Types, Local Variables, Procedures

Claude Marché

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

9 janvier 2012
Simple IMP programs
- only the integer datatype
- only global variables
- program = sequence of statements

Hoare logic:
- Deduction rules for triples \(\{Pre\} s \{Post\}\)

Weakest Precondition:
- \(\{Pre\} s \{Post\}\) valid if \(Pre \Rightarrow WP(s, Post)\)

Extensions: labels, exceptions
Next Extensions

- More base types
- Scopes, local variables
- Procedures (call by value)
Outline

Types and Local Variables

Procedures
We can assume that the language provides more base types:

- reals, standard operations +, -, *
- booleans, standard operations and, or, not, if-expression
  "if then else"
- From now on, the condition of the if then else and the while do in programs is a Boolean expression
Local logic variables

We extend the syntax of expressions by

\[ e ::= \text{let } x = e_1 \text{ in } e_2 \]

Example: approximated cosine

\[
\text{val } x : \text{ref real}
\]
\[
\text{val } \text{res : ref real}
\]
\[
\text{res} := \text{let } y = x \times x \text{ in } 1.0 - 0.5 \times y + 0.04166666 \times y \times y
\]
Types, Typing Judgement

- **Types:**

  \[ \tau ::= b \mid \text{ref } b \]

  \[ b ::= \text{int} \mid \text{real} \mid \text{bool} \]

  **ref** denotes types of mutable variables, or *references*

- **Note:** no reference to reference

- **Typing judgement:**

  \[ \Gamma \vdash e : \tau \]

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

\[ x : b \in \Gamma \quad x : \text{ref} b \in \Gamma \]
\[ \Gamma \vdash x : b \quad \Gamma \vdash x : b \]

Let binding:

\[ \begin{align*}
\Gamma \vdash e_1 : \tau_1 \\
\{ x : \tau_1 \} \cdot \Gamma \vdash e_2 : \tau_2
\end{align*} \]
\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \]

- All expressions have a base type, not reference
- Practice: Why3, as in OCaml, would require to write \(!x\) for references
Semantics

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

- $\Sigma$: maps (reference variable, label) to values
- $\Gamma$: maps immutable variables to values

\[
\begin{align*}
    [n]_{\Sigma,\Gamma} &= n \\
    [x]_{\Sigma,\Gamma} &= \Sigma(x) \\
    &\text{if } x : \text{ref } \tau \\
    [x]_{\Sigma,\Gamma} &= \Gamma(x) \\
    &\text{if } x \text{ not a reference} \\
    [e_1 \ op \ e_2]_{\Sigma,\Gamma} &= [e_1]_{\Sigma,\Gamma} \ [op] \ [e_2]_{\Sigma,\Gamma} \\
    [\text{let } x = e_1 \ \text{in } e_2]_{\Sigma,\Gamma} &= [e_2]_{\Sigma,\Gamma'} \\
    \text{where } \Gamma' &= \{ x = [e_1]_{\Sigma,\Gamma} \} \cdot \Gamma
\end{align*}
\]
Since only the language of expressions is changed, rules for Hoare logic and WP remain the same

Practical Notes:
- theorem provers should support these extensions
- Why3 manages to transform formulas when needed (e.g. transformation of if-expressions and/or let-expressions into equivalent formulas)
Local program variables

We extend the syntax of statements with

\[ s ::= \text{let } id = e \text{ in } s \]
\[ \text{let } id = \text{ref } e \text{ in } s \]

Example: isqrt revisited

```.ml
val x, res : ref int

isqrt:
res := 0;
let sum = ref 1 in
while sum \leq x do
  res := res + 1; sum := sum + 2 * res + 1
done
```
Typing rules for statements

- Judgement: $\Gamma \vdash s : wf$ ("well-formed")

\[
\frac{x : \text{ref } b \in \Gamma \quad \Gamma \vdash e : b}{\Gamma \vdash x := e : wf}
\]

\[
\frac{\Gamma \vdash e : b \quad \{x : b\} \cdot \Gamma \vdash s : wf}{\Gamma \vdash \text{let } x = e \text{ in } s : wf}
\]

\[
\frac{\Gamma \vdash e : b \quad \{x : \text{ref } b\} \cdot \Gamma \vdash s : wf}{\Gamma \vdash \text{let } x = \text{ref } e \text{ in } s : wf}
\]

- Remaining rules are standard
Operational semantics

- One-step execution is a relation of the form
  \[ \Sigma, \Gamma, s \leadsto \Sigma', \Gamma', s' \]

- Previous rules are unchanged (with additional \( \Gamma \), unmodified)

  \[ \Sigma, \Gamma, \text{let } x = e \text{ in } s \leadsto \Sigma, \{ x = [e]_{\Sigma, \Gamma} \} \cdot \Gamma, s \]

  \[ \Sigma, \Gamma, \text{let } x = \text{ref } e \text{ in } s \leadsto \{(x, \text{Here}) = [e]_{\Sigma, \Gamma}\} \cdot \Sigma, \Gamma, s \]
Exercise: propose rules for

\[
\text{let } x = e \text{ in } s
\]

and

\[
\text{let } x = \text{ref } e \text{ in } s
\]
Local program variables: Hoare rules

\[ \{P\} s\{Q\} \]
\[ \{P[x \leftarrow e]\} \text{let } x = e \text{ in } s\{Q\} \]

\[ \{P\} s\{Q\} \]
\[ \{P[x \leftarrow e]\} \text{let } x = \text{ref } e \text{ in } s\{Q\} \]
Local program variables: Hoare rules

Alternative:

\[
\begin{align*}
\{P\} s\{Q\} \\
\{\text{let } x = e \text{ in } P\}\{\text{let } x = e \text{ in } s\{Q\}\}
\end{align*}
\]

\[
\begin{align*}
\{P\} s\{Q\} \\
\{\text{let } \xi = e \text{ in } P[x \leftarrow \xi]\}\{\text{let } x = \text{ref } e \text{ in } s\{Q\}\}
\end{align*}
\]
Local program variables: WP rules

Exercise: propose rules for

\[
\text{let } x = e \text{ in } s
\]

and

\[
\text{let } x = \text{ref } e \text{ in } s
\]
Local program variables: WP rules

\[
WP(\text{let } x = e \text{ in } s, Q) = WP(s, Q)[x \leftarrow e]
\]

\[
WP(\text{let } x = \text{ref } e \text{ in } s, Q) = WP(s, Q)[x \leftarrow e]
\]
Local program variables: WP rules

Alternative:

\[ WP(\text{let } x = e \text{ in } s, Q) = \text{let } x = e \text{ in } WP(s, Q) \]

\[ WP(\text{let } x = \text{ref } e \text{ in } s, Q) = \text{let } \xi = e \text{ in } WP(s, Q)[x \leftarrow \xi] \]
Outline

Types and Local Variables

Procedures
Procedures

Declaration of the form

procedure \( p(x_1 : b_1, \ldots, x_n : b_n) : \)
contract
body

- **Contract:**
  - pre-condition
  - post-condition (label \textit{Old} available)
  - assigned variables: clause \texttt{writes}

- **Body:** sequence of statements

Parameters are not references
(call by reference addressed in another lecture)
Example: \textit{isqrt}

\begin{verbatim}
val res: ref int

procedure isqrt(x:int):
contract:
  requires: x ≥ 0
  writes: res
  ensures: res ≥ 0 and sqr(res) ≤ x < sqr(res+1)
body:
  res := 0;
  let sum = ref 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done
\end{verbatim}
Example using Old label

```plaintext
val res: ref int

procedure incr(x:int):
contract:
  requires: true
  writes: res
  ensures: res = res@Old + x
body:
  res := res + x
```
Operational semantics

procedure $p(x_1: \tau_1, \ldots, x_n: \tau_n)$:
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
  body $Body$

\[
\begin{align*}
\Gamma' &= \{ x_i \leftarrow [e_i]_{\Sigma, \Gamma} \} \\
&\quad \text{where \textit{return} is a dummy statement introduced for this definition}
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Gamma, p(e_1, \ldots, e_n) &\rightsquigarrow \Sigma, \Gamma', (\text{Old} : Body; \text{return}(post, \Gamma)) \\
&\quad \text{where \textit{return} is a dummy statement introduced for this definition}
\end{align*}
\]
procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$:
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
  body $Body$

\[
\{Pre[x_i \leftarrow e_i]\} p(e_1, \ldots, e_n) \{Post[x_i \leftarrow e_i]\}
\]

- instance of precondition holds
- instance of postcondition holds
Example: isqrt(42)

Exercise: Prove that \( \{ \text{true} \} \text{isqrt}(42) \{ \text{res} = 6 \} \) holds using Hoare logic.
Hoare rule for procedure call (with Old)

procedure \( p(x_1 : \tau_1, \ldots, x_n : \tau_n) \):
  requires \( Pre \)
  writes \( \vec{w} \)
  ensures \( Post \)
body \( Body \)

\[
\begin{align*}
\{Pre[x_i \leftarrow e_i] \land w = \xi\} & \leftarrow f(e_1, \ldots, e_n) \leftarrow \{Post[x_i \leftarrow e_i][w@Old \leftarrow \xi]\}
\end{align*}
\]

- instance of precondition holds
- instance of postcondition holds, where references to \( w@Old \) are replaced by auxiliary variables
Example: incrementation

Exercise: Prove that $\{res = 6\} \text{incr}(36)\{res = 42\}$ holds using Hoare logic.
Soundness

Assuming that for each procedure declared as

procedure \( p(x_1 : \tau_1, \ldots, x_n : \tau_n) \):
  requires \( \text{Pre} \)
  writes \( \vec{w} \)
  ensures \( \text{Post} \)
body \( \text{Body} \)

\>
  variables assigned in \( \text{Body} \) belong to \( \vec{w} \)
\>
  the triple \( \{ \text{Pre} \} \text{Body} \{ \text{Post} \} \) is valid

then for any valid triple \( \{ P \} s \{ Q \} \), \( s \) executes safely in any state satisfying \( P \), and if it terminates the resulting state satisfies \( Q \).
WLP for procedure call

procedure \( p(x_1 : \tau_1, \ldots, x_n : \tau_n) \):
  requires \( Pre \)
  writes \( \vec{w} \)
  ensures \( Post \)
  body \( Body \)

\[
WLP(p(e_1, \ldots, e_n), Q) = Pre[x_i \leftarrow e_i] \land \\
\forall \vec{y}, (Post[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}][\vec{w}@Old \leftarrow \vec{w}@Here] \Rightarrow Q[\vec{w} \leftarrow \vec{y}])
\]

- instance of precondition holds
- instance of postcondition of \( p \) implies \( Q \) for any possible resulting values \( y \) of assigned variables \( \vec{w} \).
Exercise: Prove that \{true\} isqrt(42)\{res = 6\} holds using WLP calculus

See file isqrt_call.mlw
Example: incrementation

Exercise: Prove that $\{\text{res} = 6\} \text{incr}(36)\{\text{res} = 42\}$ holds using WLP calculus.

See file incr.mlw
Soundness

Assuming that for each procedure declared as

procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$:
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
  body $Body$

- variables assigned in $Body$ belong to $\vec{w}$
- The formula $\forall \vec{y}. Pre \Rightarrow WLP(Body, Post)$ is valid

then for any triple $\{P\}s\{Q\}$ such that $P \Rightarrow WLP(s, Q)$, $s$
executes safely in any state satisfying $P$, and if it terminates the
resulting state satisfies $Q$. 
Procedures raising exceptions

A generalized contract has the form

requires $Pre$
writes $\vec{w}$
raises $E_1 \cdots E_k$
ensures $Post \mid E_1 \rightarrow Post_1 \mid \cdots \mid E_k \rightarrow Post_k$

In the rule for WLP, the implication $Post[\ldots] \Rightarrow Q$ must be replaced by a conjunction of implications

$$(Post[\ldots] \Rightarrow Q) \land \bigwedge_i (Post_i[\ldots] \Rightarrow Q_i)$$
Exceptions example: exact square root

val res: ref int
exception NotSquare

procedure isqrt(x:int):
contract:
  requires: true
  writes: res
  ensures: res ≥ 0 and sqr(res) = x
          | NotSquare → forall n:int. sqr(n) ≠ x
body:
  if x < 0 then raise NotSquare;
  res := 0;
  let sum = ref 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done;
  if res * res ≠ x then raise NotSquare
Exceptions example: exact square root

See file isqrt_exc.mlw

See file isqrt_exc_call.mlw
Recursive Procedures: Termination

If a procedure is recursive, the termination of call can be proved, provided that the procedure is annotated with a variant, similar to loop variant.

procedure $p(x_1 : \tau_1, \ldots, x_n : \tau_n)$:
  requires $Pre$
  variant $var$ for $\prec$
  writes $\vec{w}$
  ensures $Post$
  body $Body$

\[
WP(p(e_1, \ldots, e_n), Q) = Pre[x_i \leftarrow e_i] \land var \prec var@Init
\forall \vec{y}, (Post[x_i \leftarrow e_i][\vec{w} \leftarrow \vec{y}][\vec{w}@Old \leftarrow \vec{w}@Here] \Rightarrow Q[\vec{w} \leftarrow \vec{y}])
\]

where $Init$ is a label assumed to be present at the beginning of the Body
Example: McCarthy’s 91 function

\[ f_{91}(n) = \begin{cases} f_{91}(f_{91}(n + 11)) & \text{if } n \leq 100 \\ n - 10 & \text{else} \end{cases} \]

Exercise: Find adequate specifications

```plaintext
val res : ref int

procedure f91 (n:int) :
  requires ?
  variant ?
  writes ?
  ensures ?

body:
  if n \leq 100 then f91 (n + 11); f91 !res
  else res := n - 10
```