CS2PLD: Programming Language Design Paradigms

Exam May 2024

Question 1:

1. Briefly explain what *"operational semantics"* means.

   —

   Operational semantics is a kind of formal semantics, where the meaning of a program is described by giving the computation steps required to execute a program.

   Denotational and axiomatic semantics describe the effect of each command, which is useful for programmers but the advantage of operational semantics is that it can be directly used to implement an interpreter for the programming language.

2. The abstract syntax of SIMP is defined by:

$$\begin{array}{lll} C & ::= & skip \mid l := E \mid C; C \mid if\ B\ then\ C\ else\ C \\ & & \mid while\ B\ do\ C \\ E & ::= & !l \mid n \mid E\ op\ E \\ op & ::= & + \mid - \mid * \mid / \\ B & ::= & True \mid False \mid E\ bop\ E \mid \neg B \mid B \wedge B \\ bop & ::= & > \mid < \mid = \end{array}$$

   We extend the language SIMP by adding two new arithmetic operators, with the abstract syntax specified by the following grammar:

$$E ::= ++l \ \mid \ l{+}{+}$$

   where l represents a location, as usual.

   (a) Draw an abstract syntax tree for each of the programs P_1 and P_2 below:

   P_1:

$$(x := 0;\ y := {+}{+}x);\ \texttt{if}\ !x = 0\ \texttt{then}\ y :=!y - 1\ \texttt{else}\ y :=!y + 1$$

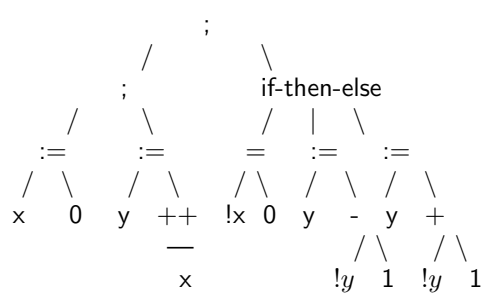
   P_2 :

$$x := 0;\ (y :=!x + 1; \texttt{if}\ !x = 0\ \texttt{then}\ y :=!y - 1\ \texttt{else}\ y :=!y + 1)$$

   —

   Both trees have a root node labelled by ;.

   The tree for P_1 has a left subtree labelled by ; with two children representing assignments $x := 0$ and $y := {+}{+}x$, and the right subtree represents the conditional:



The tree for P_2 is similar, but has a left subtree representing $x := 0$ (see the leftmost subtree above) and a right subtree representing the sequence $(y :=!x+1; if\ !x = 0\ then\ y :=!y-1\ else\ y :=!y+1)$, that is, root ;, left subtree representing $y :=!x+1$ and right subtree representing the conditional as shown above.

(b) The small-step semantics for the new operators $++l$ and $l++$ is defined by the following axioms:

$$\frac{}{\langle ++l, s\rangle \to \langle n+1, s[l \mapsto n+1]\rangle} \quad \text{if } s(l) = n$$

$$\frac{}{\langle l++, s\rangle \to \langle n, s[l \mapsto n+1]\rangle} \quad \text{if } s(l) = n$$

Explain with your own words the behaviour of the new operators.

—

$++l$ adds 1 to the value stored in l and returns the result.

$l++$ returns the value stored in l before adding 1 to the value stored in l.

(c) Recall the axioms and rules defining the (small-step) semantics of arithmetic expressions in SIMP:

$$\frac{}{\langle !l, s\rangle \to \langle n, s\rangle} \quad \text{if } s(l) = n$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1\ op\ E_2, s\rangle \to \langle E_1'\ op\ E_2, s'\rangle}$$

$$\frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle n_1\ op\ E_2, s\rangle \to \langle n_1\ op\ E_2', s'\rangle}$$

$$\frac{}{\langle n_1\ op\ n_2, s\rangle \to \langle n, s\rangle} \quad \text{if } (n_1\ op\ n_2) = n$$

Assuming the standard semantics for the assignment, sequence and if-then-else constructs, are the two programs P_1 and P_2 given in part 2a equivalent or not? Justify your answer (answers without justification will have no marks).

—

Two programs are equivalent if, when started in the same memory state, they both reach the same memory state or neither of them terminates.

In this case, when started in a memory state s, the first program reaches a state where $s(x) = 1$ and $s(y) = 2$ whereas the second terminates with $s(x) = 0$ and $s(y) = 0$. Therefore the programs are NOT equivalent.

(d) Give a principle of induction that could be used to prove that a certain property P holds for all the arithmetic expressions (E) in the extended version of SIMP.

Prove by induction that for any arithmetic expression E in the extended version of SIMP, if $\langle E, s\rangle \rightarrow^* \langle n, s'\rangle$ then s and s' can only differ in locations l such that either $++l$ or $l{+}{+}$ occur in E.

—

Induction principle:

To prove $\forall e.P(e)$ where e is an arithmetic expression in the extended language, we need to prove base cases and induction steps.

Base cases: $P(n), P(!l), P(++l), P(l{+}{+})$ for all l

Induction Step:

For any e_1, e_2, if $P(e_1)$ and $P(e_2)$ hold then $P(e_1\ op\ e_2)$ holds where op is one of $+|-|*|/$.

Proof:

Base cases: For n and $!l$, $s = s'$. For $++l$ and $l{+}{+}$ the axioms provided show that the only difference is in l, as required.

Induction step:

First notice that if $\langle E_1 op E_2, s\rangle \rightarrow^* \langle n, s'\rangle$ then $\langle E_1, s\rangle \rightarrow^* \langle n_1, s_1\rangle$, $\langle E_2, s_1\rangle \rightarrow^* \langle n_2, s_2\rangle$, $n = n_1\ op\ n_2$ and $s' = s_2$.

Assuming the property holds for E_1 and E_2, it means that s and s_1 only differ in locations l such that $++l$ or $l{+}{+}$ occur in E_1 and s_1 and s_2 only differ in locations l such that $++l$ or $l{+}{+}$ occur in E_2. Hence, s and s' can only differ in locations l such that $++l$ or $l{+}{+}$ occur in E_1 or E_2.

Question 2:

1. Consider the following SFUN program:

   ```
   big = big ∧ True
   cond(x,y,z) = if x then y else z
   ```

   Relative to the program, consider the terms

   `cond(big,False,False)`,

   `cond(False,big,False)`.

   For each of these terms, do the following two things:

   - Explain briefly whether or not this term evaluates to a value under call-by-value semantics, and why or why not.
   - Explain briefly whether or not this term evaluates to a value under call-by-name semantics, and why or why not.

   —

   The first term does not evaluate under CBV, since the argument `big` cannot be evaluated. Also, the first term does not evaluate under CBN, since in `if big ...` the condition expression cannot be evaluated.

   The second term does not evaluate under CBV, since the argument `big` cannot be evaluated. The second term does evaluate under CBN: this term is (in one step) evaluated as

   `if False then big else False`.

   According to the semantics, since the condition is False, then this term evaluates to a value so long as the term following the else does; the term following the else clearly evaluates to False.

2. Suppose the following definitions have been introduced in Haskell:

   ```
   summer x = if x <= -1 then -1 else x + summer(x-1)
   plusser (x,y) = x + y
   id x = x
   ```

   Answer the following questions:

   (a) What does the following expression evaluate to: `summer 0`

   (b) What does the following expression evaluate to: `summer 3`

   (c) What is the type of the following expression: `curry plusser`

   (d) What is the type of the following expression: `(.) id`

   (e) What is the type of the following expression: `(:)`

   The following interaction with ghci may help you recall some notions from Haskell:

```
ghci> plusser (x,y) = x + y
ghci> :t plusser
plusser :: Num a => (a, a) -> a
ghci> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
ghci> 1 : [2,3]
[1,2,3]
ghci> "a" : ["c","e","f"]
["a","c","e","f"]
```

—

(a) `-1`

(b) `5`

(c) `Num c => c -> c -> c`

(d) `(a -> c) -> a -> c`

(e) `a -> [a] -> [a]`

3. For each of the following pairs of terms, written using Prolog notation, state whether or not they unify, and if they unify, give a *most general unifier* that unifies them.

   (a) `p(X,Y), q(X,Y)`

   (b) `p(X,Y), p(Y,Z)`

   (c) `p(f(X),Y), p(Y,f(Z))`

   (d) `p(s(X),Y), p(s(Y),s(s(Z)))`

   (e) `p(f(f(X)),Y) = p(f(Y),f(X))`

   Next, consider the following 3 substitutions, written using Prolog notation. (So here, upper case letters denote variables.)

   $\alpha = \{X \mapsto Y, Y \mapsto X\}$

   $\beta = \{X \mapsto f(Z), Y \mapsto f(Z)\}$

   $\gamma = \{X \mapsto f(A), Y \mapsto f(B)\}$

   List all pairs (σ, θ), where σ and θ are among the three given substitutions, such that the relationship $\sigma \preceq \theta$ holds. For example, you would list (α, β) in your answer if you determined the relationship $\alpha \preceq \beta$ to hold. (Recall that for two substitutions σ, θ, the relationship $\sigma \preceq \theta$ is defined to hold when there exists a substitution θ' such that $\sigma = \theta\theta'$; when this holds, it is said that θ is "more general" than σ.)

   —

   (a) no

   (b) $\{X, Y \mapsto Z\}$

   (c) $\{X \mapsto Z, Y \mapsto f(Z)\}$

   (d) $\{X, Y \mapsto s(s(Z))\}$

   (e) $\{Y \mapsto f(X)\}$

Regarding the substitutions:
$(\alpha, \alpha), (\beta, \beta), (\gamma, \gamma)$
$(\beta, \alpha), (\gamma, \alpha), (\beta, \gamma)$

4. Consider the following Prolog program:

```
parent(may,noah).
parent(may,ari).
parent(kim,ryan).

sibling(kim,may).
cousin(X,Y) :- parent(A,X),parent(B,Y),sibling(A,B).
```

Relative to this program, give all of the answers that would be returned for the query

`cousin(X,Y).`

in the order that they would be given.

Assume the usual top-to-bottom, left-to-right evaluation strategy.

—

X = ryan, Y = noah ; X = ryan, Y = ari

5. Next, consider the following Prolog program:

```
parent(may,noah).
parent(may,ari).
parent(kim,ryan).

parent(rika,may).
parent(rika,kim).
sibling(A,B) :- parent(C,A),parent(C,B).
cousin(X,Y) :- parent(A,X),parent(B,Y),sibling(A,B).
```

Relative to this program, draw the SLD-resolution tree, *up to the point where the first answer is generated*, for the query

`cousin(X,Y).`

Indicate the first answer, clearly.

Assume the usual top-to-bottom, left-to-right evaluation strategy.

—

Many examples of SLD-resolution trees were seen in class. In this case, in dealing with `parent(A,X),parent(B,Y),sibling(A,B)`,
for each of the first two atoms, the first match will be with the first fact of the program: `parent(may,noah)`. Under the given program, it holds that `sibling(may,may)`, so the first answer will be `X = noah, Y = noah`.