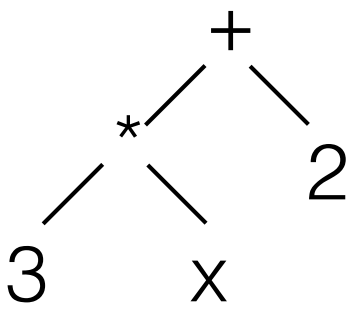
In short

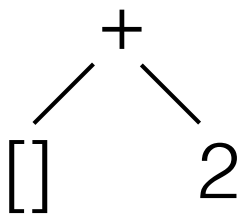
$$
\begin{aligned}
[\![x]\!]_\varepsilon &= (\varepsilon\ x) && \text{(Variable)} \\
[\![\lambda x.B]\!]_\varepsilon &= \lambda v.[\![B]\!]_{\varepsilon[x \to v]} && \text{(Abstraction)} \\
[\![(F\ A)]\!]_\varepsilon &= ([\![F]\!]_\varepsilon\ [\![A]\!]_\varepsilon) && \text{(Call)}
\end{aligned}
$$

trace of evaluation

Continuation

Intuition : A continuation denote what remains to do at a certain point





[] + 2

λr.r+2

First order

$[\![n]\!]\varepsilon\kappa \rightarrow (\kappa\ n)$

$[\![E_1\ op\ E_2]\!]\varepsilon\kappa \rightarrow [\![E_1]\!]\varepsilon\kappa'$
with $\kappa' = \lambda v_1.\ [\![E_2]\!]\varepsilon\kappa''$
with $\kappa'' = \lambda v_2.\ (k\ (v_1\ op\ v_2))$

$[\![x]\!]\varepsilon\kappa \rightarrow (\kappa\ (get\ \varepsilon\ x))$

$[\![\textbf{let}\ x = E_1\ \textbf{in}\ E_2]\!]\varepsilon\kappa \rightarrow [\![E_1]\!]\varepsilon\kappa'$
with $\kappa' = \lambda v.\ [\![E_2]\!]\varepsilon'\kappa$
with $\varepsilon' = (extend\ \varepsilon\ x\ v)$

Hight order first try

$[\![x]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ (\text{get}\ \varepsilon\ x))$

$[\![\lambda x.B]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ \lambda v.\ [\![B]\!]_{\varepsilon'\kappa})$

with $\varepsilon' = (\text{extend}\ \varepsilon\ x\ v)$

$[\![(F\ A)]\!]_{\varepsilon\kappa} \rightarrow [\![F]\!]_{\varepsilon\kappa'}$

with $\kappa' = \lambda f.[\![A]\!]_{\varepsilon\kappa''}$

with $\kappa'' = \lambda a.(k\ (f\ a))$

Hight order

$[\![x]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ (\text{get}\ \varepsilon\ x))$

$[\![\lambda x.B]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ \lambda vr.\ [\![B]\!]_{\varepsilon' r})$

with $\varepsilon' = (\text{extend}\ \varepsilon\ x\ v)$

$[\![(F\ A)]\!]_{\varepsilon\kappa} \rightarrow [\![F]\!]_{\varepsilon\kappa'}$

with $\kappa' = \lambda f.[\![A]\!]_{\varepsilon\kappa''}$

with $\kappa'' = \lambda a.(f\ a\ \kappa)$

Term → Env → (Value → Value) → Value

Continuation Passing Style

CPS transformation

$[\![x]\!]_\kappa \rightarrow (\kappa\ x)$

$[\![\lambda x.B]\!]_\kappa \rightarrow (\kappa\ \lambda v\underline{r}.\ [\![B]\!]_{\underline{r}})$

$[\![(F\ A)]\!]_\kappa \rightarrow [\![F]\!]_{\kappa'}$ with $\kappa' = \lambda\underline{f}.[\![A]\!]_{\kappa''}$

with $\kappa'' = \lambda a.(\underline{f}\ a\ \kappa)$

Term → (Value → Value) → Term

Reification

Could we bring back the continuation
to the user level ?

call/cc = call-with-current-continuation

```
(define (test)
   (+ (call/cc
         (lambda (cont)
            (begin (print "calling cont")
                   (let ( (r (cont 3)) )
                      (print "called cont”)
                      (+ r 1) ))))
      10 ))
```

The meaning of call/cc

$[\![\text{call/cc}]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ \lambda f.(f\ k))$ *abstractions have 2 args*

$[\![\text{call/cc}]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ \lambda fk'.(f\ k'\ k'))$ *arguments have 2 diff types*

$[\![\text{call/cc}]\!]_{\varepsilon\kappa} \rightarrow (\kappa\ \lambda fk'.(f\ \ \lambda vk''.(k'\ v)\ \ k'))$

puzzles : (call/cc call/cc)
((call/cc call/cc) call/cc)
((call/cc call/cc) (call/cc call/cc))

SECD machine

S(tack) *Sequence of temporary values*

E(nvironment) *Sequence of variable values*

C(ode) *Sequence of instructions*

D(ump) *Sequence of saved states*

Instructions

Code ≜ Sequence<Inst>

Inst ≜	**ldf** Code	*Create a closure*
|	**ld** Nat	*Load a variable's value*
|	**app**	*Apply a value to a closure*
|	**ret**	*Return from an application*

λ-calculus compilation

⟦Term⟧ → Code

works only
for closed terms

⟦x^i⟧ → [(**ld** i)]
⟦λx.B⟧ → [(**ldf** ⟦B⟧ • [**ret**])]
⟦(F A)⟧ → ⟦F⟧ • ⟦A⟧ • [**app**]

Code = [i_1 … i_N]

- appends two sequence of instructions

⟦λx.x^0⟧ → [(**ldf** [(**ld** 0) **ret**])] = (0R)
⟦λx.(x^0x^0)⟧ → [(**ldf** [(**ld** 0) (**ld** 0) **app ret**])] = (00AR)
⟦λx.λy.x⟧ → ((1R)R)

Running the machine

Machine -> Machine

<s,e,(**ld** i)·c,d> → <(get e i)·s,e,c,d>

<s,e,(**ldf** c_f)·c,d> → <⟨c_f e⟩·s,e,c,d>

<a·⟨c_f e_f⟩·s,e,**app**·c,d> → <[],a·e_f,c_f,<s,e,c>·d>

<a·s',e',**ret**·c',<s,e,c>·d> → <a·s,e,c,d>

Init: Code → <[],[],Code,[]>

final state <R·[],[],[],[]>

Decompilation of values

Decompilation of a list of SECD values is the list of all decompiled values

$$[\![[v_0\ v_1\ \ldots v_N]]\!] \rightarrow [[\![v_0]\!]\ [\![v_1]\!]\ \ldots [\![v_N]\!]]$$

Decompilation of a value (a closure) is an abstraction where the body is the decompilation of the code in a <s,e,c> structure. But the stack and the environment are now list of terms

$$[\![\langle c\ e\rangle]\!] \rightarrow \lambda x_i.[\![<[],x_i\cdot[\![e]\!],c\rangle]\!] \qquad i=\text{size of } e$$

$$[\![<\text{Sequence<Term>},\text{Sequence<Term>},\text{Code}\rangle]\!] \rightarrow \text{Term}$$

Decompilation of code

$[\![\langle s,e,(\textbf{ld}\ i)\cdot c\rangle]\!] \rightarrow [\![\langle (\text{get}\ e\ i)\cdot s,e,c\rangle]\!]$

$[\![\langle a\cdot f\cdot s,e,\textbf{app}\cdot c\rangle]\!] \rightarrow [\![\langle (f\ a)\cdot s,e,c\rangle]\!]$

$[\![\langle s,e,(\textbf{ldf}\ c_f)\cdot c\rangle]\!] \rightarrow [\![\langle \lambda x_i.[\![\langle [],x_i\cdot e,c_f\rangle]\!]\cdot s,e,c\rangle]\!]$

$[\![\langle a\cdot s,e,\textbf{ret}\cdot c\rangle]\!] \rightarrow a$

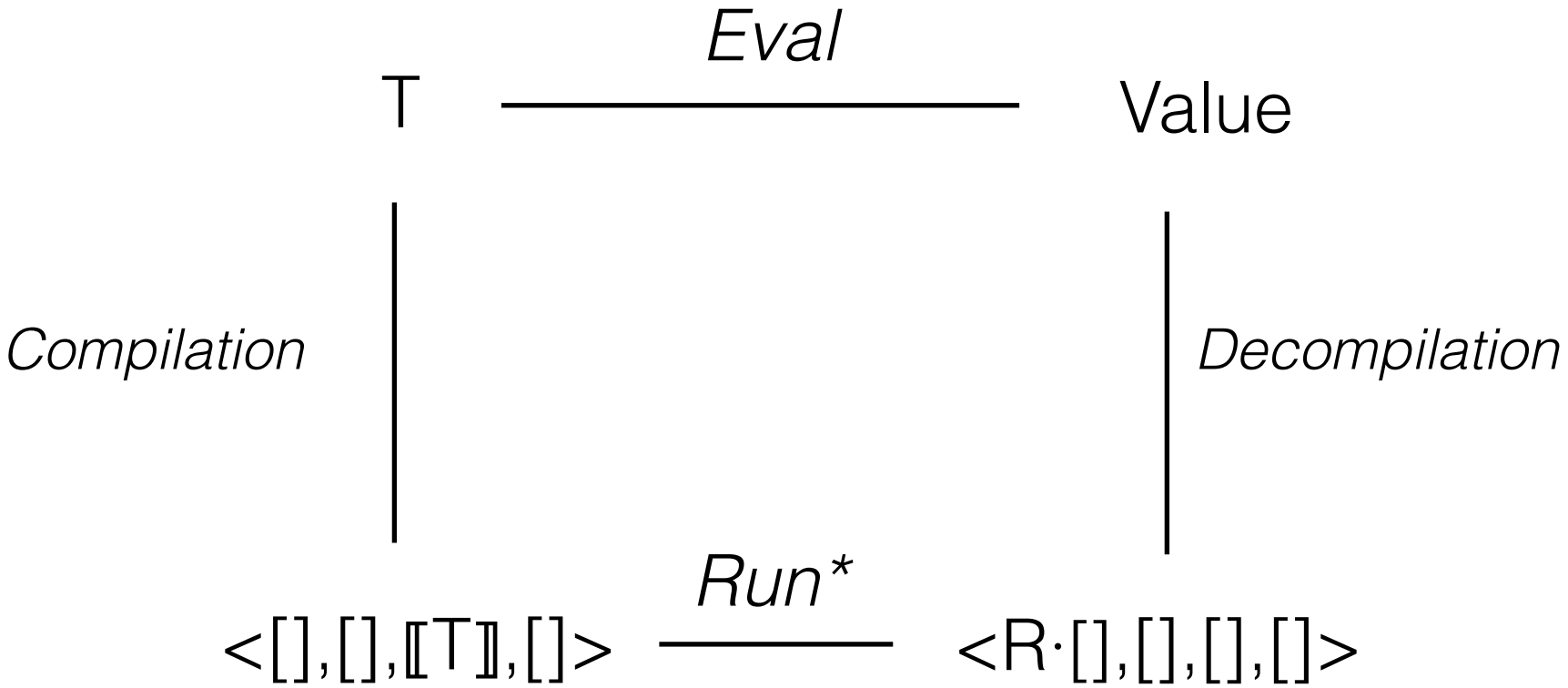
Decompilation of frames

Machine -> Term

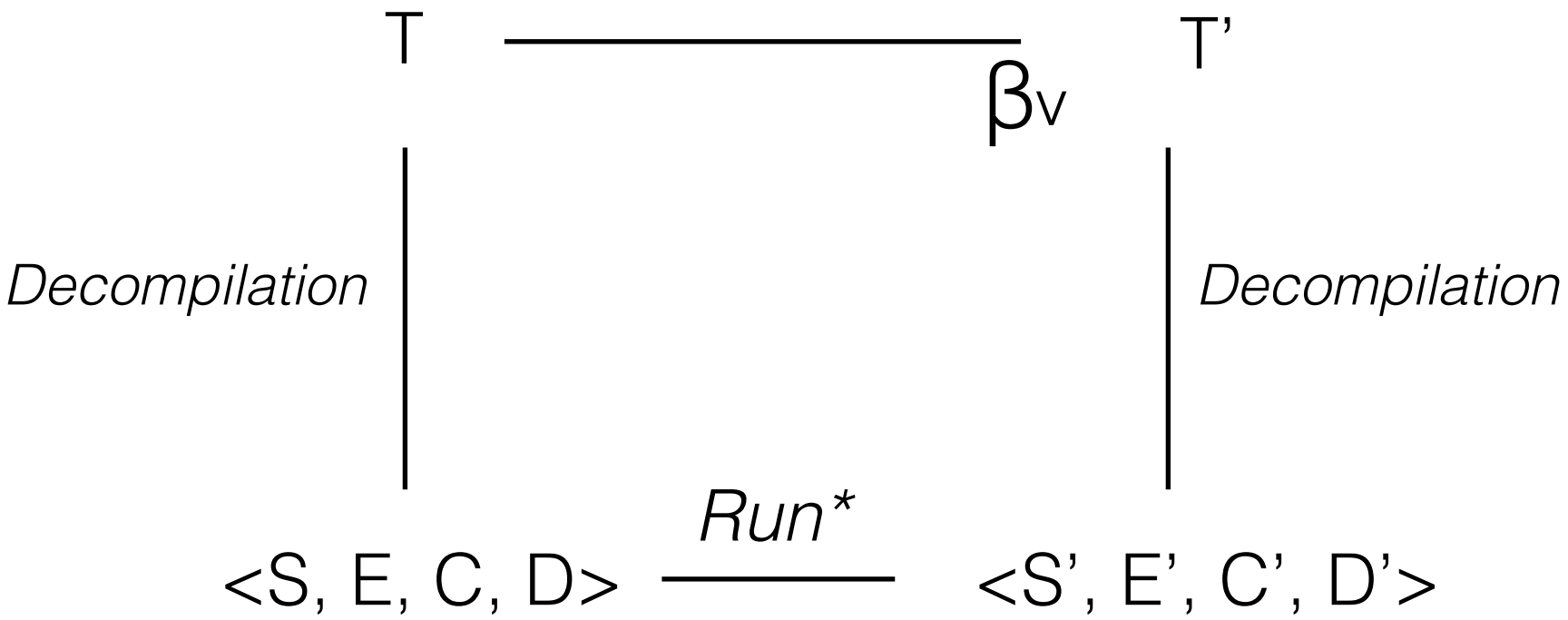
$$[\![\langle s,e,c,[]\rangle]\!] \rightarrow [\![\langle [\![s]\!],[\![e]\!],c\rangle]\!]$$

$$[\![\langle s_1,e_1,c_1,\langle s_2,e_2,c_2\rangle \cdot d\rangle]\!] \rightarrow$$
$$[\![\langle [\![\langle [\![s_1]\!],[\![e_1]\!],c_1\rangle]\!] \cdot s_2,e_2,c_2,d\rangle]\!]$$

Property



Step property



terminal recursion

Execution of Omega blows up the stack

$\langle a \cdot \langle c_f\ e_f \rangle \cdot s, e, \mathbf{app} \cdot \mathbf{ret} \cdot c, d \rangle \rightarrow \langle s, a \cdot e_f, c_f, d \rangle$

By construction of the compilation
s is nil