In the name of God

PL homework #4

Seyed Mohammad Mehdi Ahmadpanah – 9031806

from : Types and Programming Languages - Pierce

5.3.6)

For beta-reduction :

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

it is like those rules , but in judgments , we have term t_1 and t_2 , not value v_1 and v_2 . (because it isn't necessary that be value.)

For normal-order : (left-most and outermost)

$$\frac{na_1 \longrightarrow na_1'}{na_1\ t_2 \longrightarrow na_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{nanf_1\ t_2 \longrightarrow nanf_1\ t_2'} \qquad \text{(E-App2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\lambda x.t_1 \longrightarrow \lambda x.t_1'} \qquad \text{(E-Abs)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

na ::= x | t_1 t_2 (for non-abstractions)
nf ::= λx . nf | nanf (for normal forms)
nanf ::= x | nanf nf (for non-abstraction normal form)

For lazy evaluation (or call-by-value) :

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12} \qquad \text{(E-AppAbs)}$$

because this strategy is substitution name and at last evaluate names and this strategy doesn't allow to reduction in abstraction .

5.3.7)
Evaluation of expressions in the λNB may reach wrong states in the following 3 ways :
1 .
the use of λ expressions as arguments in BN . we define :
lambdaterm ::= λx . t
and add lambdaterm to the existing badbool and badnat . the existing evaluation rules will now catch instances of lambda expressions being used as arguments to terms excepting either boolean or natural numbers.
2.
an attempt to use either boolean or numerical terms as an abstraction. we define :
badabstraction ::= true | false | nv | wrong
and include the evaluation rule :
badabstraction t → wrong
3.
The use of a term deemed wrong as an argument to an abstraction. we add the evaluation rule:
t_1 wrong → wrong

5.3.8)

$$\lambda x.t \Downarrow \lambda x.t$$

$$\frac{t_1 \Downarrow \lambda x.t_{12} \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_{12} \Downarrow t'}{t_1\ t_2 \Downarrow t'}$$

big-step evaluation rule :

$$v \Downarrow v \qquad \text{(B-VALUE)}$$

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{t_1 \Downarrow nv_1}{\texttt{succ } t_1 \Downarrow \texttt{succ } nv_1} \qquad \text{(B-SUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{pred } t_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{pred } t_1 \Downarrow nv_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{iszero } t_1 \Downarrow \texttt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{iszero } t_1 \Downarrow \texttt{false}} \qquad \text{(B-ISZEROSUCC)}$$

from : Concepts in Programming Languages – Mitchell

4.4)

a. $((\lambda f . \lambda g . f (g\ 1)) (\lambda x . x+4)) (\lambda y . 3-y) \rightarrow (\lambda g . (\lambda x . x+4) (g\ 1)) (\lambda y . 3-y) \rightarrow$

$(\lambda x . x+4) ((\lambda y . 3-y)\ 1) \rightarrow (((\lambda y . 3-y)\ 1) + 4) \rightarrow 6$

b. $((\lambda f . \lambda g . f (g\ 1)) (\lambda x . x+4)) (\lambda y . 3-y) \rightarrow (\lambda g . (\lambda x . x+4) (g\ 1)) (\lambda y . 3-y) \rightarrow$

$(\lambda x . x+4) ((\lambda y . 3-y)\ 1) \rightarrow (\lambda x . x+4)\ 2 \rightarrow 6$

4.5)

$(\lambda compose . (\lambda h . compose\ h\ h\ 3) (\lambda x . x + x)) (\lambda f . \lambda g . \lambda x . f (g\ x)) \rightarrow$

$(\lambda h . (\lambda f . \lambda g . \lambda x . f (g\ x)) h\ h\ 3) (\lambda x . x + x) \rightarrow$

$(\lambda f . \lambda g . \lambda x . f (g\ x)) (\lambda x . x + x) (\lambda x . x + x)\ 3 \rightarrow$

$(\lambda g . \lambda x . (\lambda x . x + x) (g\ x)) (\lambda x . x + x)\ 3 \rightarrow$

$(\lambda x . (\lambda x . x + x) ((\lambda x . x + x)\ x))\ 3 \rightarrow$

$(\lambda x . (\lambda x . x + x) (x + x))\ 3 \rightarrow$

$(\lambda x . ((x + x) + (x + x)))\ 3 \rightarrow$

$((3 + 3) + (3 + 3)) \rightarrow 12$

in line 3 of reduction , we have $(\lambda f . \lambda g . \lambda x . f (g\ x)) (\lambda x . x + x) (\lambda x . x + x)\ 3$ (with substitute compose as $\lambda f . \lambda g . \lambda x . f (g\ x)$ and h as $(\lambda x . x + x)$ in "compose h h 3" , we can reduce expression faster) , means that x applies on g and the result applies on f (or fog(x)) .

4.6)

f ::= $\lambda g.\ g\ g$

main ::= f f = $(\lambda g.\ g\ g) (\lambda g.\ g\ g) \rightarrow (\lambda g.\ g\ g) (\lambda g.\ g\ g) \rightarrow (\lambda g.\ g\ g) (\lambda g.\ g\ g) \rightarrow \ldots$

This reduction will not finish!

4.8)

a. C[[x := 1; x := x +1]](s_0) = C [[x:=x+1]] (C [[x:=1]] (s_0))= C [[x:=x+1]] (s_1) = s_2

s_0 = { (x,0) }

s_1 = { (x,1) }

s_2 = { (x,2) }

b. C[[x := 1; x := x +1]](s) = C [[x:=x+1]] (C [[x:=1]] (s))= C [[x:=x+1]] (s_1) = s_2

s = { (x,uninit) }

s_1 = { (x,1) }

s_2 = { (x,2) }

because x initialized at s , so the value of x in s isn't effective in calculation.

4.9)

a. C[[x := 0; y := 0; if x = y then z := 0 else w := 1]](s_0)

= C[[y := 0; if x = y then z := 0 else w := 1]] (C[[x := 0]] (s_0))

= C[[if x = y then z := 0 else w := 1]] (C[[y := 0]](s_1))

= C[[if x = y then z := 0 else w := 1]] (s_2)

= if E[[x=y]](s2) = error or c[[z := 0]](s) = error or c[[w:=1]](s) = error then error

else c[[z := 0]](s_2) + c[[w:=1]](s_2)

= modify(s_0 , x , init) and modify(s_0 , y , init)

z and w is uninit (because both of them aren't initialized in "then" and "else" expression)

s_0 = {(x,uninit),(y, uninit),(z, uninit),(w, uninit)}

s_1 = {(x,0),(y, uninit),(z, uninit),(w, uninit)}

s_2 = {(x,0),(y,0),(z, uninit),(w, uninit)}

b. C[[if $x = y$ then $z := y$ else $z := w$]](s)

= if E[[x=y]](s) = error or c[[z := 0]](s) = error or c[[w:=1]](s) = error then error

else c[[z := 0]](s) + c[[w:=1]](s)

= modify(s , x , init) and modify(s , y , init)

z and w is uninit (because both of them aren't initialized in "then" and "else" expression) s = {(x,init) , (y,init) , (z,uninit) , (w,uninit)}

4.11)

a. yes . when evaluating of e_1 takes long time and g is waiting for evaluate “if (e_1=0)” .

b. because in evaluation of g , “if (e_1 = 0)” is true , and g will return “1” . (without consider e_2 , because it’s lazy evaluation. evaluation of g willn’t arrive to evaluate e_2)

c. we can evaluate e_1 and e_2 in parallel and then evaluate g or evaluate if-else if-else parts of g in parallel (means that evaluate “if” section and “else if” section and “else” section in parallel)

d. no . we can’t evaluate g in parallel in this conditions. because in parallel evaluation , maybe we will have two different result for one expression at different time evaluations.

4.13)

a.

به دلیل وجود assignment در imperative و داشتن side effect نمی توان ارزیابی موازی داشت و به این دلیل که در ذات زبانهای functional است ، این گونه زبانها به درد سیستم های بزرگ موازی (همروند) می خورد.

b.

زبانهای functional سربار اضافه بیشتری به سیستم تحمیل می کند که در مباحث تئوری مطرح نمی شود. در مباحث تئوری به ذخیره شدن state ها در imperative اشاره می شود.

بعضی از ساختمان داده ها با زبان های functional راحت تر کار می کند در حالیکه همان ساختمان داده با زبان imperative کارهای دیگری باید انجام شود.heavy allocation در زبانهای functional ذاتی است

همه این موارد باعث می شود تا حجم بیشتری حافظه توسط زبانهای functional مصرف شود.

instructions were very simple, which made hardware implementation easier

c.

باتوجه به فرضیات مطرح شده ، زبانهای functional به executable های بزرگتری بعد از کامپایل نیاز دارند ولی زبانهای imperative به دلیل این که تنها قابلیت assignment و ذخیره state ها را دارند . اما در کاربردهای امروزی باتوجه به این

که زبانهای imperative هم قابلیت تعریف higher-order function و هم garbage collector را دارند به executable های بزرگتری نیاز دارند.

.d

این مسائل باعث شد تا در انتخاب بین functional و imperative بیشتر به imperative توجه شود و در موارد خاص (مثل محاسبات موازی) به functional روی بیاورند .

e. هنوز هم بعضی از افراد به functional عقیده دارند و اعتقاد دارند که همه برنامه ها را می توان راحت تر در functional نوشت و عده دیگری به زبان های imperative اعتقاد دارند .