CS477 Formal Software Development Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides based in part on previous lectures by Mahesh Vishwanathan, and by Gul Agha

March 21, 2014
Simple Imperative Programming Language #2

\[ I \in Identifiers \]
\[ N \in Numerals \]
\[ E ::= N \mid I \mid E + E \mid E \ast E \mid E - E \mid I ::= E \]
\[ B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \]
\[ \mid E < E \mid E = E \]
\[ C ::= \text{skip} \mid C; C \mid \{C\} \mid E \]
\[ \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \]
\[ \mid \text{while } B \text{ do } C \]
Transition Semantics

- Aka “small step semantics” or “Structured Operational Semantics”
- Defines a relation of “one step” of computation, instead of complete evaluation
  - Determines granularity of atomic computations
- Typically have two kinds of “result”: configurations and final values
- Written \((C, m) \rightarrow (C', m')\) or \((C, m) \rightarrow m'\)
I \in \text{Identifiers}

N \in \text{Numerals}

E ::= N \mid I \mid E + E \mid E \cdot E \mid E - E

B ::= \text{true} \mid \text{false} \mid B \& B \mid B \lor B \mid \text{not } B

\mid E < E \mid E = E

C ::= \text{skip} \mid C; C \mid \{C\} \mid I ::= E

\mid \text{if } B \text{ then } C \text{ else } C \text{ fi}

\mid \text{while } B \text{ do } C
skip means done evaluating

When evaluating an assignment, evaluate expression first

If the expression being assigned is a value, update the memory with the new value for the identifier

When evaluating a sequence, work on the first command in the sequence first

If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory
Commands

Skip: \((\text{skip}, m) \rightarrow m\)

Assignment: \[
\begin{align*}
(E, m) &\rightarrow (E', m) \\
(l ::= E, m) &\rightarrow (l ::= E', m) \\
(l ::= V, m) &\rightarrow m[l \leftarrow V]
\end{align*}
\]

Sequencing:
\[
\begin{align*}
(C, m) &\rightarrow (C'', m') \\
(C; C', m) &\rightarrow (C''; C', m') \\
(C, m) &\rightarrow m' \\
(C; C', m) &\rightarrow (C', m')
\end{align*}
\]
Choice of level of granularity:

- Choice 1: Open a block is a unit of work
  
  \[
  (\{C\}, m) \rightarrow (C, m)
  \]

- Choice 2: Blocks are syntactic sugar
  
  \[
  (C, m) \rightarrow (C', m') \\
  \{C\}, m) \rightarrow (C', m') \\
  (C, m) \rightarrow m' \\
  \{C\}, m) \rightarrow m'
  \]
If the boolean guard in an `if_then_else` is true, then evaluate the first branch.
If it is false, evaluate the second branch.
If the boolean guard is not a value, then start by evaluating it first.
If Then Else Command

(if true then \( C \) else \( C' \) fi, \( m \)) \( \rightarrow \) \((C, m)\)

(if false then \( C \) else \( C' \) fi, \( m \)) \( \rightarrow \) \((C', m)\)

\((B, m) \rightarrow (B', m)\)

(if \( B \) then \( C \) else \( C' \) fi, \( m \)) \( \rightarrow \) (if \( B' \) then \( C \) else \( C' \) fi, \( m \))
(while $B$ do $C$, $m$) 

$\rightarrow$

(if $B$ then $C$; while $B$ do $C$ else skip fi, $m$)

In English: Expand a while into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.
(\(y := i; \text{ while } i > 0 \text{ do } \{i := i - 1; y := y * i\}, \langle i \mapsto 3 \rangle\))

\[\rightarrow \_\_\_\_\_\_\_\_\_\_\_\]
Alternate Semantics for SIMPL1

- Can mix Natural Semantics with Transition Semantics to get larger atomic computations
- Use \((E, m) \downarrow v\) and \((B, m) \downarrow b\) for arithmetics and boolean expressions
- Revise rules for commands
Revised Rules for SIMPL1

Skip: \[(\text{skip}, m) \rightarrow m\]

Assignment: \[
\frac{(E, m) \downdownarrows v}{(l ::= E, m)} \rightarrow m[l \leftarrow V]\]

Sequencing:
\[
\frac{(C, m) \rightarrow (C'', m')}{(C; C', m) \rightarrow (C''; C', m')} \quad \frac{(C, m) \rightarrow m'}{(C; C', m) \rightarrow (C', m')}\]

Blocks:
\[
\frac{(C, m) \rightarrow (C', m')}{\{C\}, m) \rightarrow (C', m')} \quad \frac{(C, m) \rightarrow m'}{\{C\}, m) \rightarrow m'}\]

If Then Else Command

\[
(B, m) \downarrow \text{true} \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \rightarrow (C, m)
\]

\[
(B, m) \downarrow \text{false} \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \rightarrow (C', m)
\]
While Command

\[
(B, m) \Downarrow \text{true} \\
(\text{while } B \text{ do } C, m) \rightarrow (C; \text{while } B \text{ do } C, m)
\]

\[
(B, m) \Downarrow \text{false} \\
(\text{while } B \text{ do } C, m) \rightarrow m
\]

- Other more fine grained options exist (eg rule given before)
Transition Semantics for SIMPL2?

- What are the choices and consequences for giving a transition semantics for the Simple Concurrent Imperative Programming Language #2, SIMP2?
- For finest grain transitions, summary:
  - Each rule for arithmetic or boolean expression must propagate changes to memory; instead of transitioning to a value, go to a value - memory pair
Transition Semantics for SIMPL2

- Second assignment rule returns value:

\[(I ::= V, m) \rightarrow (V, m[I ← V])\]

- Expressions as commands need two rules:

\[(E, m) \rightarrow (E', m')\]
\[(E, m) \rightarrow (V, m')\]
\[(E, m) \rightarrow (E', m')\]
\[(E, m) \rightarrow m'\]

Exp. as Comm.:

\[(E, m) \rightarrow (E', m')\]
\[(E, m) \rightarrow (E', m)\]
Identifiers
N \in \text{Numerals}
E ::= N \mid I \mid E + E \mid E \ast E \mid E - E
B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B
\mid E < E \mid E = E
C ::= \text{skip} \mid C; C \mid \{C\} \mid I ::= E \mid C \parallel C'
\mid \text{if } B \text{ then } C \text{ else } C \text{ fi}
\mid \text{while } B \text{ do } C
Semantics for $C_1 || C_2$ means that the actions of $C_1$ and done at the same time as, “in parallel” with, those of $C_2$

True parallelism hard to model; must handle collisions on resources

What is the meaning of

$$x := 1 || x := 0$$

True parallelism exists in real world, so important to model correctly
Weaker alternative: interleaving semantics

Each process gets a turn to commit some atomic steps; no preset order of turns, no preset number of actions

No collision for \( x := 1 \parallel x := 0 \)
- Yields only \( \langle x \mapsto 1 \rangle \) and \( \langle x \mapsto 0 \rangle \); no collision

No simultaneous substitution: \( x := y \parallel y := x \) results in \( x \) and \( y \) having the same value; not in swapping their values.
Coarse-Grained Interleaving Semantics for SCIMPL1 Commands

- Skip, Assignment, Sequencing, Blocks, If_Then_Else, While unchanged
- Need rules for $\parallel$

\[
\frac{(C_1, m) \rightarrow (C'_1, m')}{(C_1 \parallel C_2, m) \rightarrow (C'_1 \parallel C_2, m')} \\
\frac{(C_1, m) \rightarrow m'}{(C_1 \parallel C_2, m) \rightarrow (C_2, m')} \\
\frac{(C_2, m) \rightarrow (C'_2, m')}{(C_1 \parallel C_2, m) \rightarrow (C_1 \parallel C'_2, m')} \\
\frac{(C_2, m) \rightarrow m'}{(C_1 \parallel C_2, m) \rightarrow (C_1, m')}
\]
Simple Concurrent Imperative Programming Language #2 (SCIMP2)

\[ I \in \text{Identifiers} \]

\[ N \in \text{Numerals} \]

\[ E ::= N \mid I \mid E + E \mid E \times E \mid E - E \]

\[ B ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \]

\[ | E < E \mid E = E \]

\[ C ::= \text{skip} \mid C; C \mid \{ C \} \mid I ::= E \mid C || C' \mid \text{sync}(E) \]

\[ | \text{if } B \text{ then } C \text{ else } C \text{ fi} \]

\[ | \text{while } B \text{ do } C \]
Informal Semantics of sync

- \( \text{sync}(E) \) evaluates \( E \) to a value \( v \)
- Waits for another parallel command waiting to synchronize on \( v \)
- When two parallel commands are both waiting to synchronize on a value \( v \), they may both stop waiting, move past the synchronization, and carry on with whatever commands they each have left
- Only two processes may synchronize at a time (in this version).
- Problem: How to formalize?
A labeled transition system (LTS) is a 4-tuple \((Q, \Sigma, \delta, I)\) where

- \(Q\): set of states
  - \(Q\) finite or countably infinite

- \(\Sigma\): set of labels (aka actions)
  - \(\Sigma\) finite or countably infinite

- \(\delta\) \(\subseteq Q \times \Sigma \times Q\) transition relation

- \(I\) \(\subseteq Q\) initial states

Note: Write \(q \xrightarrow{\alpha} q'\) for \((q, \alpha, q') \in \delta\).
Example: Candy Machine

\[ Q = \{ \text{Start}, \text{Select}, \text{GetMarsBar}, \text{GetKitKatBar} \} \]
\[ I = \{ \text{Start} \} \]
\[ \Sigma = \{ \text{Pay}, \text{ChooseMarsBar}, \text{ChooseKitKatBar}, \text{TakeCandy} \} \]

\[ \delta = \left\{ \begin{array}{l}
(\text{Start, Pay, Select}) \\
(\text{Select, ChooseMarsBar, GetMarsBar}) \\
(\text{Select, ChooseKitKatBar, GetKitKatBar}) \\
(\text{GetMarsBar, TakeCandy, Start}) \\
(\text{GetKitKatBar, TakeCandy, Start})
\end{array} \right\} \]