Expressions with Side-Effects
Blocking Semantics

Claude Marché

Cours MPRI 2-36-1 “Preuve de Programme”

19 décembre 2012
Exercise 3

Let’s assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of x by 2 and its remainder. The following program is supposed to compute, in variable \( r \), the power \( x^n \).

\[
\begin{align*}
    r & := 1; \ p := x; \ e := n; \\
    \text{while} \ e > 0 \ \text{do} \\
    \hspace{1em} & \text{if} \ \text{mod2}(e) \neq 0 \ \text{then} \ r := r \times p; \\
    \hspace{1em} & p := p \times p; \\
    \hspace{1em} & e := \text{div2}(e); \\
\end{align*}
\]

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.
Exercise 4

The Fibonacci sequence is defined recursively by $\text{fib}(0) = 0$, $\text{fib}(1) = 1$ and $\text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n)$. The following program is supposed to compute $\text{fib}$ in linear time, the result being stored in $y$.

\[
y := 0; \quad x := 1; \quad i := 0;
\]

\[
\text{while } i < n \text{ do}
\]

\[
\text{aux} := y; \quad y := x; \quad x := x + \text{aux}; \quad i := i + 1
\]

- Assuming $\text{fib}$ exists in the logic, specify appropriate pre- and post-conditions.

- Prove the program.
Reminder of the last lecture

- Very simple programming language
  - program = sequence of statements
  - only global variables
  - only the integer data type, always well typed
- Formal operational semantics
  - small steps
  - no run-time errors
- Hoare logic:
  - Deduction rules for triples $\{\text{Pre}\} s \{\text{Post}\}$
- Weakest Liberal Precondition (WLP):
  - if $\text{Pre} \Rightarrow \text{WLP}(s, \text{Post})$ then $\{\text{Pre}\} s \{\text{Post}\}$ valid
- In lecture notes: extensions for termination
  - Total correctness of triples
  - Weakest (Strict) Precondition
This Lecture’s Goals

Extend the language
- more data types
- *logic variables*: local and *immutable*
- *labels* in specifications

Handle termination issues:
- prove properties on non-terminating programs
- prove termination when wanted

Prepare for adding later:
- run-time errors (how to prove their absence)
- local *mutable* variables, functions
- complex data types
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Extended Syntax: Generalities

- We want a few basic data types: int, bool, real, unit
- Former pure expressions are now called terms
- No difference between expressions and statements anymore

<table>
<thead>
<tr>
<th>last lecture</th>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>term</td>
</tr>
<tr>
<td>formula</td>
<td>formula</td>
</tr>
<tr>
<td>statement</td>
<td>expression</td>
</tr>
</tbody>
</table>

Basically we consider

- A purely functional language (ML-like)
- with global mutable variables

very restricted notion of modification of program states
Base Data Types, Operators, Terms

- unit type: type `unit`, only one constant ()
- Booleans: type `bool`, constants `True, False`, operators `and, or, not`
- integers: type `int`, operators `+, −, ∗` (no division)
- reals: type `real`, operators `+, −, ∗` (no division)
- Comparisons of integers or reals, returning a boolean
- “if-expression”: written `if b then t₁ else t₂`

\[
  t ::= \text{val} \quad \begin{array}{l}
  \text{(values, i.e. constants)} \\
  \text{v} \quad \text{(logic variables)} \\
  x \quad \text{(program variables)} \\
  t \text{ op } t \quad \text{(binary operations)} \\
  \text{if } t \text{ then } t \text{ else } t \quad \text{(if-expression)}
\]
Local logic variables

We extend the syntax of terms by

\[ t ::= \text{let } v = t \text{ in } t \]

Example: approximated cosine

```plaintext
let cos_x =
    let y = x*x in
    1.0 - 0.5 * y + 0.04166666 * y * y
in
...
```
Practical Notes

- Theorem provers (Alt-Ergo, CVC3, Z3) typically support these types
- may also support if-expressions and let bindings

Alternatively, Why3 manages to transform terms and formulas when needed (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas)
Syntax: Formulas

Unchanged w.r.t to last lecture, but also addition of local binding:

\[
p ::= t \quad \text{(boolean term)}
\]
\[
| \quad p \land p \mid p \lor p \mid \neg p \mid p \Rightarrow p \quad \text{(connectives)}
\]
\[
| \quad \forall v : \tau, \ p \mid \exists v : \tau, \ p \quad \text{(quantification)}
\]
\[
| \quad \text{let } v = t \text{ in } p \quad \text{(local binding)}
\]
Typing

- Types:
  \[ \tau ::= \text{int} \mid \text{real} \mid \text{bool} \mid \text{unit} \]

- Typing judgment:
  \[ \Gamma \vdash t : \tau \]

where \( \Gamma \) maps identifiers to types:

- either \( v : \tau \) (logic variable, immutable)
- either \( x : \text{ref} \tau \) (program variable, mutable)

Important

- a reference is not a value
- there is no “reference on a reference”
- no aliasing
Typing rules

Constants:

\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash r : \text{real} \]

\[ \Gamma \vdash \text{True} : \text{bool} \quad \Gamma \vdash \text{False} : \text{bool} \]

Variables:

\[ \nu : \tau \in \Gamma \quad x : \text{ref} \tau \in \Gamma \]

\[ \Gamma \vdash \nu : \tau \quad \Gamma \vdash x : \tau \]

Let binding:

\[ \Gamma \vdash t_1 : \tau_1 \quad \{ \nu : \tau_1 \} \cdot \Gamma \vdash t_2 : \tau_2 \]

\[ \Gamma \vdash \text{let} \ \nu = t_1 \ \text{in} \ t_2 : \tau_2 \]

- All terms have a base type (not a reference)
- In practice: Why3, as in OCaml, requires to write \( !x \) for references
Program states are augmented with a stack of local (immutable) variables

- $\Sigma$: maps program variables to values (a map)
- $\Pi$: maps logic variables to values (a stack)

$$\begin{align*}
[\text{val}]_{\Sigma,\Pi} &= \text{val} & \text{(values)} \\
[x]_{\Sigma,\Pi} &= \Sigma(x) & \text{if } x : \text{ref } \tau \\
[v]_{\Sigma,\Pi} &= \Pi(v) & \text{if } v : \tau \\
[t_1 \text{ op } t_2]_{\Sigma,\Pi} &= [t_1]_{\Sigma,\Pi} \text{ op } [t_2]_{\Sigma,\Pi} \\
[\text{let } v = t_1 \text{ in } t_2]_{\Sigma,\Pi} &= [t_2]_{\Sigma,\Pi} \{(v = [t_1]_{\Sigma,\Pi}) : \Pi\}
\end{align*}$$

**Warning**

Semantics is now a partial function
Type Soundness Property

Our logic language satisfies the following standard property of purely functional language

**Theorem (Type soundness)**

*Every well-typed terms and well-typed formulas have a semantics*

Proof: induction on the derivation tree of well-typing
Expressions: generalities

- Former statements are now expressions of type unit
  Expressions may have Side Effects
- Statement `skip` is identified with `()`
- The sequence is replaced by a local binding
- From now on, the condition of the `if then else` and the `while do` in programs is a Boolean expression
Syntax

\[ e ::= t \quad \text{(pure term)} \]
\[ | \quad e \ op \ e \quad \text{(binary operation)} \]
\[ | \quad x ::= e \quad \text{(assignment)} \]
\[ | \quad \text{let } v = e \text{ in } e \quad \text{(local binding)} \]
\[ | \quad \text{if } e \text{ then } e \text{ else } e \quad \text{(conditional)} \]
\[ | \quad \text{while } e \text{ do } e \quad \text{(loop)} \]

- sequence \( e_1; e_2 \) : syntactic sugar for

\[
\text{let } v = e_1 \text{ in } e_2
\]

when \( e_1 \) has type \text{unit} and \( v \) not used in \( e_2 \)
Toy Examples

\[
z := \text{if } x \geq y \text{ then } x \text{ else } y
\]

\[
\text{let } v = r \text{ in } (r := v + 42; v)
\]

\[
\text{while } (x := x - 1; x > 0) \text{ do ()}
\]

\[
\text{while } (\text{let } v = x \text{ in } x := x - 1; v > 0) \text{ do ()}
\]
Typing Rules for Expressions

Assignment:
\[ \frac{\tau \in \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \text{unit}} \]

Let binding:
\[ \frac{\Gamma \vdash e_1 : \tau_1 \quad \{ v : \tau_1 \} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } v = e_1 \text{ in } e_2 : \tau_2} \]

Conditional:
\[ \frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } c \text{ then } e_1 \text{ else } e_2 : \tau} \]

Loop:
\[ \frac{\Gamma \vdash c : \text{bool} \quad \Gamma \vdash e : \text{unit}}{\Gamma \vdash \text{while } c \text{ do } e : \text{unit}} \]
Operational Semantics

Novelties

- Need for *context rules*
- Precise the order of evaluation: left to right

- One-step execution has the form

  \[ \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \]

- Values do not reduce
Operational Semantics

▶ Assignment

\[
\Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \quad \frac{\Sigma, \Pi, x := e \leadsto \Sigma', \Pi', e'}{\Sigma, \Pi, x := val \leadsto \Sigma[x \leftarrow \text{val}], \Pi, ()}
\]

▶ Let binding

\[
\Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e'_1 \quad \frac{\Sigma, \Pi, \text{let } v = e_1 \text{ in } e_2 \leadsto \Sigma', \Pi', \text{let } v = e'_1 \text{ in } e_2}{\Sigma, \Pi, \text{let } v = val \text{ in } e \leadsto \Sigma, \{v = \text{val}\} \cdot \Pi, e}
\]
Operational Semantics, Continued

- Binary operations

\[
\begin{align*}
\Sigma, \Pi, e_1 & \leadsto \Sigma', \Pi', e'_1 \\
\Sigma, \Pi, e_1 + e_2 & \leadsto \Sigma', \Pi', e'_1 + e_2 \\
\Sigma, \Pi, e_2 & \leadsto \Sigma', \Pi', e'_2 \\
\Sigma, \Pi, \text{val}_1 + e_2 & \leadsto \Sigma', \Pi', \text{val}_1 + e'_2 \\
\text{val} & = \text{val}_1 + \text{val}_2 \\
\Sigma, \Pi, \text{val}_1 + \text{val}_2 & \leadsto \Sigma, \Pi, \text{val}
\end{align*}
\]
Operational Semantics, Continued

- **Conditional**

\[
\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c' \\
\Sigma, \Pi, \text{if } c \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma', \Pi', \text{if } c' \text{ then } e_1 \text{ else } e_2
\]

\[
\Sigma, \Pi, \text{if } \text{True} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1
\]

\[
\Sigma, \Pi, \text{if } \text{False} \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2
\]

- **Loop**

\[
\Sigma, \Pi, \text{while } c \text{ do } e \rightsquigarrow \\
\Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ do } e) \text{ else } ()
\]
Remark: most of the context rules can be avoided

- An equivalent operational semantics can be defined using `let v = ... in ...` instead, e.g.:

\[
\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2
\]

- Thus, only the context rule for `let` is needed
Type Soundness

Theorem

Every well-typed expression evaluate to a value or execute infinitely

Classical proof:

- type is preserved by reduction
- execution of well-typed expressions that are not values can progress
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Blocking Semantics: General Ideas

- add *assertions* in expressions
- failed assertions = “run-time errors”

First step: modify expression syntax with

- new expression: assertion
- adding loop invariant in loops

\[ e ::= \text{assert } p \quad \text{(assertion)} \]
\[ \quad \mid \text{while } e \text{ invariant } / \text{do } e \quad \text{(annotated loop)} \]
Toy Examples

\[ z := \text{if } x \geq y \text{ then } x \text{ else } y ; \]
\[ \text{assert } z \geq x \land z \geq y \]

while \((x := x - 1; x > 0)\)
    invariant \(x \geq 0\) do ();
\[ \text{assert } (x = 0) \]

while \((\text{let } v = x \text{ in } x := x - 1; v > 0)\)
    invariant \(x \geq -1\) do ();
\[ \text{assert } (x < 0) \]
Blocking Semantics: Modified Rules

\[
[P]_{\Sigma, \Pi} \text{ holds}
\]
\[
\Sigma, \Pi, \text{assert } P \rightsquigarrow \Sigma, \Pi, ()
\]

\[
[I]_{\Sigma, \Pi} \text{ holds}
\]
\[
\Sigma, \Pi, \text{while } c \text{ invariant } l \text{ do } e \rightsquigarrow
\]
\[
\Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } l \text{ do } e) \text{ else } ()
\]

Important

Execution blocks as soon as an invalid annotation is met
Soundness of a program

Definition

Execution of an expression in a given state is \textit{safe} if it does not block: either terminates on a value or runs infinitely.

Definition

A triple \( \{ P \} e \{ Q \} \) is valid if for any state \( \Sigma, \Pi \) satisfying \( P \), \( e \) \textit{executes safely} in \( \Sigma, \Pi \), and if it terminates, the final state satisfies \( Q \).

New addition in the specification language:

- keyword \textit{result} in post-conditions
- denotes the value of the expression executed
Toy Examples, Continued

{ true }
if \( x \geq y \) then \( x \) else \( y \)
{ result \geq x \land result \geq y \}

{ x \geq 0 }
c := 0; sum := 1;
while sum \leq x \ do
  c := c + 1; sum := sum + 2 \ast c + 1
done;
c
{ result \geq 0 \land
  result \ast result \leq x < (result+1)\ast (result+1) \}
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Weakest Preconditions Revisited

Goal:

- construct a new calculus $\text{WP}(e, Q)
- expected property: in any state satisfying $\text{WP}(e, Q)$, $e$ is guaranteed to execute safely

Remark:

- Stating this for $Q = \text{true}$ is enough to ensure safety
- But need to state this for any $Q$ to prove soundness (by induction)
New Weakest Precondition Calculus

- Pure terms:
  \[ WP(t, Q) = Q[result \leftarrow t] \]

- Let binding:
  \[
  WP(\text{let } x = e_1 \text{ in } e_2, Q) = \\
  WP(e_1, WP(e_2, Q)[x \leftarrow result])
  \]
Weakest Preconditions, continued

- **Assignment:**

\[
WP(x := e, Q) = WP(e, Q[result ← ()]; x ← result))
\]

- **Alternative:**

\[
\begin{align*}
WP(x := e, Q) &= WP(let \ v = e \ in \ x := v, Q) \\
WP(x := t, Q) &= Q[result ← ()]; x ← t])
\end{align*}
\]
WP: Exercise

\[
\text{WP}(\text{let } v = x \text{ in } (x := x + 1; v), x > \text{result}) = ?
\]
Weakest Preconditions, continued

- Conditional

\[
WP(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = WP(e_1, if \ \text{result} \ \text{then } WP(e_2, Q) \ \text{else } WP(e_3, Q))
\]

- Alternative with let: (exercise!)
Weakest Preconditions, continued

- Assertion

\[ WP(\text{assert } P, Q) = P \land Q = P \land (P \Rightarrow Q) \]

(second version useful in practice)

- While loop

\[
WP(\text{while } c \text{ invariant } I \text{ do } e, Q) = \\
I \land \\
\forall \vec{v}, (I \Rightarrow WP(c, \text{if result then } WP(e, I) \text{ else } Q))[w_i \leftarrow v_i]
\]

where \( w_1, \ldots, w_k \) is the set of assigned variables in expressions \( c \) and \( e \) and \( v_1, \ldots, v_k \) are fresh logic variables
General Properties of $WP$

**Lemma (Monotonicity)**

If $\vdash P \Rightarrow Q$ then $\vdash WP(e, P) \Rightarrow WP(e, Q)$

Proof: structural induction on $e$

Remark: true only when quantified on *all states*

**Lemma (Conjunction Property)**

If $\Sigma, \Pi \vdash WP(e, P)$ and $\Sigma, \Pi \vdash WP(e, Q)$ then $\Sigma, \Pi \vdash WP(e, P \land Q)$

Proof: structural induction on $e$
Soundness of WP

Lemma (Preservation by Reduction)

If $\Sigma, \Pi \models WP(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$ then $\Sigma', \Pi' \models WP(e', Q)$

Proof: predicate induction of $\rightsquigarrow$.

Lemma (Progress)

If $\Sigma, \Pi \models WP(e, Q)$ and $e$ is not a value then there exists $\Sigma', \Pi, e'$ such that $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$

Proof: structural induction of $e$.

Corollary (Soundness)

If $\Sigma, \Pi \models WP(e, Q)$ then $e$ executes safely in $\Sigma, \Pi$. 
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Labels: Syntax and Typing

Add in syntax of \textit{terms}:

\[ t ::= x@L \quad \text{(labeled variable access)} \]

Add in syntax of \textit{expressions}:

\[ e ::= L : e \quad \text{(labeled expressions)} \]

Typing:

- only mutable variables can be accessed through a label
- labels must be declared before use

Implicit labels:

- Here, available in every formula
- Old, available in post-conditions
Toy Examples, Continued

{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 ∧ result = r@Old }

{ true }
let tmp = x in x := y; y := tmp
{ x = y@Old ∧ y = x@Old }

SUM revisited:

{ y ≥ 0 }
L:
while y > 0 do
    invariant { x + y = x@L + y@L }
x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
Labels: Operational Semantics

Program state

- becomes a collection of maps indexed by labels
- value of variable $x$ at label $L$ is denoted $\Sigma(x, L)$

New semantics of variables in terms:

\[
\begin{align*}
[x]_{\Sigma, \Pi} &= \Sigma(x, \text{Here}) \\
[x@L]_{\Sigma, \Pi} &= \Sigma(x, L)
\end{align*}
\]

The operational semantics of expressions is modified as follows

\[
\begin{align*}
\Sigma, \Pi, x := \text{val} & \mapsto \Sigma\{(x, \text{Here}) \leftarrow \text{val}\}, \Pi, () \\
\Sigma, \Pi, L : e & \mapsto \Sigma\{(x, L) \leftarrow \Sigma(x, \text{Here}) \mid x \text{ any variable}\}, \Pi, e
\end{align*}
\]

Syntactic sugar: term $t@L$

- attach label $L$ to any variable of $t$ that does not have an explicit label yet.
- example: $(x + y@K + 2)@L + x$ is $x@L + y@K + 2 + x@\text{Here}$. 
New rules for WP

New rules for computing WP:

\[
\begin{align*}
\text{WP}(x := t, Q) & = Q[x@Here \leftarrow t] \\
\text{WP}(L : e, Q) & = \text{WP}(e, Q)[x@L \leftarrow x@Here \mid x \text{ any variable}] 
\end{align*}
\]

Exercise:

\[\text{WP}(L : x := x + 42, x@Here > x@L) = ?\]
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Termination

Goal
Prove that a program terminates (on all inputs satisfying the precondition

With our simple language

- amounts to show that loops are never infinite

Solution: annotate loops with \textit{loop variants}

- a term that \textit{decreases at each iteration}
- for some \textit{well-founded ordering} \(\prec\) (i.e. there is no infinite sequence \(\text{val}_1 \succ \text{val}_2 \succ \text{val}_3 \succ \cdots\))
- A typical ordering on integers:

\[
x \prec y = x < y \land 0 \leq y
\]
Syntax

New syntax construct:

\[ e ::= \text{while } e \text{ invariant } \| \text{variant } t, \prec \text{ do } e \]

Example:

\[
\{ y \geq 0 \}
\]

L:

\text{while } y > 0 \text{ do}

\text{invariant } \{ x + y = x@L + y@L \}

\text{variant } \{ y \}

x := x + 1; y := y - 1

\{ x = x@Old + y@Old \land y = 0 \}

Demo

See Why3 version in sum.mlw
Operational semantics

\[
\llbracket I \rrbracket_{\Sigma, \Pi} \text{ holds} \\
\Sigma, \Pi, \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e \mapsto \\
\Sigma, \Pi, \text{if } c \\
\quad \text{then } (e; \text{assert } t \prec \llbracket t \rrbracket_{\Sigma, \Pi}; \\
\quad \quad \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e) \\
\quad \text{else } ()
\]

Alternative:

\[
\llbracket I \rrbracket_{\Sigma, \Pi} \text{ holds} \\
\Sigma, \Pi, \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e \mapsto \\
\Sigma, \Pi, L : \text{if } c \\
\quad \text{then } (e; \text{assert } t \prec t@L; \\
\quad \quad \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e) \\
\quad \text{else } ()
\]
Weakest Precondition

No distinction liberal/strict:

- presence of loop variants tells if one wants to prove termination or not

\[ WP(\text{while } c \text{ invariant } I \text{ variant } t, \prec \text{ do } e, Q) = I \land \forall \vec{v}, (I \Rightarrow WP(L : c, \text{if result then } WP(e, I \land t \prec t@L) \text{ else } Q)) \]

[\[ w_i \leftarrow v_i \]]
Outline

A ML-like Programming Language

Blocking Operational Semantics

Weakest Preconditions Revisited

Labels

Termination, Variants

Exercises
Example ISQRT, revisited

```plaintext
let old_x = x in
x := 0; sum := 1;
while sum ≤ old_x do
    x := x + 1;
    sum := sum + 2 * x + 1
done;
x
```

- Propose pre- and post-condition
- Propose suitable loop invariant and variant
Exponentiation

\[
\begin{align*}
r &:= 1.0; \\
p &:= x; \\
\textbf{while } n > 0 \textbf{ do} \\
&\quad \textbf{if } \text{mod } n 2 = 1 \textbf{ then } r := r \times p; \\
&\quad p := p \times p; \\
&\quad n := \text{div } n 2 \\
\textbf{done}; \\
r
\end{align*}
\]

▶ Propose pre- and post-condition
▶ Propose suitable loop invariant and variant
▶ add lemmas and assertions as hints for the proof