CS 520 Problem Set 11

Solutions by Jue Wang

December 17, 2004

Problem 1

1. Given that Bool $\triangleq$ Unit + Unit, we define
   true $\triangleq$ inl unit
   false $\triangleq$ inr unit

2. if t_1 then t_2 else $t_3 \triangleq$ case t_1 (λ_:Unit. t_2) (λ_:Unit. t_3)

3. We now verify our definitions.
   The term if true then t_1 else t_2 replaced with the definitions from above yields the term: case (inl unit) (λ_:Unit. t_1) (λ_:Unit. t_2) which evaluates to ((λ_:Unit. t_1) unit) by the (New-E-CaseInl) rule defined in Handout 24, page 7 and further evaluates to t_1 as desired by (E-App).

   The term if false then t_1 else t_2 replaced with the definitions from above yields the term: case (inr unit) (λ_:Unit. t_1) (λ_:Unit. t_2) which evaluates to ((λ_:Unit. t_2) unit) by the (New-E-CaseInr) rule defined in Handout 24, page 7 and further evaluates to t_2 as desired by (E-App).

Problem 2

We show how to treat variants as syntactic sugar using sums.

We define:

$< l_1 : T_1, l_2 : T_2, \ldots, l_{n-1} : T_{n-1}, l_n : T_n > \triangleq (T_1 + (T_2 + \ldots (T_{n-1} + T_n) \ldots))$

If $i < n$, $l_i^{<l_1:T_1,\ldots,l_n:T_n>}$ $t_i \triangleq$ (inr (inr ($\ldots$ (inl t_i) $\ldots$))) where inr occurs i - 1 times.

If $i = n$, $l_i^{<l_1:T_1,\ldots,l_n:T_n>}$ $t_i \triangleq$ (inr (inr ($\ldots$ (inr t_i) $\ldots$))) where inr occurs n - 1 times.

$case^{<l_1:T_1,\ldots,l_n:T_n>}$ t_0 $(\lambda t_1\ u_1)$ $(\lambda t_2\ u_2)$ $\ldots$ $(\lambda t_{n-1}\ u_{n-1})$ $(\lambda t_n\ u_n) \triangleq$
    case t_0 $(\lambda t_1\ u_1)$
        $(\lambda a_2$ (case a_2 $(\lambda t_2\ u_2)$
            $\vdots$
            $(\lambda a_{n-1}$ (case a_{n-1} $(\lambda t_{n-1}\ u_{n-1})$ $(\lambda t_n\ u_n)))$ $\ldots$)

Problem 3

We show how to treat the enumeration $(l_i, \ldots, l_n)$ as syntactic sugar for the variant $< l_i : Unit, \ldots, l_n : Unit >$. For simplicity, we use the syntax for variants as presented in problem 2 before it is desugared into its corresponding representation as sum type. For the field labels introduced as part of an enumeration, we assign the value unit. To compute on enumerations, we use the case of variants and construct an abstraction that disregards the argument it is supplied. Below is a concrete example of desugaring enumeration into variants.

Here, We show how to translate the following program phrase containing enumerations and literals into a term with "Unit" and variants. We define an enumeration type Days which describes the days of a week and the function isWeekend:Days $\rightarrow$ Bool which decides whether the given day is part of the weekend.

```
Days = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)

isWeekend = λd:Days
    case d of
        Monday -> false
        Tuesday -> false
        Wednesday -> false
        Thursday -> false
        Friday -> false
        Saturday -> true
        Sunday -> true

isWeekend Monday ⇒ false
```

Here, we show the above program phrase expressed with variants.

```
Days = <Monday:Unit, Tuesday:Unit, Wednesday:Unit, Thursday:Unit, Friday:Unit, Saturday:Unit, Sunday:Unit>

isWeekend = λd:Days
    case d of
        (λx:Unit. false)
        (λx:Unit. false)
        (λx:Unit. false)
        (λx:Unit. false)
        (λx:Unit. false)
        (λx:Unit. true)
        (λx:Unit. true)

isWeekend (Monday unit) ⇒ false
```

Problem 4

We show that every term of the pure lambda-calculus is typable in a system of recursive types. Let D be the type D = μX.X $\rightarrow$ X. Note that when we unfold the type D once, we get the type [X $\mapsto$ (μX. X $\rightarrow$ X)]X $\rightarrow$ X = (μX. X $\rightarrow$ X) $\rightarrow$ (μX. X $\rightarrow$ X). Similarly, we can fold (μX. X $\rightarrow$ X) $\rightarrow$ (μX. X $\rightarrow$ X) to return to μX.X $\rightarrow$ X. We now show that every term of the pure lambda-calculus can be assigned the type D through induction on the size of the term. For an arbitrary term t, we consider the following cases.

- The term t is a variable. Then we can assign D to be the type of t. Every time a new variable is introduced through a lambda binding, it will be assigned the type D.
- The term t is a lambda abstraction of the form (λx:D. M). By the induction hypothesis, M has the type D. so t has the type D $\rightarrow$ D. By folding D $\rightarrow$ D, we get the type D, and thus t has the type D.

- The term t is an application of the form (M N). By the induction hypothesis, M and N both have the type D. We can then unfold D to yield the type D $\rightarrow$ D. Thus, given that M has the type D $\rightarrow$ D, and N has type D, (M N) has the type D.

Problem 5

1. $\lceil 0 \rceil \triangleq$ inl unit
   $\lceil n \rceil \triangleq inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots))$. Note, we use subscript to denote the number of occurrences of inr. In the case above, inr occurs n times.

2. succ $\triangleq$ (λx:Nat. inr x)

3. pred $\triangleq$ (λx:Nat. case x (λy:Unit. 0) (λz:Nat. z))

4. iszero $\triangleq$ (λx:Nat. case x (λy:Unit. true) (λz:Nat. false))

5. We verify the following expressions.

   (a) $\text{succ}\ \lceil n \rceil = \text{succ}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots)))$
   $\Rightarrow (inr_{n+1}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots)))) = \lceil n+1 \rceil$

   (b) $(\text{iszero}\ \lceil 0 \rceil) = (\text{iszero (inl unit)})$
   $\Rightarrow$ case (inl unit) (λy:Unit. true) (λz:Nat. false)
   $\Rightarrow$ true
   $(\text{iszero}\ \lceil n \rceil) = (\text{iszero}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots))))$
   $\Rightarrow \text{case}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots)))$ (λy:Unit. true) (λz:Nat. false)
   $\Rightarrow$ false

   (c) $(\text{pred}\ (\text{succ}\ \lceil n \rceil)) = \text{pred}\ (\text{succ}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots))))$
   $\Rightarrow \text{pred}\ (inr_{n+1}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots\ ))))$
   $\Rightarrow \text{case}\ (inr_{n+1}\ (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots\ ))))$
   (λy:Unit. inl y) (λz:Nat. z)
   $\Rightarrow (inr_n\ (inr_{n-1}\ (\ldots\ (inr_1\ (\text{inl unit}))\ \ldots\ )))) = \lceil n \rceil$

Problem 6

We prove the progress and preservation theorems for the iso-recursive system as defined in Pierce, page 276.

Progress Theorem

We show this through induction on a derivation of t:T. We extend the proof of the progress theorem that appeared in Pierce, page 96 with the following two cases to address the new type rules in this iso-recursive system.

(Case T-Fld) t = fold[U]t_1: U U = μX.T_1 t_1:[X $\mapsto$ U]T_1
By the induction hypothesis, t_1 is either a value or there is some t_1' such that $t_1 \rightarrow t_1'$. If t_1 is a value, then fold[U]t_1 is a value as well. Otherwise, fold[U]t_1 $\Rightarrow$ fold[U]t_1' by the evaluation rule (E-Fld).

(Case T-Unfld) t = unfold[U]t_1: [X $\mapsto$ U]T_1 U = μX.T_1 t_1:U
By the induction hypothesis, t_1 is either a value or there is some t_1' such that $t_1 \rightarrow t_1'$. If t_1 is a value, then since t_1:μX.T_1, t_1 = fold[T]v. Thus, unfold[U](fold[T]v) $\rightarrow$ v by the evaluation rule (E-UnfldFld). Otherwise, unfold[U]t_1 $\Rightarrow$ unfold[U]t_1' by the evaluation rule (E-Unfld).

Preservation Theorem

We show this through induction on a derivation of t:T. We extend the proof of the preservation theorem that appears in Pierce, page 97 with the following two cases to address the new type rules in this iso-recursive system.

Case (T-Fld) t = fold[U]t_1: U $\qquad$ U = μX.T_1 $\qquad$ t_1:[X $\mapsto$ U]T_1.

Case 1: If t_1 is a value, then fold[U]t_1 is a value as well. Since we can not further evaluate fold[U]t_1, then the theorem holds in this case.

Case 2: we take one step of evaluation using the (E-Fld) rule. There exists a t_1' such that $t_1 \Rightarrow t_1'$ and fold[U]$t_1 \Rightarrow$ fold[U]t_1'. By the induction hypothesis, t_1':[X $\mapsto$ U]T_1 since t_1:[X $\mapsto$ U]T_1. Thus, fold[U]t_1':U by (T-Fld) since the premises did not change.

Case (T-Unfld) t = unfold[U]t_1: [X $\mapsto$ U]T_1 $\qquad$ U = μX.T_1 $\qquad$ t_1:U

Case 1: we take a step of evaluation using the (E-Unfld) evaluation rule. Then there exists a t_1' such that $t_1 \rightarrow t_1'$ and unfold[U]$t_1 \rightarrow$ unfold[U]t_1'. By the induction hypothesis, t_1':U since t_1':U. Thus, unfold[U]t_1:[X $\mapsto$ U]T_1 by (T-Unfld) since the premises did not change.

Case 2: we take one step of evaluation using the (E-UnfldFld) evaluation rule. Then t_1 = fold[T]v and unfold[U](fold[T]v) $\Rightarrow$ v. We now show that v:[X $\mapsto$ U]T_1. Since fold[T]v=t_1:U and U = μX.T_1, we can use (T-Fld) to show that v:[X $\mapsto$ U]T_1 as desired.

Now, we consider the equi-recursive approach. The progress and preservation theorems hold here as well although in order to prove that we have to use a mechanism presented in [Pierce] Chapter 21 (and not presented in class). The important thing to notice here is that the equi-recursive approach makes explicit use of the "infinite interpretation" of recursive types, and so we can no longer use induction in proofs.