CMCS 22100 — Programming Languages
Midterm Solutions
November 5 , 2009

1. [10 points] Evaluate the Arith expression (we omit the num and var syntax constructors for brevity):

```
let x = plus(3,2)
 in let y = times(x,3)
     in plus(y,times(x,2))
```

Use the rules for $\mapsto$, and give derivations of the transitions for the last three steps.

Solution: The expression is given in "concrete syntax" style, but for greater compactness of expression we will use the syntax constructor style.

```
(0)  let(plus(3,2), x.let(times(x,3), y.plus(y, times(x,2))))   ↦
(1)  let(5, x.let(times(x,3), y.plus(y, times(x,2))))   ↦
(2)  let(times(5,3), y.plus(y, times(5,2)))   ↦
(3)  let(15, y.plus(y, times(5,2)))   ↦
(4)  plus(15, times(5,2))   ↦
(5)  plus(15, 10)   ↦
(6)  25
```

The derivation of (3) $\mapsto$ (4) is a single rule derivation using rule E3 (the Let instruction). The derivation of (4) $\mapsto$ (5) is a two-step derivation using instruction E2 followed by search rule E5. The derivation of (5) $\mapsto$ (6) is a single rule derivation using E1.

2. [10 points] State the formal induction principle as a logical formula for proving properties of lists of natural numbers, defined by the abstract syntax:

```
list ::= nil | cons(n,list)
```

Solution:

$$P(\texttt{nil})\ \&\ (\forall l.P(l) \Longrightarrow \forall n.P(\texttt{cons}(n,l))) \Longrightarrow \forall l.P(l).$$

3. [20 points] Let $\mapsto$ be the small-step transition relation for Arith, Prove that $\texttt{plus}(e_1, e_2) \overset{n}{\mapsto} \texttt{plus}(e_1', e_2)$ if, and only if, $e_1 \overset{n}{\mapsto} e_1'$.

Proof: [$\Longrightarrow$] The proof is by induction on n.

Case $n = 0$: Assume $\texttt{plus}(e_1, e_2) \overset{0}{\mapsto} \texttt{plus}(e_1', e_2)$. By the definition of $\overset{n}{\mapsto}$, for $n = 0$, we have $plus(e_1, e_2) = \texttt{plus}(e_1', e_2)$, so $e_1 = e_1'$, and hence $e_1 \overset{0}{\mapsto} e_1'$.

Case $n = k + 1$: The by the definition of $\overset{n}{\mapsto}$, there exists an expression e such that

$$\texttt{plus}(e_1, e_2) \quad \mapsto \quad e \tag{1}$$

$$e \quad \overset{k}{\mapsto} \quad \texttt{plus}(e_1', e_2) \tag{2}$$

Claim: The first transition is by the left search rule for plus, $e = \texttt{plus}(e', e_2)$ for some e' such that $e_1 \mapsto e'$.

The transition could not be by the instruction for plus, since then e would be a number n, which is a final state, an there is no transition sequence from n to $\texttt{plus}(e_1', e_2)$. If the transition was by the left search rule, then e_1 would have to be a value n and $e = \texttt{plus}(n, e_2')$, where $e_2 \mapsto e_2'$. But then we can prove by induction on k that all the transitions in the sequence for $e \overset{k}{\mapsto} \texttt{plus}(e_1', e_2)$ must also use the right search rule for plus, and the corresponding premises of these transition rules give a transition sequence $e_2' \overset{k}{\mapsto} e_2$, which is impossible, since we can prove that in Arith, a nonempty transition sequence cannot return to its starting point (this is a corollary of the proof of termination of Arith expression evaluation).

Thus by this claim, $e = \texttt{plus}(e', e_2) \overset{k}{\mapsto} \texttt{plus}(e_1', e_2)$. The Induction Hypothesis then says that $e' \overset{k}{\mapsto} e_1'$. This together with the assumption that $e_1 \mapsto e'$ yields $e_1 \overset{n}{\mapsto} e_1'$ by the definition of $\overset{n}{\mapsto}$.

[$\Longleftarrow$]: We assume $e_1 \overset{n}{\mapsto} e_1'$ and must show that $\texttt{plus}(e_1, e_2) \overset{n}{\mapsto} \texttt{plus}(e_1', e_2)$. Again the proof is by induction on n.

Case n = 0. Then $e_1 = e_1'$ so and hence $\texttt{plus}(e_1, e_2) \overset{0}{\mapsto} \texttt{plus}(e_1', e_2)$.

Case n = k+1. Then by the definition of $\overset{n}{\mapsto}$, there exists an e such that $e_1 \mapsto e$ and $e \overset{k}{\mapsto} e_1'$. The induction hypothesis is:

$$\text{(IH)} \quad \texttt{plus}(e', e_2) \overset{k}{\mapsto} \texttt{plus}(e_1', e_2)$$

But $\texttt{plus}(e_1, e_2) \mapsto \texttt{plus}(e', e_2)$ by the left search rule for plus, and this together with the IH gives $\texttt{plus}(e_1, e_2) \overset{n}{\mapsto} \texttt{plus}(e_1', e_2)$.

4. [10 points] The self-apply function could be expressed in MinML as $\texttt{fun}(x : \tau)\ \texttt{is}\ \texttt{apply}(x, x)$. Find a type τ such that this is well typed according to the typing rules, or show that this is impossible.

Solution: Here we are using the simple, nonrecursive version of function expressions. By the typing rule for a fun expression, we would have the premise $[x : \tau] \vdash \texttt{apply}(x, x) : \tau'$. In order to derive this premise by the apply rule (the only applicable rule), we would have to have the two premises $[x : \tau] \vdash x : \tau \to \tau'$ and also $[x : \tau] \vdash x : \tau$. The second of these certainly holds for any τ by the variable rule. The first would only hold if $\tau = \tau- > \tau'$. Since the size of type expression on the left is clearly smaller than the type expression on the right, this is clearly impossible.

5. [10 points] The Small Step evaluation rules for MinML define Call-by-Value evaluation. The Call-by-Name (CBN) version of MinML differs from the one discussed in class in one way: in function applications the function arguments are "passed" before they are evaluated, rather than after evaluation. The primitive operator expressions like $\texttt{plus}(e_1, e_2)$ still need to have their arguments evaluated before they can be reduced.

(a) [15 points] Give any new or changed rules for small-step evaluation ($\mapsto$) for the CBN MinML.

Solution: Rule (9.21), the right search rule for apply is dropped, because we don't want to evaluate the argument of an application. Rule (9.20), the left search rule for apply is unchanged since application still requires that the function be evaluated. Rule (9.16), the apply instruction, is replaced by

$$\frac{(v = \texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ e)}{\texttt{apply}(v, e_2) \mapsto \{v, e_2/f, x\}e} \qquad (9.16')$$

All the other rules are unchanged.

(b) [10 points]: Do the typing rules for CBN MinML differ from those of the normal CBV MinML? If so, show the altered typing rules.

Solution: The typing rules for Call-By-Name are the same as those for Call-By-Value. Typing judgements tell us about the kind of value computed by an expression, and this value and its type will not change when we change the order of evaluation.

In a purely functional language like MinML, the only aspect of a computation that is affected by the order of evaluation is whether a computation will terminate. Because CBN makes it possible to avoid evaluating a function argument that is not used by the function (e.g. $\texttt{fun}\ f(x : \tau_1) : \tau_2\ \texttt{is}\ 3$), the evaluation of more expressions will terminate in CBN. But typing judgements don't say anything about termination of the expression being typed – they just say that if the evaluation of the expression *does* terminate, the resulting value will have the specified type.