Written Assignment 2

CS 538, Spring 2020

February 10, 2020

We will work with the lambda calculus with booleans. The syntax of types, values, expressions, and contexts are defined as follows.

$$
\begin{aligned}
t &::= \mathit{Bool} \mid t_1 \to t_2 \\
v &::= x \mid \lambda x : t.\ e \mid \mathit{true} \mid \mathit{false} \\
e &::= x \mid \lambda x : t.\ e \mid e_1\ e_2 \mid \mathit{true} \mid \mathit{false} \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \\
G &::= \cdot \mid G, x : t
\end{aligned}
$$

Note that we have made a small tweak in the syntax for functions (lambdas): each function includes a *type annotation* describing the type of its input. Also note that we do not have the fixed point construct *fix* $f.\ e$; do not use this construct in any of your answers. The typing rules are given as follows.

$$\frac{G(x) = t}{G \vdash x : t} \qquad \frac{G, x : t_1 \vdash e : t_2}{G \vdash \lambda x : t_1.\ e : t_1 \to t_2} \qquad \frac{G \vdash e_1 : t_1 \to t_2 \qquad G \vdash e_2 : t_1}{G \vdash e_1\ e_2 : t_2}$$

$$\frac{}{G \vdash \mathit{true} : \mathit{Bool}} \qquad \frac{}{G \vdash \mathit{false} : \mathit{Bool}} \qquad \frac{G \vdash e_1 : \mathit{Bool} \qquad G \vdash e_2 : t \qquad G \vdash e_3 : t}{G \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : t}$$

Throughout this assignment, we will work with a call-by-value operational semantics, i.e., we use the small-step rules:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{(\lambda x : t.\ e_1)\ e_2 \to (\lambda x : t.\ e_1)\ e_2'} \qquad \frac{}{(\lambda x : t.\ e_1)\ v \to e_1[x \mapsto v]}$$

1 Types, Terms, and Contexts (5)

1.1 Fill in the Types

Find a context G and a type t that makes the given program well-typed. For example, for $G \vdash \lambda x : \mathit{Bool}.y : t$, you could find the context $G = y : \mathit{Bool}$ and the type $t = \mathit{Bool} \to \mathit{Bool}$ for the entire expression. Or, you could take $G = y : (\mathit{Bool} \to \mathit{Bool})$ and $t = \mathit{Bool} \to (\mathit{Bool} \to \mathit{Bool})$.

Remember: all free variables in a program must be listed in the context, along with their types. (But variables in the context do not all need to appear in the program.)

(a) $G \vdash \mathit{if}\ x\ \mathit{then}\ x\ \mathit{else}\ y : t$

(b) $G \vdash x\ y : t$

(c) $G \vdash \lambda x : \mathit{Bool}.\ x : t$

(d) $G \vdash \lambda x : \mathit{Bool} \to \mathit{Bool}.\ y\ x\ \mathit{true} : t$

(e) $G \vdash \lambda x : \mathit{Bool} \to \mathit{Bool}.\ y\ (x\ \mathit{true}) : t$

(f) $G \vdash \lambda x : \mathit{Bool}.\ \mathit{if}\ y\ \mathit{then}\ x\ \mathit{else}\ y : t$

(g) $G \vdash \lambda x : \mathit{Bool}.\ \lambda y : \mathit{Bool}.\ \lambda z : \mathit{Bool}.\ \mathit{if}\ z\ \mathit{then}\ x\ \mathit{else}\ y : t$

1.2 Fill in the Terms

Find a program e that has the indicated type under the given context. (Note: $\cdot$ is the empty context.)

(a) $\cdot \vdash e : \mathit{Bool} \to \mathit{Bool}$

(b) $\cdot \vdash e : \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$

(c) $\cdot \vdash e : (\mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool}$

(d) $\cdot \vdash e : (\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool}$

(e) $x : \mathit{Bool}, y : \mathit{Bool} \to (\mathit{Bool} \to \mathit{Bool}) \vdash e : \mathit{Bool} \to \mathit{Bool}$

2 Inhabitants of a Type (10)

2.1 How Many Programs?

(a) There are infinitely many closed programs of type *Bool*, but how many different closed *values* are there of type *Bool*? List them out.

(b) How many different closed values are there of type $\mathit{Bool} \to \mathit{Bool}$? List them out. For the purposes of this exercise, consider two functions f, g to be equal if $f\ x$ and $g\ x$ reduce to equal Booleans for $x \in \{\mathit{true}, \mathit{false}\}$

(c) More generally, suppose that there are m different values of type X and n different values of type Y. How many different closed values are there of type $X \to Y$? You don't need to list them out. (The function type $X \to Y$ is sometimes written Y^X.)

2.2 Programs as Proofs

Suppose we add a few more types to our lambda calculus.

$$t ::= \cdots \mid P \mid Q \mid R \mid t_1 \times t_2 \mid t_1 + t_2$$

where $\times$ and $+$ are the product and sum types we saw in class, and we have corresponding constructors

$$e ::= \cdots \mid (e_1, e_2) \mid \mathit{left}(e) \mid \mathit{right}(e)$$

and destructors:

$$e ::= \cdots \mid \mathit{fst}(e) \mid \mathit{snd}(e) \mid \mathit{case}\ e\ \mathit{of}\ \{\mathit{left}(x) \to e_l; \mathit{right}(y) \to e_r\}$$

Do *not* assume that there are programs of type P, Q, or R—our language has no constants for these types, like *true*, *false* for the booleans. Find a closed program e that has the indicated type under the given context.

(a) $\cdot \vdash e : P \to P$

(b) $\cdot \vdash e : P \times Q \to P$

(c) $\cdot \vdash e : P \times Q \to Q \times P$

(d) $\cdot \vdash e : P \to (P + Q)$

(e) $\cdot \vdash e : P \times (P \to Q) \to Q$

(f) $\cdot \vdash e : P \to (Q \to P)$

(g) $\cdot \vdash e : (P \to Q) \times (Q \to R) \to (P \to R)$

Are there programs of the following types?

(a) $\cdot \vdash e : P \to Q$

(b) $\cdot \vdash e : (P + Q) \to P$

(c) $\cdot \vdash e : (P \to Q) \to P$

To get a hint of what is going on here, think of + as "or", × as "and", and → as "implies". What is common across the first group? What about the second group? (We are not looking for a very formal answer here.)

3 Adding Triples (15)

To get some practice with modeling types, we are going to extend the lambda calculus with booleans with types for triples. Recall that we need to provide three ingredients to model any new type: (1) we need to describe the grammar of the new type, (2) we need to extend the program grammar with things to construct/introduce programs of the new type, and (3) we need to extend the program grammar with things to eliminate/use/take apart/destruct programs of the new type.

There are (at least) two typical ways to incorporate products into a language, differing in how to eliminate/destruct products: via *projections*, or via *pattern matching/case analysis*. These choices are not mutually exclusive—modern functional languages like Haskell support both operations. To gain a better idea how these features work, you will model both of them for triples.

The first part is easiest: we add a new type for triples:

$$t ::= \cdots \mid (t_1, t_2, t_3)$$

You will do the second and third parts.

3.1 Projections

(a) Extend the program grammar to give a way to construct triples, and also project out from triples. You can pick any reasonable syntax, e.g., (e_1, e_2, e_3) and $\mathit{fst}(e)$, $\mathit{snd}(e)$, $\mathit{thd}(e)$.

(b) Provide typing rules for each of your new constructs. Hint: there should be four new typing rules in all: one for the constructor, and one for each of the three projections.

(c) Provide a small-step operational semantics for your extensions under call-by-value (eager) evaluation: each component of a triple should be first evaluated all the way to a value before projecting out. Hint: you do this with nine rules: three for the triple, and two for each of the projections.

3.2 Pattern matching

A different kind of construct for taking triples apart is *pattern matching*.

(a) Extend the program grammar to eliminate triples by pattern matching. You will add one new construct to the language, which should allow the programmer to write (a) the expression we want to take apart, (b) three variable names, to refer to the three components of the triple, and (c) a body expression that can mention the variable names. Again, you can pick whatever reasonable syntax you like. (Note: take a look at the case analysis construct above. Your case construct will have just a single case, and there is nothing wrong with that.)

(b) Provide one typing rule for your destructor.

(c) Provide a small-step operational semantics for your extension, still under call-by-value (eager) evaluation. Hint: you can do this with five rules: three for the triple, and two for the pattern match.

As we have discussed in class, Haskell is not a CBV language so the triples that you have added don't behave exactly like the triples in Haskell (both under projection and under pattern matching). Open up GHCi and see if you find a program using triples in Haskell that does not behave like the eager triples you added. It is not hard to modify the operational semantics to model *lazy* triples, which do not evaluate all the way to values when eliminating. There are multiple kinds of lazy triples, which differ in how far they evaluate before eliminating.