Exceptions, Functions

Guillaume Melquiond

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

9 janvier 2012

Reminder of the Last 2 Lectures

- Simple IMP programs:

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).
- Weakest precondition computation:

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).
- Weakest precondition computation:
  - $\{Pre\}e\{Post\}$ valid if $Pre \Rightarrow \mathrm{WP}(e, Post)$,

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).

- Weakest precondition computation:
  - $\{Pre\}e\{Post\}$ valid if $Pre \Rightarrow \mathrm{WP}(e, Post)$,
  - notion of preservation by reduction.

Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).
- Weakest precondition computation:
  - $\{Pre\}e\{Post\}$ valid if $Pre \Rightarrow \mathrm{WP}(e, Post)$,
  - notion of preservation by reduction.
- Extension: labels.

Next Extensions

- Mutable local variables.
- Exceptions.
- Functions (call by value).

Outline

Mutable Local Variables

We extend the syntax of expressions with

$$e ::= \texttt{let ref } id = e \texttt{ in } e$$

Example: isqrt revisited

```
val x, res : ref int

isqrt:
  res := 0;
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done
```

Operational Semantics

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Π no longer contains just immutable variables.

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

$$\frac{}{\Sigma, \Pi, \texttt{let ref } x = v \texttt{ in } e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

Operational Semantics

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Π no longer contains just immutable variables.

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

$$\frac{}{\Sigma, \Pi, \texttt{let ref } x = v \texttt{ in } e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

$$\frac{x \text{ local variable}}{\Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

And labels too.

Mutable Local Variables: WP rules

Exercise: propose rules for $\mathrm{WP}(\texttt{let ref } x = e_1 \texttt{ in } e_2, Q)$, $\mathrm{WP}(x := e, Q)$, and $\mathrm{WP}(L : e, Q)$.

Mutable Local Variables: WP rules

$$\mathrm{WP}(\texttt{let ref}\ x = e_1\ \texttt{in}\ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathsf{result}])$$

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

$$\mathrm{WP}(L : e, Q) = \mathrm{WP}(e, Q)[x\texttt{@}L \leftarrow x, \text{for all } x\texttt{@}L]$$

Outline

Exceptions

We extend the syntax of expressions with

$$\begin{array}{lcl} e & ::= & \texttt{raise}\ exn \\ & | & \texttt{try}\ e\ \texttt{with}\ exn \Rightarrow e \end{array}$$

with *exn* a set of exception identifiers.

Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\texttt{let}\ \mathit{x} = \texttt{raise}\ \mathit{exn}\ \texttt{in}\ \mathit{e}) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ \mathit{exn}$$

Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\texttt{let}\ x = \texttt{raise}\ \mathit{exn}\ \texttt{in}\ e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ \mathit{exn}$$

Reduction of try-with:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, (\texttt{try}\ e\ \texttt{with}\ \mathit{exn} \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\texttt{try}\ e'\ \texttt{with}\ \mathit{exn} \Rightarrow e'')}$$

Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\texttt{let}\ x = \texttt{raise}\ exn\ \texttt{in}\ e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ exn$$

Reduction of try-with:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, (\texttt{try}\ e\ \texttt{with}\ exn \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\texttt{try}\ e'\ \texttt{with}\ exn \Rightarrow e'')}$$

Normal execution:

$$\Sigma, \Pi, (\texttt{try}\ v\ \texttt{with}\ exn \Rightarrow e') \rightsquigarrow \Sigma, \Pi, v$$

Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\texttt{let}\ x = \texttt{raise}\ exn\ \texttt{in}\ e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ exn$$

Reduction of try-with:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, (\texttt{try}\ e\ \texttt{with}\ exn \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\texttt{try}\ e'\ \texttt{with}\ exn \Rightarrow e'')}$$

Normal execution:

$$\Sigma, \Pi, (\texttt{try}\ v\ \texttt{with}\ exn \Rightarrow e') \rightsquigarrow \Sigma, \Pi, v$$

Exception handling:

$$\Sigma, \Pi, (\texttt{try}\ \texttt{raise}\ exn\ \texttt{with}\ exn \Rightarrow e) \rightsquigarrow \Sigma, \Pi, e$$

$$\frac{exn \neq exn'}{\Sigma, \Pi, (\texttt{try}\ \texttt{raise}\ exn\ \texttt{with}\ exn' \Rightarrow e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ exn}$$

Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

$$\{P\}e\{Q \mid exn_i \Rightarrow R_i\}$$

Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

$$\{P\}e\{Q \mid exn_i \Rightarrow R_i\}$$

Validity: if e is executed in a state where P holds,
it does not block and

- if it terminates normally with value v in state Σ,
  then $Q[\text{result} \leftarrow v]$ holds in Σ;

Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

$$\{P\}e\{Q \mid exn_i \Rightarrow R_i\}$$

Validity: if e is executed in a state where P holds, it does not block and

- if it terminates normally with value v in state Σ, then $Q[\text{result} \leftarrow v]$ holds in Σ;
- if it terminates with exception exn in state Σ, then there exists i such that $exn = exn_i$ and R_i holds in Σ.

Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

$$\{P\}e\{Q \mid exn_i \Rightarrow R_i\}$$

Validity: if e is executed in a state where P holds,
it does not block and

- if it terminates normally with value v in state Σ,
  then $Q[\text{result} \leftarrow v]$ holds in Σ;
- if it terminates with exception exn in state Σ,
  then there exists i such that $exn = exn_i$ and R_i holds in Σ.

Note: if e terminates with an exception not in the set $\{exn_i\}$,
the triple is not valid.

WP Rules

Function WP modified to allow exceptional post-conditions too:

$$\mathrm{WP}(e, Q, exn_i \Rightarrow R_i)$$

Implictly, $R_k = False$ for any $exn_k \notin \{exn_i\}$.

WP Rules

Function WP modified to allow exceptional post-conditions too:

$$\mathrm{WP}(e, Q, exn_i \Rightarrow R_i)$$

Implictly, $R_k = \mathit{False}$ for any $exn_k \notin \{exn_i\}$.

Extension of WP for simple expressions:

$$\mathrm{WP}(x := t, Q, exn_i \Rightarrow R_i) = Q[\text{result} \leftarrow (), x \leftarrow t]$$

$$\mathrm{WP}(\texttt{assert } R, Q, exn_i \Rightarrow R_i) = R \wedge Q$$

WP Rules

Extension of WP for composite expressions:

$$\begin{array}{l}\mathrm{WP}(\texttt{let } x = e_1 \texttt{ in } e_2, Q, exn_i \Rightarrow R_i) = \\ \qquad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i)[\mathrm{result} \leftarrow x], exn_i \Rightarrow R_i)\end{array}$$

$$\begin{array}{l}\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q, exn_i \Rightarrow R_i) = \\ \qquad \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q, exn_i \Rightarrow R_i) \\ \qquad\qquad \texttt{else } \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i)\end{array}$$

$$\begin{array}{l}\mathrm{WP}\left(\begin{array}{c}\texttt{while } c \texttt{ invariant } I \\ \texttt{variant } v, \prec \texttt{ do } e\end{array}, Q, exn_i \Rightarrow R_i\right) = I \wedge \forall x_1, \ldots, x_k, \\ \quad (I \wedge \texttt{if } c \texttt{ then } \mathrm{WP}(L : e, I \wedge v \prec v@L, exn_i \Rightarrow R_i) \\ \qquad\qquad\qquad\qquad\qquad \texttt{else } Q)[w_i \leftarrow x_i]\end{array}$$

where $w_1, \ldots, w_k$ is the set of assigned variables in expressions and $x_1, \ldots, x_k$ are fresh logic variables.

WP Rules

Exercise: propose rules for $\mathrm{WP}(\texttt{raise}\ exn, Q, exn_i \Rightarrow R_i)$ and $\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ exn \Rightarrow e_2, Q, exn_i \Rightarrow R_i)$.

WP Rules

$$\mathrm{WP}(\texttt{raise}\ exn_k, Q, exn_i \Rightarrow R_i) = R_k$$

$$\mathrm{WP}((\texttt{try}\ e_1\ \texttt{with}\ exn \Rightarrow e_2), Q, exn_i \Rightarrow R_i) =$$

$$\mathrm{WP}\left(e_1, Q, \begin{cases} exn \Rightarrow \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i) \\ exn_i \backslash exn \Rightarrow R_i \end{cases}\right)$$

Outline

Functions

Program structure:

$$\begin{array}{rcl} \textit{prog} & ::= & \textit{decl}^* \\ \textit{decl} & ::= & \textit{vardecl} \mid \textit{fundecl} \\ \textit{vardecl} & ::= & \text{val } \textit{id} : \text{ref } \textit{basetype} \end{array}$$

Functions

Program structure:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{vardecl} \mid \mathit{fundecl} \\
\mathit{vardecl} & ::= & \mathrm{val}\ \mathit{id} : \mathrm{ref}\ \mathit{basetype} \\
\mathit{fundecl} & ::= & \mathrm{function}\ \mathit{id}(\ (\mathit{param},)^*\ ):\mathit{basetype} \\
 & & \qquad \mathit{contract}\ \mathrm{body}\ \mathit{e} \\
\mathit{param} & ::= & \mathit{id} : \mathit{basetype} \\
\mathit{contract} & ::= & \mathrm{requires}\ \mathit{t}\ \mathrm{writes}\ (\mathit{id},)^*\ \mathrm{ensures}\ \mathit{t}
\end{array}
$$

Functions

Program structure:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{vardecl} \mid \mathit{fundecl} \\
\mathit{vardecl} & ::= & \text{val } \mathit{id} : \text{ref } \mathit{basetype} \\
\mathit{fundecl} & ::= & \text{function } \mathit{id}(\ (\mathit{param},)^*\ ):\mathit{basetype} \\
 & & \qquad \mathit{contract} \text{ body } e \\
\mathit{param} & ::= & \mathit{id} : \mathit{basetype} \\
\mathit{contract} & ::= & \text{requires } t \text{ writes } (\mathit{id},)^* \text{ ensures } t
\end{array}
$$

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label *Old* available),
  - assigned variables: clause writes .
- Body: expression.

Example: isqrt

```
function isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
```

Example using *Old* label

```
val res: ref int

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

Typing

Definition d of function f:

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

Typing

Definition d of function f:

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

Well-formed definitions:

$$\frac{\begin{array}{lr} \Gamma' = \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma & \vec{w} \subseteq \Gamma \\ \Gamma' \vdash Pre, Post : formula & \Gamma' \vdash Body : \tau \\ \vec{w}_g \subseteq \vec{w} \text{ for each call } g & y \in \vec{w} \text{ for each assign } y \end{array}}{\Gamma \vdash d : wf}$$

where Γ contains the global declarations.

Typing

Definition d of function f:

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
requires *Pre*
writes $\vec{w}$
ensures *Post*
body *Body*

Well-typed function calls:

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

Note: t_i are immutable expressions.

Operational Semantics

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

$$\frac{\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\} \qquad \Sigma, \Pi' \models \mathit{Pre}}{\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (\mathit{Old} : \texttt{frame}\ \Pi', \mathit{Body}, \mathit{Post})}$$

Operational Semantics of Function Call

`frame` is a dummy operation that keeps track of the local variables of the callee:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi'', (\texttt{frame}\ \Pi, e, P) \rightsquigarrow \Sigma', \Pi'', (\texttt{frame}\ \Pi', e', P)}$$

It also checks that the post-condition holds at the end:

$$\frac{\Sigma, \Pi' \models P[\text{result} \leftarrow v]}{\Sigma, \Pi, (\texttt{frame}\ \Pi', v, P) \rightsquigarrow \Sigma, \Pi, v}$$

WP Rule of Function Call

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

$$\begin{array}{l} \mathrm{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \wedge \\ \quad \forall \vec{v},\ (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j]) \end{array}$$

Example: isqrt(42)

Exercise: prove that $\{true\}isqrt(42)\{\mathrm{result} = 6\}$ holds.

```
function isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
```

Example: Incrementation

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

```
val res: ref int

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

Soundness of WP

Assuming that for each function defined as

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

we have

- variables assigned in $Body$ belong to $\vec{w}$,
- $\models Pre \Rightarrow \mathrm{WP}(Body, Post)[w_i@Old \leftarrow w_i]$ holds,

then for any formulas P and Q and any expression e,
$\{P\}e\{Q\}$ is a valid triple if $\models P \Rightarrow \mathrm{WP}(e, Q)$.

Soundness Proof

To prove soundness of WP rules:

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

   By structural induction on e.

Soundness Proof

To prove soundness of WP rules:

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

   By structural induction on e.

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

   By predicate induction on $\rightsquigarrow$.

Soundness Proof

To prove soundness of WP rules:

1. If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$,
   then $\Sigma', \Pi' \models \mathrm{WP}(e', Q)$.
   By structural induction on e.

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

Monotony lemma:

Given an expression e and its assigned variables $\vec{w}$,
if $\Sigma, \Pi \models \forall \vec{v},\ (P \Rightarrow Q)[w_i \leftarrow v_i]$,
then $\Sigma, \Pi \models \mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)$.

Functions Raising Exceptions

A generalized contract has the form

function $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$
  requires Pre
  raises $E_1 \cdots E_k$
  writes $\vec{w}$
  ensures $Post \mid E_1 \to Post_1 \mid \cdots \mid E_k \to Post_k$

In the WP, the implication $Post[\dots] \Rightarrow Q$ must be replaced by a conjunction of implications:

$$(Post[\dots] \Rightarrow Q) \wedge \bigwedge_i (Post_i[\dots] \Rightarrow R_i)$$

Example: Exact Square Root

```
exception NotSquare

function isqrt(x:int): int
  requires true
  raises NotSquare
  ensures result ≥ 0 ∧ sqr(result) = x
        | NotSquare → forall n:int. sqr(n) ≠ x
body
  if x < 0 then raise NotSquare;
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  if res * res ≠ x then raise NotSquare;
  res
```

Recursive Functions: Termination

If a function is recursive, termination of call can be proved, provided that the function is annotated with a variant.

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

WP for function call:

$$\begin{aligned} \mathrm{WP}(f(t_1, \ldots, t_n), Q) = {} & Pre[x_i \leftarrow t_i] \wedge var[x_i \leftarrow t_i] \prec var@Init \wedge \\ & \forall \vec{y},\ (Post[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow y_j]) \end{aligned}$$

with *Init* a label assumed to be present at the start of *Body*.

Example: Division

Exercise: find adequate specifications.

```
function div(x:int,y:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
```

Example: McCarthy's 91 Function

$$f91(n) = \texttt{if } n \leq 100 \texttt{ then } f91(f91(n+11)) \texttt{ else } n - 10$$

Exercise: find adequate specifications.

```
function f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n ≤ 100 then f91(f91(n + 11)) else n - 10
```