Programming Languages Hw2

Arya Banaeizadeh 9431029

Benjamin Pierce, Types and Programming Languages

3.5.10 Exercise [*]: Rephrase Definition 3.5.9 as a set of inference rules

$$\frac{\mathbf{t \to t'}}{\mathbf{t \to {}^{*}t'}} \qquad\qquad \frac{}{\mathbf{t \to {}^{*}t'}} \qquad\qquad \frac{\mathbf{t \to {}^{*}t' ,\ t' \to {}^{*}t''}}{\mathbf{t \to {}^{*}t''}}$$

3.5.17 Exercise [*]: Rephrase Definition 3.5.9 as a set of inference rules

v ⇓ v == v -> v

t1 ⇓ true t2 ⇓ v2 == t1-> true , t2-> v2

____

if t1 then t2 else t3 ⇓ v2 == if t1 then t2 else t3 -> if true then t2 else t3 -> **t2 -> v2**
[t2 -> * v2]

t1 ⇓ false t3 ⇓ v3 == t1 -> false , t3 -> v3

____

if t1 then t2 else t3 ⇓ v3 == if t1 then t2 else t3 -> if false then t2 else t3 -> **t3 -> v3**
[t3 -> * v3]

t1 ⇓ nv1 == t1 -> nv1

____

succ t1 ⇓ succ nv1 == succ t1 -> succ nv1

t1 ⇓ 0 == t1 -> 0

____

pred t1 ⇓ 0 == pred t1 -> 0

t1 ⇓ succ nv1 == t1 -> succ nv1

____

pred t1 ⇓ nv1 == pred t1 -> nv1

t1 ⇓ 0 == t1 -> 0

____

iszero t1 ⇓ true == iszero t1 -> true

t1 ⇓ succ nv1 == t1 -> succ nv1

____

iszero t1 ⇓ false == iszero t1 -> false

therefore we have

t ⇓ v => t -> v

t -> v => t -> *v

t ⇓ v => t -> *v

t -> *v if t ⇓ v

and:

t -> v => t ⇓ v

t -> *v => t -> v

t ⇓ vt if t-> *v

3.5.18 Suppose we want to change the evaluation strategy of our language so that the then and else branches of an if expression are evaluated (in that order) before the guard is evaluated. Show how the evaluation rules need to change to achieve this effect.

t2 -> t2' , t3 -> t3'

____

if t1 then t2 else t3 -> if t1 then t2' else t3'

this ensures that t2 and t3 are evaluated before t1

t2, t3 are terminals , t1 → t1'

____

If t1 then t2 else t3 → If t1' then t2 else t3

t2, t3 are terminals

____

If true then t2 else t3 -> t2

t2, t3 are terminals

____

If false then t2 else t3 -> t3

These ensure the outer if is evaluated before the inner if.

5.2.2 Find another way to define the successor function on Church numerals.

Church's successor function:

Suc = λn.λs.λz.(s)(n s z)

Custom successor function:

custom = λn.λs.λz.n s (s z)

5.2.3 Is it possible to define multiplication on Church numerals without using plus ?

Yes:

λa.λb.λs.λz.a(b s)z

5.2.4 Define a term for raising one number to the power of another.

λa.λb.(a (times b) c1) indicates b^a

5.2.5 Use prd to define a subtraction function

λa.λb.(b prd a) indicates a - b

5.2.7 Write a function Equal that tests two numbers for equality and returns a Church Boolean

equal = λx.λy.and (iszro (x prd y)) (iszro (y prd x))

5.2.8 A list can be represented in the lambda-calculus by its fold function. (OCaml's name for this function is fold_left ; it is also sometimes called reduce .) For example, the list [x,y,z] becomes a function that takes two arguments c and n and returns c x (c y (c z n))) . What would the representation of nil be? Write a function cons that takes an element h and a list (that is, a fold function) t and returns a similar representation of the list formed by prepending h to t . Write isnil and head functions, each taking a list parameter. Finally, write a tail function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define prd for numbers).

Nil would be: pair tru tru

Function cons = λh.λt.pair fls (pair h t)

Isnil = fst

Head = λh.fst(snd h)

Tail = λt.snd(snd t)

5.2.10 Define a function churchnat that converts a primitive natural number into the corresponding Church numeral.

ch = λa.λb.if iszero x then c0 else scc (b (pred a))

churchnat = fix ch

5.2.11 Use fix and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals

sl = λm. λn. test (isnil n) (λx . c0) (λx . (plus (head n) (m (tail n)))) c0

sumlist = fix sl

John Mitchell, Concepts in Programming Languages

The Algol-like program fragment can be written as the following lambda expression. Reduce the expression to a normal form in two different ways, as described below

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the left as possible.

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

$(\lambda g.(\lambda x.x+4)(g\ 1))(\lambda y.3-y)$ ->

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

$((\lambda y.3-y)1)+4$ ->

$3-1+4$ ->

6

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the right as possible.

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

$(\lambda g.(\lambda x.x+4)(g\ 1))(\lambda y.3-y)$ ->

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

$(\lambda x.x+4)(3-1)$ ->

$2+4$ ->

6

4.5 Here is a “sugared” lambda expression that uses let declarations: let compose = λf. λg. λx. f(g x) in let h = λx. x + x in compose h h 3 The “desugared” lambda expression, obtained when each let z = U in V is replaced with (λz. V)U is (λcompose. (λh. compose h h 3) λx. x + x) λf. λg. λx. f(g x). This is written with the same variable names as those of the let form to make it easier to read the expression. Simplify the desugared lambda expression by using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

(λcompose.(λh.compose h h 3)(λx.x+x))(λf.λg.λx. f(g x)) ->

(λh.(λf.λg.λx.f(g x))h h 3)(λx.x+x) ->

(λf.λg.λx.f(g x))(λx.x+x)(λx.x+x)3 ->

(λg.λx.(λx.x+x)(g x))(λx.x+x)3 ->

(λx.(λx.x+x)((λx.x+x)x))3 ->

(λx.(λx.x+x)(x + x))3 ->

(λx.((x+x)+(x + x)))3 ->

((3+3)+(3+3)) ->

12

4.6 A programmer is having difficulty debugging the following C program. In theory, on an “ideal” machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, as extra space is required every time a function call is made.) int f(int (*g)(...)){ /* g points to a function that returns an int */ return g(g); } int main(){ int x; x = f(f); printf(”Value of x = %d\n”, x); return 0; } Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application f(f). This program assumes that the type checker does not check the types of arguments to functions.

F = λ.ff

F(F): (λf. ff)(λg. gg) → (λg. gg)(λg. gg) → (λg. gg)(λg. gg)

This function result in a term and will run indefinitely.

4.8 The text describes a denotational semantics for the simple imperative language given by the grammar P ::= x := e | P1;P2 | if e then P1 else P2 |while e do P. Each program denotes a function from states to states, in which a state is afunction from variables to values.

(a) Calculate the meaning C[[x := 1; x := x + 1;]](s0) in approximately the same detail as that of the examples given in the text, where s0 = λv ∈ variables. 0, giving every variable the value 0.

C[[x := 1; x := x +1]](s0) =

C [[x:=x+1]] (C [[x:=1]] (s0))=

C [[x:=x+1]] (s1) =

s2

s0 = { (x,0) }, s1 = { (x,1) } , s2 = { (x,2) }

(b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why C[[x := 1; x := x + 1;]](s) = C[[x := 2;]](s) for every state s.

C[[x := 1; x := x +1]](s) =

C [[x:=x+1]] (C [[x:=1]] (s))=

C [[x:=x+1]] (s1) =

s2

s = { (x,uninit) } , s1 = { (x,1) } , s2 = { (x,2) }

x doesn't affect the result since it is initialized at s.

4.9 A nonstandard denotational semantics describing initialize-before-use analysis is presented in the text.

(a) What is the meaning of C[[x := 0; y := 0; if x = y then z := 0 else w := 1]](s0) in the state s0 = λy ∈ variables.uninit? Show how to calculate your answer.

C[[x := 0; y := 0; if x = y then z := 0 else w := 1]](s0) =

C[[y := 0; if x = y then z := 0 else w := 1]] (C[[x := 0]](s0)) =

C[[if x = y then z := 0 else w := 1]] (C[[y := 0]](s1)) =

C[[if x = y then z := 0 else w := 1]] (s2) =

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

else c[[z := 0]](s2) + c[[w:=1]](s2) = modify(s0 , x , init) and modify(s0 , y , init)

s0 = {(x,uninit),(y, uninit),(z, uninit),(w, uninit)} , s1 = {(x,0),(y, uninit),(z, uninit),(w, uninit)} , s2 = {(x,0),(y,0),(z, uninit),(w, uninit)}

(c) Calculate the meaning C[[if x = y then z := y else z := w]](s) in state s with s(x) = init, s(y) = init, and s(v) = uninit or every other variable v.

C[[if x = y then z ≔ y else z ≔ w]]($s0$) =

if E[[x == y]]($s0$) $then$ C[[$z \coloneqq y$]]($s0$) $else$ C[[$z \coloneqq w$]]($s0$) =

if E[[x == y]]($s0$) $then$ $modify$($s0$,z,y)$else$ $modify$($s0$,z,w)

4.10

(a) Show how to calculate the meaning of the expression if false then 0 else 1 in the environment η0 = λy ∈ Var. type error. Exercises 87

(b) Suppose e1 and e2 are expressions with V[[e1]]η = integer and V[[e2]]η = boolean in every environment. Show how to calculate the meaning of the expression let x :int=e1 in (if e2 then e1 else x) in environment η0 = λy ∈ Var. type error.

(c) In declaration let x :τ=e in e, the type of x is given explicitly. It is also possible to leave the type out of the declaration and infer it from context. Write a semantic clause for the alternative form, let x=e1 in e2 by using the following partial solution (i.e., fill in the missing parts of this definition):

4.11

(a) Assume we evaluate g(e1, e2) by starting to evaluate g, e1, and e2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

Yes, when evaluating e1 is pending and g is waiting for evaluation of if (e1 = 0)

(d) Now, suppose the value of e1 is zero and evaluation of e2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression g(e1, e2) will terminate in error. What will happen with lazy evaluation? Parallel evaluation?

In evaluating g, 1 is returned

(e) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate g(e1, e2) by starting g, e1, and e2 in parallel, if the value of e1 is zero and evaluation of e2 terminates in an error?

We can evaluate e1 and e2 simultaneously and then evaluate g

(f) Suppose now that the language contains side effects. What if e1 is z and e2 contains an assignment to z; can you still evaluate the arguments of g(e1, e2) in parallel? How? Or why not?

No, since the expression evaluation may vary in different times.

4.12

(a) Explain how you might execute parts of the sample program x = 5; y = f(g(x),h(x)); if y==5 then z=g(x) else z=h(x); in parallel. More specifically, assume that your implementation will schedule the following processes in some way: process 1 - set x to 5 process 2 - call g(x) process 3 - call h(x) process 4 - call f(g(x),h(x)) and set y to this value process 5 - test y==5 process 6 - call g(x) and then set z=g(x) process 7 - call h(x) and then set z=h(x) For each process, list the processes that this process must wait for and list the processes that can be executed in parallel with it. For simplicity, assume that a call cannot be executed until the parameters have been evaluated and assume that processes 6 and 7 are not divided into smaller processes that execute the calls but do not assign to z. Assume that parameter passing in the example code is by value.

Waiting for process | can be called simultaneously

Process 1: nothing | all other processes

Process 2: process 1,4 | processes 3 ,5 , 6, 7

Process 3: 4 | 2, 5, 6, 7

Process 4: 2,3,5,1 | 6,7

Process 5: 4 | 1,2,3,6,7

Process 6: 1,5| 2,3,4,7

Process 7: 1,5 | 2,3,4,6

(b) If you further divide process 6 into two processes, one that calls g(x) and one that assigns to z, and similarly divide process 7 into two processes, can you execute the calls g(x) and h(x) in parallel? Could your compiler correctly eliminate these calls from processes 6 and 7? Explain briefly.

No. it is possible that z is set to g(x) before g(x) is initialized and cannot be assigned again since we have to satisfy the single assignment condition. We also can say this for h(x).

(c) Would the parallel execution of processes you describe in parts (a) and (b), if any, be correct if the program does not satisfy the single-assignment condition? Explain briefly.

Yes as explained we can do this.

(d) Is the single-assignment condition decidable? Specifically, given an program written in asubset of C, for concreteness, is it possible for acompiler to decide whether this program satisfies the single-assignment condition? Explain why or why not. If not, can you think of a decidable condition that implies the single-assignment condition and allows many useful single-assignment programs to be recognized?

Yes, as explained in the section if the compiler finds out a double assignment in the program body or a while or for loop without return In the end it is recognizable.

(e) Suppose a single-assignment language has no side-effect operations other than assignment. Does this language pass the declarative language test? Explain why or why not.

Yes. Assigning a value to a term for a once is equivalent to declaring that constant.

4.13

(a) Are there inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks? Why?

Imperative languages are easier to use for programmers and probably because they run faster. On the other hand functional programs are mostly used for parallel programming.

(b) Which variety (imperative or functional) is easier to implement on machines with limited disk and memory sizes? Why?

Imperative languages would be better in this case since functional languages often have more overhead considering their use of heavy allocation, garbage collection etc.

(c) Which variety (imperative or functional) would require bigger executables when compiled? Why?

Functional programs require bigger executables when compiled as said in the previous section more resources are needed. A major reason would be them being garbage collected and being able to define higher order functions.

(d) What consequence might these facts have had in the early days of computing?

When making an efficient use of memory was an issue, a diversion to imperative languages would have occurred. However, under special circumstances, say parallel programming functional programming had a higher priority.

(g) Are these concerns still valid today?

It may be less of an issue since we can be more generous with resources.

4.14 It can be difficult to write programs that run on several processors concurrently because a task must be decomposed into independent subtasks and the results of subtasks often must be combined at certain points in the computation. Over the past 20 years, many researchers have tried to develop programming languages that would make it easier to write concurrent programs. In his Turing Lecture, Backus advocated functional programming because he believed functional programs would be easier to reason about and because he believed that functional programs could be executed efficiently in parallel. Explain why functional programming languages do not provide a complete solution to the problem of writing programs that can be executed efficiently in parallel. Include two specific reasons in your answer.

Some of the functional programming languages are not designed to be used for parallel programming and therefore could not be used in that way

A higher rate of threads may result in more overhead and thus less efficiency

Some functional programs may implemented functionally, for example they may be expression based syntax wise But implemented with assignments

Some functional languages use assignments and are not fully functional