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.

```
r := 1; p := x; e := n;
while e > 0 do
  if mod2(e) ≠ 0 then r := r * p;
  p := p * p;
  e := div2(e);
```

- 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 $fib(0) = 0$, $fib(1) = 1$ and $fib(n+2) = fib(n+1) + fib(n)$. The following program is supposed to compute *fib* in linear time, the result being stored in y.

```
y := 0; x := 1; i := 0;
while i < n do
   aux := y; y := x; x := x + aux; i := i + 1
```

- Assuming *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 $\{Pre\}s\{Post\}$
- Weakest Liberal Precondition (WLP):
  - if $Pre \Rightarrow \mathrm{WLP}(s, Post)$ then $\{Pre\}s\{Post\}$ 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

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

last lecture	now
expression	term
formula	formula
statement	expression

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` t_1 `else` t_2

t	$::=$	*val*	(values, i.e. constants)
	$\mid$	v	(logic variables)
	$\mid$	x	(program variables)
	$\mid$	t *op* t	(binary operations)
	$\mid$	`if` t `then` t `else` t	(if-expression)

Local logic variables

We extend the syntax of terms by

$$t ::= \texttt{let}\ v = t\ \texttt{in}\ t$$

Example: approximated cosine

```
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:

$$
\begin{array}{lcll}
p & ::= & t & \text{(boolean term)} \\
 & | & p \wedge p \mid p \vee p \mid \neg p \mid p \Rightarrow p & \text{(connectives)} \\
 & | & \forall v : \tau,\ p \mid \exists v : \tau,\ p & \text{(quantification)} \\
 & | & \texttt{let}\ v = t\ \texttt{in}\ p & \text{(local binding)}
\end{array}
$$

Typing

- Types:

$$\tau \quad ::= \quad \texttt{int} \mid \texttt{real} \mid \texttt{bool} \mid \texttt{unit}$$

- Typing judgment:

$$\Gamma \vdash t : \tau$$

  where Γ maps identifiers to types:
  - either $v : \tau$ (logic variable, immutable)
  - either $x : \texttt{ref}\ \tau$ (program variable, mutable)

Important

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

Typing rules

Constants:

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash r : \texttt{real}}$$

$$\frac{}{\Gamma \vdash \mathit{True} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathit{False} : \texttt{bool}}$$

Variables:

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{x : \texttt{ref}\ \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Let binding:

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \{v : \tau_1\} \cdot \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ v = t_1\ \texttt{in}\ t_2 : \tau_2}$$

- All terms have a base type (not a reference)
- In practice: Why3, as in OCaml, requires to write !x for references

Formal Semantics: Terms and Formulas

Program states are augmented with a stack of local (immutable) variables

- Σ: maps program variables to values (a map)
- Π: maps logic variables to values (a stack)

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

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	(pure term)
	$\mid$	$e\ op\ e$	(binary operation)
	$\mid$	$x := e$	(assignment)
	$\mid$	$\texttt{let}\ v = e\ \texttt{in}\ e$	(local binding)
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	(conditional)
	$\mid$	$\texttt{while}\ e\ \texttt{do}\ e$	(loop)

- sequence $e_1 ; e_2$: syntactic sugar for

$$\texttt{let}\ v = e_1\ \texttt{in}\ e_2$$

when e_1 has type `unit` and v not used in e_2

Toy Examples

```
z := if x ≥ y then x else y
```

```
let v = r in (r := v + 42; v)
```

```
while (x := x - 1; x > 0) do ()
```

```
while (let v = x in x := x - 1; v > 0) do ()
```

Typing Rules for Expressions

Assignment:

$$\frac{x : \texttt{ref}\ \tau \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \texttt{unit}}$$

Let binding:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \{v : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ v = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Conditional:

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Loop:

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash e : \texttt{unit}}{\Gamma \vdash \texttt{while}\ c\ \texttt{do}\ e : \texttt{unit}}$$

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

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, x := e \rightsquigarrow \Sigma', \Pi', e'}$$

$$\frac{}{\Sigma, \Pi, x := val \rightsquigarrow \Sigma[x \leftarrow val], \Pi, ()}$$

- Let binding

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let } v = e_1 \texttt{ in } e_2 \rightsquigarrow \Sigma', \Pi', \texttt{let } v = e_1' \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let } v = val \texttt{ in } e \rightsquigarrow \Sigma, \{v = val\} \cdot \Pi, e}$$

Operational Semantics, Continued

- Binary operations

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma', \Pi', e_1' + e_2}$$

$$\frac{\Sigma, \Pi, e_2 \rightsquigarrow \Sigma', \Pi', e_2'}{\Sigma, \Pi, val_1 + e_2 \rightsquigarrow \Sigma', \Pi', val_1 + e_2'}$$

$$\frac{val = val_1 + val_2}{\Sigma, \Pi, val_1 + val_2 \rightsquigarrow \Sigma, \Pi, val}$$

Operational Semantics, Continued

- Conditional

$$\frac{\Sigma, \Pi, c \rightsquigarrow \Sigma', \Pi', c'}{\Sigma, \Pi, \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 \rightsquigarrow \Sigma', \Pi', \texttt{if } c' \texttt{ then } e_1 \texttt{ else } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{True} \texttt{ then } e_1 \texttt{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_1}$$

$$\frac{}{\Sigma, \Pi, \texttt{if } \mathit{False} \texttt{ then } e_1 \texttt{ else } e_2 \rightsquigarrow \Sigma, \Pi, e_2}$$

- Loop

$$\frac{}{\Sigma, \Pi, \texttt{while } c \texttt{ do } e \rightsquigarrow \Sigma, \Pi, \texttt{if } c \texttt{ then } (e; \texttt{while } c \texttt{ do } e) \texttt{ else } ()}$$

Context Rules versus Let Binding

Remark: most of the context rules can be avoided

- An equivalent operational semantics can be defined using $\texttt{let}\ v = \ldots\ \texttt{in}\ \ldots$ instead, e.g.:

$$\frac{v_1, v_2 \text{ fresh}}{\Sigma, \Pi, e_1 + e_2 \rightsquigarrow \Sigma, \Pi, \texttt{let}\ v_1 = e_1\ \texttt{in let}\ v_2 = e_2\ \texttt{in}\ v_1 + v_2}$$

- 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

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

$$
\begin{array}{lcll}
e & ::= & \texttt{assert}\ p & \text{(assertion)} \\
 & | & \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{do}\ e & \text{(annotated loop)}
\end{array}
$$

Toy Examples

```
z := if x ≥ y then x else y ;
assert z ≥ x ∧ z ≥ y
```

```
while (x := x - 1; x > 0)
  invariant x ≥ 0 do ();
assert (x = 0)
```

```
while (let v = x in x := x - 1; v > 0)
  invariant x ≥ -1 do ();
assert (x < 0)
```

Blocking Semantics: Modified Rules

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

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

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 *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 Σ, Π satisfying P, e *executes safely* in Σ, Π, and if it terminates, the final state satisfies Q

New addition in the specification language:

- keyword `result` in post-conditions
- denotes the value of the expression executed

Toy Examples, Continued

```
{ true }
if x ≥ y then x else y
{ result ≥ x ∧ result ≥ y }
```

```
{ x ≥ 0 }
c := 0; sum := 1;
while sum ≤ x do
  c := c + 1; sum := sum + 2 * c + 1
done;
c
{ result ≥ 0 ∧
  result * result ≤ x < (result+1)*(result+1) }
```

Outline

Weakest Preconditions Revisited

Goal:

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

Remark:

- Stating this for $Q = \mathit{true}$ 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:

$$\begin{aligned} &\mathrm{WP}(\texttt{let } x = e_1 \texttt{ in } e_2, Q) = \\ &\quad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow result]) \end{aligned}$$

Weakest Preconditions, continued

- Assignment:

$$\mathrm{WP}(x := e, Q) = \mathrm{WP}(e, Q[\mathit{result} \leftarrow (); x \leftarrow \mathit{result}])$$

- Alternative:

$$\begin{array}{rcl} \mathrm{WP}(x := e, Q) & = & \mathrm{WP}(\texttt{let } v = e \texttt{ in } x := v, Q) \\ \mathrm{WP}(x := t, Q) & = & Q[\mathit{result} \leftarrow (); x \leftarrow t]) \end{array}$$

WP: Exercise

$$\mathrm{WP}(\texttt{let } v = x \texttt{ in } (x := x + 1; v), x > \mathit{result}) = ?$$

Weakest Preconditions, continued

- Conditional

$$\mathrm{WP}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, Q) = \\ \mathrm{WP}(e_1, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e_2, Q) \texttt{ else } \mathrm{WP}(e_3, Q))$$

- Alternative with let: (exercise!)

Weakest Preconditions, continued

- Assertion

$$\begin{aligned} \mathrm{WP}(\texttt{assert}\ P, Q) &= P \wedge Q \\ &= P \wedge (P \Rightarrow Q) \end{aligned}$$

  (second version useful in practice)

- While loop

$$\begin{aligned} &\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = \\ &\quad I \wedge \\ &\quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathit{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))[w_i \leftarrow v_i] \end{aligned}$$

  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 $\models P \Rightarrow Q$ *then* $\models \mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)$

Proof: structural induction on e
Remark: true only when quantified on *all states*

Lemma (Conjunction Property)

If $\Sigma, \Pi \models \mathrm{WP}(e, P)$ *and* $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *then*
$\Sigma, \Pi \models \mathrm{WP}(e, P \wedge Q)$

Proof: structural induction on e

Soundness of WP

Lemma (Preservation by Reduction)

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

Proof: predicate induction of $\rightsquigarrow$.

Lemma (Progress)

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

Proof: structural induction of e.

Corollary (Soundness)

If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *then* e *executes safely in* Σ, Π.

Outline

Labels: Syntax and Typing

Add in syntax of *terms*:

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

Add in syntax of *expressions*:

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

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:

$$\begin{aligned} [\![x]\!]_{\Sigma,\Pi} &= \Sigma(x, \mathit{Here}) \\ [\![x@L]\!]_{\Sigma,\Pi} &= \Sigma(x, L) \end{aligned}$$

The operational semantics of expressions is modified as follows

$$\begin{aligned} \Sigma, \Pi, x := \mathit{val} &\rightsquigarrow \Sigma\{(x, \mathit{Here}) \leftarrow \mathit{val}\}, \Pi, () \\ \Sigma, \Pi, L : e &\rightsquigarrow \Sigma\{(x, L) \leftarrow \Sigma(x, \mathit{Here}) \mid x \text{ any variable}\}, \Pi, e \end{aligned}$$

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 $x@L + y@K + 2 + x@\mathit{Here}$.

New rules for WP

New rules for computing WP:

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

Exercise:

$$\mathrm{WP}(L : x := x + 42, x@\mathit{Here} > x@L) = ?$$

Outline

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 *loop variants*

- a term that *decreases at each iteration*
- for some *well-founded ordering* $\prec$ (i.e. there is no infinite sequence $val_1 \succ val_2 \succ val_3 \succ \cdots$
- A typical ordering on integers:

$$x \prec y \quad = \quad x < y \wedge 0 \leq y$$

Syntax

New syntax construct:

$$e \ ::= \ \texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e$$

Example:

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

Demo

See Why3 version in `sum.mlw`

Operational semantics

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec [\![t]\!]_{\Sigma,\Pi}; \\ \qquad\quad \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

Alternative:

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, L : \texttt{if}\ c \\ \qquad \texttt{then}\ (e; \texttt{assert}\ t \prec t@L; \\ \qquad\quad \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e) \\ \qquad \texttt{else}\ ()\end{array}}$$

Weakest Precondition

No distinction liberal/strict:

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

$$\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{variant}\ t, \prec\ \texttt{do}\ e, Q) =$$
$$I \wedge$$
$$\forall \vec{v}, (I \Rightarrow \mathrm{WP}(L : c, \texttt{if}\ \mathit{result}\ \texttt{then}\ \mathrm{WP}(e, I \wedge t \prec t@L)\ \texttt{else}\ Q))$$
$$[w_i \leftarrow v_i]$$

Outline

Example ISQRT, revisited

```
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

```
r := 1.0;
p := x;
while n > 0 do
  if mod n 2 = 1 then r := r *. p;
  p := p *. p;
  n := div n 2
done;
r
```

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