CMSC22100 HW #4

Jaime Arana-Rochel

April 30, 2014

Problem 1 : Evaluation Semantics

TaPL 3.5.18.
New Evaluation Rules:

$$if\ true\ then\ v_1\ else\ v_2 \longrightarrow v_1\ (E-IFTRUE)$$

$$if\ false\ then\ v_1\ else\ v_2 \longrightarrow v_2\ (E-IFFALSE)$$

$$\frac{t_2 \longrightarrow t_2'}{if\ t_1\ then\ t_2\ else\ t_3\ \longrightarrow if\ t_1\ then\ t_2'\ else\ t_3}\ (E-IFTHEN)$$

$$\frac{t_3 \longrightarrow t_3'}{if\ t_1\ then\ v_1\ else\ t_3\ \longrightarrow if\ t_1\ then\ v_1\ else\ t_3'}\ (E-IFELSE)$$

$$\frac{t_1 \longrightarrow t_1'}{if\ t_1\ then\ v_1\ else\ v_2\ \longrightarrow if\ t_1'\ then\ v_1\ else\ v_2}\ (E-IF)$$

These new rules show that *E-IFTRUE* and *E-IFFALSE* can only be used as a rule when the *then* and *else* branches have already been evaluated. *E-IFTHEN* states that if t_2 evaluates to t_2', then t_2 is evaluated first in the conditional. *E-IFELSE* states that if t_3 evaluates to t_3', then t_3 is evaluated in the conditional, but only if t_2 is a value (aka evaluated). *E-IF* states that if t_1 evaluates to t_1', then t_1 is evaluated in the conditional, but only if t_2 and t_3 are values.

Problem 2: Proofs about Programming Languages

TaPL 8.2.3

Theorem 8.2.3. *Every subterm of a well-typed term is well typed*

Proof. By induction on the typing derivations. The term in the cases *T-ZERO*, *T-TRUE*, and *T-FALSE* are themselves well-typed terms without subterms, so the requirements of the theorem are satisfied.
Case: *T-IF* $\frac{t_1:Bool \;\; t_2:T \;\; t_3:T}{if\ t_1\ then\ t_2\ else\ t_3:T}$
It follows from the typing definitions and Lemma 8.2.2 that given a well-typed term $t =$ if t_1 then t_2 else t_3: T, then the subterms are themselves well typed (t_1 : Bool, t_2 : T, and t_3 : T)
Case: *T-SUCC* $\frac{t_1 : Nat}{succ\ t_1 : Nat}$
Like in case (T-IF), it follows from the typing definitions and Lemma 8.2.2 that given a well-typed term $t =$ succ t_1 : Nat, the subterm t_1 must itself be well-typed.
Cases: *T-PRED* and *T-ISZERO*
Similar □

TaPL 8.3.4

Theorem [Preservation]. *If t : T and t $\longrightarrow$ t', then t' : T*

Proof. By induction on evaluation derivations. It goes without saying that the theorem is vacuously satisfied for terms which are values since there are no evaluation rules for them.
Case: *E-IFTRUE* if true then t_2 else $t_3 \longrightarrow t_2$
We know from this rule that t has the form if true then t_2 else t_3 and that it evaluates to $t' = t_2$ ($t \longrightarrow t'$). There is only one typing derivation for this kind of term, *T-IF*, which means that t has type T, and it's subterms must have types t_1 : Bool, t_2 : T, and t_3 : T. Given that t evaluates to t_2 : T, we then see that t' must also have type T. The *E-IFFALSE* case proceeds similarly, so it is omitted.
Case: *E-IF* $\frac{t_1 \longrightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \longrightarrow if\ t_1'\ then\ t_2\ else\ t_3}$
This rule tells us that our term t has the form if t_1 then t_2 else t_3, and that it evaluates to $t' =$ if t_1' then t_2 else t_3. From the rule *T-IF*, we know that t_1 has type Bool, meaning that by induction t_1' also has type Bool. Using the typing derivation, we know t_2 : T, and t_3 : T for t, but this applies to

t' as well due to the evaluation. Know we can use the typing derivation to conclude that given t_1' : Bool, t_2 : T, and t_3: T, that t' : T.

Case: *E-SUCC* $\quad \frac{t_1 \longrightarrow t_1'}{succ\ t_1 \longrightarrow succ\ t_1'}$

This rule tells us that the term t (succ t_1) evaluates to t' (succ t_1'), given the evaluation of the subterm of t. From the typing derivation *T-SUCC*, we know that t_1 must be of type Nat, and t : Nat. So by induction t_1' must also be of type Nat. Therefore by the typing derivation, t' : Nat (t' : T). The cases for *E-PRED*, and *E-ISZERO* follow a similar argument.

Case: *E-PREDZERO* $\quad$ pred $0 \longrightarrow 0$

The rule tells us that t evaluates to $t' = 0$ which is a nv. From the typing derivation *T-ZERO* we know 0 : Nat, therefore by *T-PRED* t : Nat. From this we can conclude that t' has type Nat as well, so t' : T. The cases for *E-PREDSUCC*, *E-ISZEROZERO*, and *E-ISZEROSUCC* follow a similar argument.

$\square$

Problem 3: The Untyped Lambda Calculus

xor

The encoding for xor will be similar to the test abstraction in TaPL, except that if given two arguments (a b) then if a is fls then it should return b, otherwise it should return the logical negation (not) of b.

"The not encoding is the one used in the solution section of TaPL"

$$not = \lambda b.\ b\ fls\ tru$$

$$xor = \lambda a.\ \lambda b.\ a\ not(b)\ b$$

Rational Numbers

The strategy to encode rational numbers as a pair was discovered from the wikipedia article "Church Encoding"

Rational numbers may be encoded as a Church pair of two numbers. So the constructor is the same as the pair constructor found in TaPL, with the same fst and snd functions :

$$rational = \lambda num.\ \lambda dem.\ \lambda b.\ b\ num\ dem$$

$$fst = \lambda p.\ p\ tru$$

$$snd \;=\; \lambda p.\, p\; fls$$

The reciprocal operator *recip* merely switches the order of the elements in a pair.

$$recip \;=\; \lambda r.\, pair\; snd(r)\; fst(r)$$

Problem 4: Type Systems

The grammar for the types in the character and string grammar is as follows:

T ::= Char | String

Typing Rules for Terms:

$$nil \;:\; String \quad \text{(T-NIL)}$$

$$c \;:\; Char \quad \text{(T-C)}$$

$$\frac{c \;:\; Char \qquad t \;:\; String}{put(c\,,\, t) \;:\; String} \quad \text{(T-PUT)}$$

$$\frac{t \;:\; String}{get(t) \;:\; Char} \quad \text{(T-GET)}$$

$$\frac{t \;:\; String \qquad t \;:\; String}{cat(t\,,\, t) \;:\; String} \quad \text{(T-CAT)}$$

$$\frac{c \;:\; Char \qquad t \;:\; String}{del(c\,,\, t) \;:\; String} \quad \text{(T-DEL)}$$

$$\frac{t \;:\; String}{rev(t) \;:\; String} \quad \text{(T-REV)}$$