CS232F: Programming Languages
II semester 2001-02

Minor 2 Thu 14 Mar 2002 V 417-418 14:30-15:30 Max Marks 50

Note:

1. Answer in the space provided on the question paper.
2. The answer booklet you have been given is for **rough work only** and will not be collected.

1. In the semantics for arithmetic expressions, we have seen that the transition system gets "stuck" if there is a name which does not occur in the domain of the environment. Now consider the *small-step semantics* of expressions.

   (a) Add a new element $\perp$ to the set Num of values, such that all operations such as ADD are strict with respect to this new element. Define *additional* semantic rules for the expression language

   $$e ::= x \mid n \mid (e_1 + e_2)$$

   so that no configuration gets stuck.

   (b) How will you extend the semantics when the language of expressions is expanded to include private definitions, i.e.

   $$e ::= x \mid n \mid (e_1 + e_2) \mid \mathbf{let}\ \ x \stackrel{def}{=} e_1\ \ \mathbf{in}\ \ e_2$$

   Solution

   (a) Since ADD is strict with respect to $\perp$ have $ADD(\perp, \_) = ADD(\_, \perp) = \perp$. Hence we need to add just one rule which ensures that configurations don't get stuck (see Table 2 in the notes).

   $$(vbl_\perp)\quad \frac{}{\gamma \vdash x \longrightarrow^e_1 \perp} \quad \text{provided } x \notin dom(\gamma).$$

   (b) There is no need at all to add any more new rules to include private definitions. The existing rules (reproduced below)

   $$\frac{\gamma \vdash e_1 \longrightarrow^e_1 e_1'}{\gamma \vdash \mathbf{let}\ x \stackrel{def}{=} e_1\ \mathbf{in}\ e_2 \longrightarrow^e_1 \mathbf{let}\ x \stackrel{def}{=} e_1'\ \mathbf{in}\ e_2}$$

   $$\frac{\gamma[x \mapsto n_1] \vdash e_2 \longrightarrow^e_1 e_2'}{\gamma \vdash \mathbf{let}\ x \stackrel{def}{=} n_1\ \mathbf{in}\ e_2 \longrightarrow^e_1 \mathbf{let}\ x \stackrel{def}{=} n_1\ \mathbf{in}\ e_2'}$$

   $$\frac{}{\gamma \vdash \mathbf{let}\ x \stackrel{def}{=} n_1\ \mathbf{in}\ n_2 \longrightarrow^e_1 n_2}$$

   are adequate to take into account the presence of $\perp$. Note the following however, when e_1 evaluates to $\perp$.

   i. $\mathbf{let}\ \ x \stackrel{def}{=} e_1\ \ \mathbf{in}\ \ e_2$ is not strict with respect to $\perp$, i.e. if $x \notin fv(e_2)$ and $x \mapsto \perp$ then it is quite possible that the **let**-expression evaluates to a well-defined value in Num.

   ii. The scope rules make it necessary to have mappings of the form $[x \mapsto \perp]$ when evaluating e_2 since it is quite possible that there is a previous well-defined value in an outer scope for x.

   iii. If e_2 evaluates to $\perp$ due to the presence of an undefined variable in $fv(e_2)$ or because some defined variable has the value $\perp$ then the **let**-expression too evaluates to $\perp$.

(4 + 6 = 10 marks)

2. Consider the semantics of the WHILE language *without* any local declarations. The following program segment generates the n-th Fibonacci number for positive values of n. The set of states of this program is the set of 5-tuples $\langle a, b, c, j, n\rangle$ where b is the value of the j-th Fibonacci number. The initial state from which the while loop is executed is $\sigma_0 = \langle 0, 1, 1, 1, n\rangle$.

$$\textbf{while} \ \ (j <= n) \ \ \textbf{do} \ \ c := b + a; \ a := b; \ b := c; \ j := j + 1 \ \ \textbf{end}$$

(a) Define the relation $\mathcal{F}_i$ inductively in terms of the state components and i.

(b) Determine the smallest value k such that for all $m \geq k$, $\mathcal{F}_k = \mathcal{F}_m$.

(c) Prove that for all $m \geq k$, $\mathcal{F}_k = \mathcal{F}_m$.

Solution Let $\langle\sigma_0, \sigma_i\rangle$ be typical elements of the relation $\mathcal{F}_i$, for each $i \geq 0$. Further the values of the variables in each state σ_i are denoted a_i, b_i, c_i, j_i respectively (since the value of n does not change, we ignore the subscript on n. It is further clear that $\langle\sigma_0, C\rangle \Longrightarrow \sigma_1$, where C is the body of the **while**-loop and $a_1 = b_0$, $b_1 = c_1 = b_0 + a_0$ and $j_1 = j_0 + 1$.

(a) We have $\mathcal{F}_0 = \{\langle\sigma_0, \sigma_0\rangle \mid j_0 > n\}$. In other words, $\langle\sigma_0, \sigma_0'\rangle \in \mathcal{F}_0 \Leftrightarrow j_0 > n \wedge \sigma_0 = \sigma_0'$. Further, we have

$$\langle\sigma_0, \sigma_1\rangle \in \mathcal{F}_1 \Leftrightarrow (j_0 > n \wedge \langle\sigma_0, \sigma_1\rangle \in \mathcal{F}_0) \quad \vee \quad (j_0 + 1 > n \ \wedge \ \langle\sigma_0, C\rangle \Longrightarrow \sigma_1 \ \wedge \ \langle\sigma_1, \sigma_1\rangle \in \mathcal{F}_0)$$

and

$$\begin{aligned}\langle\sigma_0, \sigma_2\rangle \in \mathcal{F}_2 \Leftrightarrow \quad & (j_0 > n \wedge \langle\sigma_0, \sigma_2\rangle \in \mathcal{F}_0) \ \vee \ (j_0 + 1 > n \ \wedge \ \langle\sigma_0, C\rangle \Longrightarrow \sigma_1 \ \wedge \ \langle\sigma_1, \sigma_2\rangle \in \mathcal{F}_1) \ \vee \\ & (j_0 + 2 > n \ \wedge \ \langle\sigma_0, C\rangle \Longrightarrow \sigma_1 \ \wedge \ \langle\sigma_1, \sigma_2\rangle \in \mathcal{F}_2)\end{aligned}$$

which by an inductive generalization yields, for all $i > 0$,

$$\begin{aligned}\langle\sigma_0, \sigma_i\rangle \in \mathcal{F}_i \Leftrightarrow \quad & (j_0 > n \wedge \langle\sigma_0, \sigma_i\rangle \in \mathcal{F}_0) \vee \\ & (\exists l[0 < l \leq i \wedge j_0 + l > n] \ \wedge \ \langle\sigma_0, C\rangle \Longrightarrow \sigma_1 \ \wedge \ \langle\sigma_1, \sigma_i\rangle \in \mathcal{F}_{i-1})\end{aligned}$$

(b) Given $j_0 = 1$, the smallest value of k such that for all $m \geq k$, $\mathcal{F}_k = \mathcal{F}_m$ is $k = n$, since for $i = n$, the predicate $\exists l[0 < l \leq i \wedge j_0 + l > n]$ is always true. Hence the relation $\mathcal{F}$ stabilizes to the value of $\mathcal{F}_n$, where

$$\begin{aligned}\langle\sigma_0, \sigma_1\rangle \in \mathcal{F}_n \Leftrightarrow \quad & (j_0 > n \wedge \langle\sigma_0, \sigma_n\rangle \in \mathcal{F}_0) \ \vee \\ & (\exists l[0 < l \leq i \wedge j_0 + l > n] \ \wedge \ \langle\sigma_0, C\rangle \Longrightarrow \sigma_1 \ \wedge \ \langle\sigma_1, \sigma_n\rangle \in \mathcal{F}_{n-1})\end{aligned}$$

(c) It is sufficient to prove that $\mathcal{F}_{n+1} = \mathcal{F}_n$, which is the basis of an inductive proof for all $m \geq 1$ in order to show that $\mathcal{F}_{n+m} = \mathcal{F}_n$. We already know that $\mathcal{F}_n \subseteq \mathcal{F}_{n+1}$. Now suppose there exists, $\langle\sigma_0, \sigma_{n+1}\rangle \in \mathcal{F}_{n+1} - \mathcal{F}_n$. Then since $\langle\sigma_0, \sigma_{n+1}\rangle \notin \mathcal{F}_0$, we have $\langle\sigma_1, \sigma_{n+1}\rangle \in \mathcal{F}_n$ but $\langle\sigma_1, \sigma_{n+1}\rangle \notin \mathcal{F}_{n-1}$. This implies $j_1 + n > n$ but $j_1 + n - 1 \not> n$, which is clearly a contradiction since $j_1 = 2$.

(5 + 2 + 8 = 15 marks)

3. **Multiple assignment**. In order to transform the store more than one location at a time, it is useful to consider the following generalization of the assignment command. $x_1, \ldots, x_n := e_1, \ldots, e_n$

   (a) Define <u>small-step</u> structural operational rules for this statement so that it is semantically equivalent to the following block of code.

   $$\textbf{var}\ \ t_1; \ldots; t_n\ \ \textbf{begin}\ \ t_1 := e_1;\ \ \ldots\ \ t_n := e_n;\ x_1 := t_1;\ \ \ldots;\ x_n := t_n\ \ \textbf{end}$$

   where the variables $t_1, \ldots, t_n$ are fresh and do not occur anywhere in the program.

   (b) Prove that your semantics is equivalent to the above command.

<u>Solution</u>

(a) The semantic equivalence in Question 3a suggests that the multiple assignment cannot be evaluated in parallel (for instance one cannot allow a simultaneous substitution of the form $\sigma[\gamma(x_1) \mapsto m_1), \ldots, \gamma(x_n) \mapsto m_n]$. This is of course, deliberate, since there is *no* guarantee that the variables $x_1, \ldots, x_n$ are all distinct. Hence if not all variables are different, then there has to be a systematic *sequential* method of evaluation which is given by the following algorithm.

   i. Evaluate each of the expressions $e_1, \ldots, e_n$ <u>*in order of occurrence*</u>.
   ii. Store the values temporarily in some new locations $t_1, \ldots, t_n$ respectively. But for specifying the operational semantics we don't need these temporary locations, we may directly use substitutions to replace each expression by its value.
   iii. Copy the values in $t_1, \ldots, t_n$ <u>*in order*</u> into the locations of $x_1, \ldots, x_n$ respectively.

   This yields the following rules[1]

$$\frac{\gamma, \sigma\ \vdash\ e_i \longrightarrow_1^e e_i'}{\begin{array}{l}\gamma,\ \vdash\ \langle \sigma, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n := m_1, \ldots, m_{i-1}, e_i, e_{i+1}, \ldots, e_n \rangle \longrightarrow_1 \\ \langle \sigma, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n := m_1, \ldots, m_{i-1}, e_i', e_{i+1}, \ldots, e_n \rangle\end{array}}\ 1 \leq i \leq n$$

$$\frac{\gamma, \sigma\ \vdash\ e_i \longrightarrow_1^e m_i}{\begin{array}{l}\gamma,\ \vdash\ \langle \sigma, x_1, \ldots, x_{i-1}, x_i x_{i+1}, \ldots, x_n := m_1, \ldots, m_{i-1}, e_i, e_{i+1}, \ldots, e_n \rangle \longrightarrow_1 \\ \langle \sigma, x_1, \ldots, x_i, x_{i+1}, \ldots, x_n := m_1, \ldots, m_i, e_{i+1}, \ldots, e_n \rangle\end{array}}\ 1 \leq i \leq n$$

$$\frac{}{\gamma\ \vdash\ \langle \sigma, x_1, \ldots, x_n := m_1, \ldots, m_n \rangle \longrightarrow_1 \sigma[\gamma(x_1) \mapsto m_1] \ldots [\gamma(x_n) \mapsto m_1]}$$

(b) The proof is intuitively quite easy, given the structure of the above rules (but it maybe messy to write). Here's an outline. Let γ and σ, be the environment and store respectively in which the multiple assignment and the block given Question 3a are both executed.

   $\gamma\ \vdash\ \langle \sigma, x_1, \ldots, x_n := e_1, \ldots, e_n \rangle (\longrightarrow_1)^* \sigma'$. Clearly the last step of the proof was an application of the last rule, and $\sigma' = \sigma[\gamma(x_1) \mapsto m_1] \ldots [\gamma(x_n) \mapsto m_1]$. Hence there exist steps $j_1 \leq j_2 \leq \ldots \leq j_n$ where the values $m_1, \ldots, m_n$ were obtained as values of expressions $e_1, \ldots, e_n$ respectively.

   Now consider the block in Question 3a. It is then possible to use these subproofs in the proof of store σ'' corresponding to an environment γ'' which includes the fresh variables $t_1, \ldots, t_n$. The problem then reduces to showing that if for all i, $1 \leq i \leq n$, if $\sigma''(\gamma''(t_i)) = m_i$ then the effect of the assignments $x_1 := t_1;\ \ldots;\ x_n := t_n$ is to obtain a new state σ''' such that for all i, $1 \leq i \leq n$, $\sigma'''(\gamma''(x_i)) = \sigma'(\gamma(x_i))$. Finally given that the variables t_i are fresh we have $dom(\gamma) = dom(\gamma'') - \{t_i \mid 1 \leq i \leq n\}$, for all $y \in dom(\gamma)$, $\gamma(y) = \gamma''(y)$ and hence $\sigma'' \wr dom(\sigma) = \sigma'$.

(6 + 8 = 14 marks)

[1] the rules have been written in slightly non-standard notation because they are too wide for the page.

4. Write a CSP program for the following problem of multiset partitioning. There are two processes P and Q. P has a list of $m > 0$ values $u_1, \ldots, u_m$ and Q has a list of $n > 0$ values $v_1, \ldots, v_n$. The two processes keep exchanging values so that eventually P has a list of values $U_1, \ldots, U_m$ and Q has a list $V_1, \ldots, V_n$ such that

   - $[U_1, \ldots, U_m, V_1, \ldots, V_n]$ is a permutation of $[u_1, \ldots, u_m, v_1, \ldots, v_n]$, and
   - $max\{U_1, \ldots, U_m\} \leq min\{V_1, \ldots, V_n\}$

   Both processes should terminate after the required state has been reached.

   Solution The following points should be quite clear in order to avoid deadlock and non-termination.

   - The two processes exchange values strictly in turn (like in most 2-person games)
   - One of them has to initiate the computation (just like a 2-person game). The solution can't be perfectly symmetric.
   - Each process should know when the other wants to communicate with it,
   - Communications really cannot be in guards, since then termination is not guaranteed and one of the processes is likely to get blocked on a communication wait.
   - We will assume that the following functions are freely available:
     - min, max for finding minimum, maximum of lists/multisets
     - $L + x$, $L - x$ for inserting or deleting an element x to or from a list/multiset L.
     - The two processes terminate as soon as they realize that they are exchanging either the same values or values which should remain with themselves.
     - The variables that the two processes use are as follows:

       P uses
       * `S`: the current list of values it possesses,
       * `mymax`: the value of the maximum element in `S`,
       * `qmin`: the current minimum value with Q.

       Q uses
       * `T`: the current list of values it possesses,
       * `mymin`: the value of the minimum element in `T`,
       * `pmax`: the current maximum value with P.

```
P::  S := [u1, ..., um];
     mymax := max S;
     Q!mymax;
     Q?qmin;
     do qmin < mymax |>

          S := (S - mymax) + qmin;
          mymax := max S;
          Q!mymax;
          Q?qmin
     od
```

```
Q::  T := [v1, ..., vn];
     mymin := min T;
     P?pmax;

     do mymin > pmax |>
          P!mymin;
          T := (T - mymin) + pmax;
          mymin := min T;
          P?pmax;

     od;
     P!mymin
```

(11 marks)