1. Determinacy of IMP

Consider the following theorem and proof:

- Determinancy of heap lookup:

  \[
  \text{(For all } H, x, c_1, \text{ and } c_2) \quad \text{If } H \ll x \leadsto c_1 \text{ and } H \ll x \leadsto c_2, \text{ then } c_1 = c_2.\]

\textbf{Proof:} By structural induction on (the derivation) \( H \ll x \leadsto c_1 \).

\begin{itemize}
  \item \( H \ll x \leadsto c_1 \equiv \cdot \ll x \leadsto 0 \) :  
    \begin{itemize}
      \item Therefore, \( H = \cdot \) and \( c_1 = 0 \).
      \item From \( H \ll x \leadsto c_2 \) and \( H = \cdot \), we have \( \cdot \ll x \leadsto c_2 \).
      \item By inversion of \( \cdot \ll x \leadsto c_2 \), we have \( c_2 = 0 \).
      \item Therefore, \( c_1 = 0 = c_2 \).
    \end{itemize}
  \item \( H \ll x \leadsto c_1 \equiv H', x \mapsto c_1 \ll x \leadsto c_1 \) :  
    \begin{itemize}
      \item Therefore, \( H = H', x \mapsto c_1 \).
      \item From \( H \ll x \leadsto c_2 \) and \( H = H', x \mapsto c_1 \), we have \( H', x \mapsto c_1 \ll x \leadsto c_2 \).
      \item By inversion of \( H', x \mapsto c_1 \ll x \leadsto c_2 \), we have \( c_2 = c_1 \).
      \item Therefore, \( c_1 = c_2 \).
    \end{itemize}
  \item \( H \ll x \leadsto c_1 \equiv H', y' \mapsto c' \ll x \leadsto c_1 \) :  
    \begin{itemize}
      \item Therefore, \( H = H', y' \mapsto c' \).
      \item From \( H \ll x \leadsto c_2 \) and \( H = H', y' \mapsto c' \), we have \( H', y' \mapsto c' \ll x \leadsto c_2 \).
      \item By inversion of \( H', y' \mapsto c' \ll x \leadsto c_2 \) (with \( x \neq y' \)), we have \( H' \ll x \leadsto c_2 \).
      \item By the inductive hypothesis applied \( H' \ll x \leadsto c_1 \) with \( H' \ll x \leadsto c_2 \), we have \( c_1 = c_2 \).
      \item Therefore, \( c_1 = c_2 \).
    \end{itemize}
\end{itemize}
(a) Complete the following proof.

- Hint: Use “Determinacy of heap lookup”.

Determinacy of expressions:

(For all \(H, e, c_1,\) and \(c_2\))

\[\text{If } H; e \Downarrow c_1 \text{ and } H; e \Downarrow c_2, \text{ then } c_1 = c_2.\]

Proof: By structural induction on (the derivation) \(H; e \Downarrow c_1\).

- \(H; e \Downarrow c_1 \equiv H; c_1 \Downarrow c_1 \): \text{CONST}

\[\text{Therefore, } e = c_1.\]

***COMPLETE THE \textit{CONST} CASE.***

- \(H; e \Downarrow c_1 \equiv H; x \Downarrow c_1 \): \text{VAR}

\[\text{Therefore, } e = x \text{ and } H \oplus x \Downarrow c_1.\]

***COMPLETE THE \textit{VAR} CASE.***

- \(H; e \Downarrow c_1 \equiv H; e_a \Downarrow c_{a_1} \oplus H; e_b \Downarrow c_{b_1} \): \text{ADD}

\[\text{Therefore, } e = e_a \oplus e_b, c_1 = c_{a_1} + c_{b_1}, H; e_a \Downarrow c_{a_1}, \text{ and } H; e_b \Downarrow c_{b_1}.\]

\[\text{From } H; e \Downarrow c_2 \text{ and } e = e_a \oplus e_b, \text{ we have } H; e_a \Downarrow c_2.\]

\[\text{By inversion of } H; e_a \Downarrow c_2, \text{ we have } c_2 = c_{a_2} + c_{b_2}, H; e_a \Downarrow c_{a_2}, \text{ and } H; e_b \Downarrow c_{b_2}.\]

(Note: Inverting \(H; e_a \Downarrow c_2 \text{ yielded derivations that evaluate } e_a \text{ and } e_b \text{ to constants } c_{a_2} \text{ and } c_{b_2}.\))

***COMPLETE THE \textit{ADD} CASE.***

- \(H; e \Downarrow c_1 \equiv H; e_a \Downarrow c_{a_1} \oplus H; e_b \Downarrow c_{b_1} \): \text{MULT}

\[\text{Therefore, } e = e_a \pl e_b, c_1 = c_{a_1} \ast c_{b_1}, H; e_a \Downarrow c_{a_1}, \text{ and } H; e_b \Downarrow c_{b_1}.\]

***COMPLETE THE \textit{MULT} CASE.***
(b) Complete the following proof.

- Hint: Use “Determinacy of expressions”.

Determinacy of statements:

(For all $H, s, H_1', s_1', H_2', s_2'$, and $s_2'$)

If $H; s \rightarrow H_1' ; s_1'$ and $H; s \rightarrow H_2' ; s_2'$, then $H_1' = H_2'$ and $s_1' = s_2'$.

Proof: By structural induction on (the derivation) $H; s \rightarrow H_1' ; s_1'$.

- $H; s \rightarrow H_1' ; s_1' \equiv H; \mathsf{e} \downarrow c_1 \quad \text{ assigns}$

  Therefore, $s = x \odot e \rightarrow H, x \rightarrow c_1; \mathsf{skip}$,

  and $H; e \downarrow c_1$.

  ***COMPLETE THE assigns CASE.***

- $H; s \rightarrow H_1' ; s_1' \equiv H; \mathsf{while e} \ s_w \rightarrow H; \mathsf{if e} (s_w; \mathsf{while e} \ s_w) \mathsf{skip}$

  Therefore, $s = \mathsf{while e} \ s_w, H_1' = H$, and $s_1' = \mathsf{if e} (s_w; \mathsf{while e} \ s_w) \mathsf{skip}$.

  ***COMPLETE THE while CASE.***

- $H; s \rightarrow H_1' ; s_1' \equiv H; \mathsf{skip} ; s_b \rightarrow H; s_b$.

  Therefore, $s = \mathsf{skip} ; s_b, H_1' = H, s_1' = s_b$.

  ***COMPLETE THE skip CASE.***

- $H; s_a \rightarrow H_1' ; s_a_1$ \quad \text{ seqStep}

  Therefore, $s = s_a ; s_b, s_1' = s_a_1 ; s_b$, and $H; s_a \rightarrow H_1' ; s_a_1$.

  From $H; s \rightarrow H_2' ; s_2'$ and $s = s_a ; s_b$, we have $H; s_a ; s_b \rightarrow H_2' ; s_2'$.

  By inversion of $H; s_a ; s_b \rightarrow H_2' ; s_2'$, we have two cases:

  - $H; s_a ; s_b \rightarrow H_2' ; s_2' \equiv H; \mathsf{skip} ; s_b \rightarrow H; s_b$.

    Therefore, $s_a = \mathsf{skip}, H_2' = H$, and $s_2' = s_b$.

    From $H; s_a \rightarrow H_1' ; s_a_1$ and $s_a = \mathsf{skip}$, we have $H; \mathsf{skip} \rightarrow H_1' ; s_a_1$.

    But, $H; \mathsf{skip} 

    

    ***COMPLETE THE seqStep/seqStep CASE.***

(continued)
\[ H;e \downarrow c_1 \quad c_1 > 0 \quad \text{wT} \]

- \( H; s \rightarrow H'_1; s'_1 = H; i f \ e \ s_t \ s_f \rightarrow H; s_t \) :
  Therefore, \( s = i f \ e \ s_t \ s_f, H'_1 = s_t, H; e \downarrow c_1, \) and \( c_1 > 0 \).
From \( H; s \rightarrow H'_2; s'_2 \) and \( s = i f \ e \ s_t \ s_f \), we have \( H; i f \ e \ s_t \ s_f \rightarrow H'_2; s'_2 \).
By inversion of \( H; i f \ e \ s_t \ s_f \rightarrow H'_2; s'_2 \), we have two cases:

\[ H; e \downarrow c_2 \quad c_2 > 0 \quad \text{wT} \]

- \( H; i f \ e \ s_t \ s_f \rightarrow H'_2; s'_2 \equiv H; i f \ e \ s_t \ s_f \rightarrow H; s_t \) :
  Therefore, \( H'_2 = H, s'_2 = s_t, H; e \downarrow c_2, \) and \( c_2 > 0 \).
  ***COMPLETE THE \textit{ifT/IF T} CASE.***

\[ H; e \downarrow c_2 \quad c_2 \leq 0 \quad \text{wF} \]

- \( H; i f \ e \ s_t \ s_f \rightarrow H'_2; s'_2 \equiv H; i f \ e \ s_t \ s_f \rightarrow H; s_f \) :
  Therefore, \( H'_2 = H, s'_2 = s_f, H; e \downarrow c_2, \) and \( c_2 \leq 0 \).
  ***COMPLETE THE \textit{IF T/IF F} CASE.***

  Hint: Use “Determinacy of expressions” to show that this case is impossible.

\[ H; e \downarrow c_1 \quad c_1 \leq 0 \quad \text{wF} \]

- \( H; s \rightarrow H'_1; s'_1 = H; i f \ e \ s_t \ s_f \rightarrow H; s_f \) :
  Therefore, \( s = i f \ e \ s_t \ s_f, H'_1 = s_f, H; e \downarrow c_1, \) and \( c_1 \leq 0 \).
  ***COMPLETE THE \textit{IF F} CASE.***
2. PM: A Pattern Matching Language

This problem considers a small “language” for ML-style pattern matching, with some twists. A “program” is a pattern \( p \) and a value \( v \). If \( p \) “matches” \( v \), then the result is a list of bindings \( b \). If \( p \) does not “match” \( v \), then there is no result.

Syntax definition:

\[
\begin{align*}
    v & ::= c \mid (v,v) \mid s(v) \\
    p & ::= _ \mid x \mid c \mid (p,p) \mid s(p) \mid \ldots (p) \\
    b & ::= _ \mid b, x \mapsto v \\
    (c & \in \{-2,-1,0,1,2,\ldots\}) \\
    (s & \text{ any non-empty string of letters}) \\
    (x & \text{ any non-empty string of letters})
\end{align*}
\]

For values, we have constants, pairs, and tagged values. (The tag is any non-empty string, which is unlike ML where constructors must be declared in preceding \texttt{datatype} declarations.) For patterns, we have wildcard, variables, constants, pairs, tagged patterns, and the “descendent” pattern \( \ldots (p) \).

Informal semantics:

- Pattern _ matches every value and produces the empty list of bindings (_).
- Pattern \( x \) matches every value and produces the one-element binding list \( _ \), \( x \mapsto v \) when matched with \( v \). Note \( x \) can be any variable.
- Pattern \( c \) matches only the value that is the same constant and produces the empty list of bindings.
- Pattern \( (p_1, p_2) \) matches only pairs of values and only if \( p_1 \) and \( p_2 \) match the corresponding components of the pair. The result is the two binding lists from the nested matches appended together.
- Pattern \( s(p) \) matches only a tagged value where the tag is the same (i.e., the same string \( s \)) and \( p \) matches the corresponding value. The result is the result of the nested match.
- Pattern \( \ldots (p) \) matches a value \( v \) if \( p \) matches any descendent of \( v \) in the abstract syntax tree, \textit{including} \( v \) itself. Put another (very useful) way, it matches if \( p \) matches \( v \) or if \( \ldots (p) \) matches a child of \( v \) in the abstract syntax tree. The result is the result of (any) match that leads to “success”.
- Assume a pattern does not have the same variable more than once; you do not need to check for this.

Example: using the concrete syntax for the scanner/parser provided (note parentheses are necessary, the pattern and value must be on separate lines, and there can be no line breaks within the pattern or value):

\[
\begin{align*}
    \text{bar} & ((x, (\ldots ((18, z)), _))) \\
    \text{bar} & ((42, (\text{foo} ((17, (18, (0, 20)))), 19)))
\end{align*}
\]

The only match produces a binding list where \( x \) maps to 42 and \( z \) maps to \((0, 20)\).
(a) (Formal large-step operational semantics) Give a formal large-step operational semantics for pattern-matching. Your judgment should have the form $p;v \Downarrow b$, meaning $p$ matches $v$ producing $b$. If $p$ does not match $v$ there must be no derivation for any $b$. Hints:
   - Have 9 inference rules, 3 of which are axioms.
   - You need multiple rules for descendent patterns.
   - Write $b_1 @ b_2$ for the result of appending $b_1$ and $b_2$. (This arises only once; do not worry about formalizing append.)

(b) Give a $p$ and $v$ where multiple $b$ are possible. That is, show the large-step semantics is nondeterministic.

(c) (SML warm-up) Implement `valueToString` of type `Ast.value -> string` for converting values to concrete syntax. Implement `bindingListToString` of type `((string * Ast.value) list -> string)` for converting a binding list to a string. The actual string is unimportant; we recommend putting each binding on a separate line and putting a “;” between the variable and the value. Note that the provided `printAns` uses `bindingListToString`.

(d) (SML large-step operational semantics) Implement `large` of type `Ast.pattern * Ast.value -> (string * Ast.value) list option` to implement pattern-matching. Your code should largely correspond to your inference rules (Hint: use a `case` match on the pair $(p,v)$.) with these differences:
   - Return `NONE` if there is no match and `SOME b` if there is a match with binding-list $b$.
   - You may resolve the nondeterminism in any manner you like, i.e., if there is more than one match your implementation should just “find one” and return it. Your implementation must always produce one if there is one.

(e) (Formal small-step operational semantics) Give a formal small-step operational semantics for pattern-matching. Your judgment should have the form $p;v;\cdot \rightarrow * \_;v';b'$. Pattern-matching is “done” when $p$ is $\_$. Otherwise, if $p$ matches $v$ there should be a rule that simplifies $p$ or $v$ or both by turning them into $p'$ and $v'$. The binding list $b'$ is either $b$ or something added onto $b$. A result for the “whole program” $p$ and $v$ is a $b'$ where $p;v; \cdot \rightarrow * \_;v';b'$. If $p$ and $v$ do not match, there must be no way to derive $p;v; \cdot \rightarrow * \_;v';b'$. Hints:
   - Have 9 inference rules, 8 of which are axioms.
   - For the nonaxiom, “simplify” the “left side” of a pair-match. 1 axiom rule simplifies a pair-match whose left side is the $\_$. pattern.
   - Almost all the axioms produce the same $b$ they start with.
   - A couple axioms will turn a pattern into $\_$. This is similar to IMP’s assign rule where we turn an assignment into a skip.
   - 1 axiom simplifies a tag-match (if the tags match) by just “stripping off the tags”.

Note: These “hints” are perhaps more for “checking your work” than guiding you.

(f) Explain the difficulty that would arise when turning the small-step operational semantics into an ML interpreter, which takes a program consisting of a pattern $p$ and a value $v$ and returns a list of bindings $b$ if and only if $p;v; \cdot \rightarrow * \_;v';b$. 
Debriefing

- How many hours did you spend on this assignment?
- Would you rate it as easy, moderate, or difficult?
- How deeply do you feel you understand the material it covers (0% – 100%)?
- If you have any other comments about the assignment, then please include them with your submission or send email to mtf@cs.rit.edu.

Submission

Programming components of the assignment:

- problems 2c and 2d

must be submitted to the Homework 3 Assignment on MyCourses by the due date.

Written components of the assignment:

- problem 1
- problems 2a, 2b, 2e, and 2f
- debriefing

must either be submitted to the instructor in class or submitted as a PDF file to the Homework 3 Assignment on MyCourses by the due date.

Document History

September 18, 2019
Original version