Termination
Ghost code, Lemma functions
More data types (lists, trees)
Handling Exceptions
Computer Arithmetic

Claude Marché

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

4 janvier 2022
Outline

Reminders, Solutions to Exercises
   Function calls
   Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
   Ghost code
   Ghost Functions
   Lemma functions

Modeling Continued: Specifying More Data Types
   Sum Types
   Lists

Exceptions

Application: Computer Arithmetic
   Handling Machine Integers
   Floating-Point Computations
Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
Function Calls

\[
\text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{requires } Pre \\
\text{writes } \vec{w} \\
\text{ensures } Post \\
\text{body } Body
\]

\[
\text{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land \\
\forall \vec{v}, (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@Old \leftarrow w_j] \rightarrow Q[w_j \leftarrow v_j])
\]

Modular proof

When calling function \( f \), only the contract of \( f \) is visible, not its body.
Soundness Theorem for a Complete Program

Assuming that for each function defined as

\[
\text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{requires } Pre \\
\text{writes } \vec{w} \\
\text{ensures } Post \\
\text{body } Body
\]

we have

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

then for any formula \( Q \) and any expression \( e \),
if \( \Sigma, \pi \models WP(e, Q) \) then execution of \( \Sigma, \pi, e \) is safe

Remark: (mutually) recursive functions are allowed
Programs on Arrays

- applicative maps as a built-in theory
- array = record (length, pure map)
- handling of out-of-bounds index check

```plaintext
type array 'a = { length : int; elts : int -> 'a}

val get (ref a:array 'a) (i:int) : 'a
  requires 0 <= i < a.length
  ensures  result = select(a.elts,i)

val set (ref a:array 'a) (i:int) (v:'a) : unit
  requires 0 <= i < a.length
  writes a
  ensures a.length = a@Old.length /
    a.elts = store(a@Old.elts,i,v)
```

- `a[i]` interpreted as a call to `get(a,i)`
- `a[i] <- v` interpreted as a call to `set(a,i,v)`
Exercise: Search Algorithms

 var a: array int

 let search(v:int): int
     requires 0 <= a.length
     ensures { ? } = ?

 1. Formalize postcondition: if v occurs in a, between 0 and a.length – 1, then result is an index where v occurs, otherwise result is set to –1

 2. Implement and prove linear search:
     res ← –1;
     for each i from 0 to a.length – 1: if a[i] = v then res ← i;
     return res

 See file lin_search.mlw
low = 0; high = a.length – 1;
while low ≤ high:
    let m be the middle of low and high
    if a[m] = v then return m
    if a[m] < v then continue search between m and high
    if a[m] > v then continue search between low and m

See file bin_search.mlw
Home Work: “for” loops

Syntax:  
\[ \text{for } i = e_1 \text{ to } e_2 \text{ do } e \]

Typing:
- \( i \) visible only in \( e \), and is immutable
- \( e_1 \) and \( e_2 \) must be of type \text{int}, \( e \) must be of type \text{unit}

Operational semantics:
(assuming \( e_1 \) and \( e_2 \) are values \( v_1 \) and \( v_2 \))

\[
\begin{align*}
\text{if } v_1 > v_2 \\
\text{then } \{ \Sigma, \pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \mapsto \Sigma, \pi, () \}
\end{align*}
\]

\[
\begin{align*}
\text{if } v_1 \leq v_2 \\
\text{then } \{ \Sigma, \pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \mapsto \Sigma, \pi, (\text{let } i = v_1 \text{ in } e); (\text{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e) \}
\end{align*}
\]
Home Work: “for” loops

Propose a Hoare logic rule for the for loop:

\[
\{?\} e \{?\} \\
\{?\} \text{for } i = v_1 \text{ to } v_2 \text{ do } e \{?\}
\]

Propose a rule for computing the WP:

\[
\text{WP}(\text{for } i = v_1 \text{ to } v_2 \text{ invariant } I \text{ do } e, Q) = ?
\]
Home Work: “for” loops

Notice: loop invariant \( i \) typically has \( i \) as a free variable

Informal vision of execution, stating when invariant is supposed to hold and for which value of \( i \):

\[
\begin{align*}
\{ & i[v1] \} \\
& i \leftarrow v1 \\
\{ & / \} \\
& * \ \\
& \{ & i \leftarrow i + 1 \} \\
& i \leftarrow i + 1 \\
\{ & / \} \\
& * \ \\
& \vdots \\
\{ & / \} \\
& * \ \\
& \{ & i \leftarrow i + 1 \} \\
& i \leftarrow i + 1 \\
(* \ assuming \ now \ i = v2, \ last \ iteration *) \\
\{ & / \} (* \ where \ i = v2 *) \\
& * \ \\
& \{ & i \leftarrow i + 1 \} (* \ and \ still \ i=v2, \ hence *) \\
& \{ & i \leftarrow v2 + 1 \} \\
\end{align*}
\]
Home Work: “for” loops

So we deduce the Hoare logic rule

\[
\frac{\{ I \land v_1 \leq i \leq v_2 \}\ e\ \{ I[i \leftarrow i + 1] \} }{\{ I[i \leftarrow v_1] \land v_1 \leq v_2 \} \text{ for } i = v_1 \text{ to } v_2 \text{ do } e\ \{ I[i \leftarrow v_2 + 1] \}}
\]

Remark

Some rule should be stated for case $v_1 > v_2$, left as exercise

and then a rule for computing the WP:

\[
\text{WP}(\text{for } i = v_1 \text{ to } v_2 \text{ invariant } I \text{ do } e, Q) = \\
\quad v_1 \leq v_2 \land I[i \leftarrow v_1] \land \\
\quad \forall v', ( \\
\quad \quad (\forall i, I \land v_1 \leq i \leq v_2 \rightarrow \text{WP}(e, I[i \leftarrow i + 1]))) \land \\
\quad \quad (I[i \leftarrow v_2 + 1] \rightarrow Q)) [w_j \leftarrow v_j]
\]

Additional exercise: use a for loop in the linear search example

lin_search_for.mlw
Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
Termination

**Goal**

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

Amounts to show that

- loops never execute infinitely many times
- (mutual) recursive calls cannot occur infinitely many times
Case of loops

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 \( \text{val}_1 \succ \text{val}_2 \succ \text{val}_3 \succ \cdots \))
- A typical ordering on integers:

\[
x \prec y \quad \equiv \quad x < y \land 0 \leq y
\]
Syntax

New syntax construct:

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

Example:

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

L:

while \ y > 0 \ do
  invariant \{ \ x + y = x@L + y@L \}
  variant \{ \ y \}
  x <- x + 1; \ y <- y - 1
\}

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

\[
\left\lfloor l \right\rfloor_{\Sigma, \pi} \text{ holds}
\]

\[
\Sigma, \pi, \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e \leadsto \\
\Sigma, \pi, L : \text{if } c \\
\quad \text{then } (e; \text{assert } t \prec t@L; \\
\quad \quad \text{while } c \text{ invariant } l \text{ variant } t, \prec \text{ do } e) \\
\quad \text{else } ()
\]

(new parts shown in red)
Weakest Precondition

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

\[ [w_i \leftarrow v_i] \]

In practice with Why3

- presence of loop variants tells if one wants to prove termination or not
- warning issued if no variant given
- keyword \texttt{diverges} in contract for non-terminating functions
- default ordering determined from type of \( t \)
Examples

Exercise: find adequate variants

\begin{verbatim}
i <- 0;
while i <= 100
  variant ?
do i <- i+1
done;
\end{verbatim}

\begin{verbatim}
while sum <= x
  variant ?
do
  res <- res + 1; sum <- sum + 2 * res + 1
done;
\end{verbatim}
Examples

Exercise: find adequate variants

```plaintext
i <- 0;
while i <= 100
  variant ?
do i <- i+1
done;

while sum <= x
  variant ?
do
    res <- res + 1; sum <- sum + 2 * res + 1
done;
```

Solutions:

variant 100 - i
invariant res >= 0
variant x - sum
Recursive Functions: Termination

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

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

WP for function call:

$$WP(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land var[x_i \leftarrow t_i] \prec var@Old \land$$
$$\forall \vec{y}, (Post[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@Old \leftarrow w_j] \rightarrow Q[w_j \leftarrow y_j])$$
Example of variant on a recursive function

```
let fib (x:int) : int
  variant ?
  body
  if x <= 1 then 1 else fib (x-1) + fib (x-2)
```
Example of variant on a recursive function

```
let fib (x:int) : int
   variant ?
   body
     if x <= 1 then 1 else fib (x-1) + fib (x-2)
```

Solution:

```
variant x
```
Case of mutual recursion

Assume two functions $f(\vec{x})$ and $g(\vec{y})$ that call each other

- each should be given its own variant $v_f$ (resp. $v_g$) in their contract
- with the same well-founded ordering $\prec$

When $f$ calls $g(\vec{t})$ the WP should include

$$v_g[\vec{y} \leftarrow \vec{t}] \prec v_f@Old$$

and symmetrically when $g$ calls $f$
Home Work 1: McCarthy’s 91 Function

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

Find adequate specifications

\[
\begin{align*}
\text{let } & f_{91}(n:\text{int}): \text{int} \\
\text{requires } & ? \\
\text{variant } & ? \\
\text{writes } & ? \\
\text{ensures } & ? \\
\text{body} & \\
\text{if } & n \leq 100 \text{ then } f_{91}(f_{91}(n + 11)) \text{ else } n - 10
\end{align*}
\]

Use canvas file \text{mccarthy.mlw}
Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
(Why3) Logic Language (reminder)

- (First-order) logic, built-in arithmetic (integers and reals)
- **Definitions** à la ML
  - logic (i.e. pure) *functions, predicates*
  - structured types, pattern-matching (to be seen in this lecture)
- *type polymorphism* à la ML
- *higher-order logic as a built-in theory of functions*
- Axiomatizations
- Inductive predicates (not detailed here)

**Important note**

Logic functions and predicates are *always totally defined*
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```
q <- 0; r <- x;
while r >= y do
    invariant { x = q * y + r }
    r <- r - y; q <- q + 1
```
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

\[
\begin{align*}
& r \leftarrow x; \\
& \textbf{while } r \geq y \textbf{ do} \\
& \quad \textbf{invariant } \{ \text{exists } q. \ x = q \times y + r \} \\
& \quad r \leftarrow r - y;
\end{align*}
\]
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

```plaintext
q <- 0; r <- x;
while r >= y do
  invariant { x = q * y + r }
  r <- r - y; q <- q + 1
```
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

\[
q \leftarrow 0; \ r \leftarrow x;
\]
\[
\text{while } r \geq y \text{ do}
\]
\[
\quad \text{invariant } \{ x = q \ast y + r \}
\]
\[
\quad r \leftarrow r - y; \ q \leftarrow q + 1
\]

Ghost code, ghost variables

- Cannot interfere with regular code (checked by typing)
- Visible only in annotations

(See Why3 file euclidean_rem.mlw)
Extend the post-condition of Euclid’s algorithm for GCD to express the Bézout property:

$$\exists a, b, \text{result} = x \cdot a + y \cdot b$$

Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`
More Ghosts: Programs turned into Logic Functions

If the program $f$ is

- **Proved terminating**
- **Has no side effects**

then there exists a logic function:

\[
\begin{align*}
\text{let } & f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{requires } & \text{Pre} \\
\text{variant } & \text{var}, \prec \\
\text{ensures } & \text{Post} \\
\text{body } & \text{Body}
\end{align*}
\]

function $f \tau_1 \ldots \tau_n : \tau$

lemma $f_{\text{spec}} : \forall x_1, \ldots, x_n. \text{Pre} \rightarrow \text{Post}[\text{result} \leftarrow f(x_1, \ldots, x_n)]$

and if $\text{Body}$ is a pure term then

lemma $f_{\text{body}} : \forall x_1, \ldots, x_n. \text{Pre} \rightarrow f(x_1, \ldots, x_n) = \text{Body}$

Offers an important alternative to axiomatic definitions

In Why3: done using keywords `let function`
Example: axiom-free specification of factorial

let function fact (n:int) : int
  requires { n >= 0 }
  variant { n }
= if n=0 then 1 else n * fact(n-1)

generates the logic context

function fact int : int

axiom f_body: forall n. n >= 0 ->
    fact n = if n=0 then 1 else n * fact(n-1)
Example of Factorial

Exercise: Find appropriate precondition, postcondition, loop invariant, and variant, for this program:

```ml
let fact_imp (x:int): int
    requires ?
    ensures ?
body
    let ref y = 0 in
    let ref res = 1 in
    while y < x do
        y <- y + 1;
        res <- res * y
    done;
    res
```

See file `fact.mlw`
More Ghosts: Lemma functions

- If a program function is \textit{without side effects} and \textit{terminating}:

  let $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \text{unit}$
  
  requires $Pre$
  
  variant $\text{var}, \prec$
  
  ensures $Post$
  
  body $Body$

  then it is a proof of

  $$\forall x_1, \ldots, x_n. Pre \rightarrow Post$$

- If $f$ is recursive, it simulates a proof by induction
Example: sum of odds

```
function sum_of_odd_numbers int : int
  (** `sum_of_odd_numbers n` denote the sum of
      odd numbers from `1` to `2n-1` *)

axiom sum_of_odd_numbers_base : sum_of_odd_numbers 0 = 0

axiom sum_of_odd_numbers_rec : forall n. n >= 1 ->
  sum_of_odd_numbers n = sum_of_odd_numbers (n-1) + 2*n-1

goal sum_of_odd_numbers_any:
  forall n. n >= 0 -> sum_of_odd_numbers n = n * n
```

See file `arith_lemma_function.mlw`
Example: sum of odds as lemma function

\[
\begin{align*}
\text{let rec lemma } & \text{ sum_of_odd_numbers_any } (n:\text{int}) \\
& \text{ requires } \{ \ n \geq 0 \ \} \\
& \text{ variant } \{ \ n \ \}
\end{align*}
\]

\[
\begin{align*}
& \text{ ensures } \{ \ \text{sum_of_odd_numbers } n = n \times n \ \} \\
& = \text{ if } n > 0 \text{ then sum_of_odd_numbers_any } (n-1)
\end{align*}
\]
Home work 3

Prove the helper lemmas stated for the fast exponentiation algorithm

See `power_int_lemma_functions.mlw`
Prove Fermat’s little theorem for case $p = 3$:

$$\forall x, \exists y. x^3 - x = 3y$$

using a lemma function

See `little_fermat_3.mlw`
Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
Sum Types

▶ Sum types à la ML:

```ocaml
type t =
| C₁ τ₁,₁ ⋯ τ₁,n₁
| ...
| Cₖ τₖ,₁ ⋯ τₖ,nₖ
```

▶ Pattern-matching with `match`

```ocaml
match e with
| C₁ (p₁, ⋯, pₙ₁) → e₁
| ...
| Cₖ (p₁, ⋯, pₙₖ) → eₖ
end
```

▶ Extended pattern-matching, wildcard: _
Sum Types

- Sum types à la ML:

\[
\text{type } t = \\
| C_1 \tau_{1,1} \cdots \tau_{1,n_1} \\
| \vdots \\
| C_k \tau_{k,1} \cdots \tau_{k,n_k}
\]

- Pattern-matching with

\[
\text{match } e \text{ with} \\
| C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1 \\
| \vdots \\
| C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k \\
\text{end}
\]
Sum Types

- Sum types à la ML:
  
  type t =
  |  C₁ τ₁₁ ⋯ τ₁ₙ₁
  |  ...
  |  Cₖ τₖ₁ ⋯ τₖₙₖ

- Pattern-matching with

  match e with
  |  C₁(p₁, ⋯ , pₙ₁) → e₁
  |  ...
  |  Cₖ(p₁, ⋯ , pₙₖ) → eₖ
  end

- Extended pattern-matching, wildcard: _
Recursive Sum Types

- Sum types can be recursive.
- Recursive definitions of functions or predicates
  - Must terminate (only total functions in the logic)
  - In practice in Why3: recursive calls only allowed on structurally smaller arguments.
type list 'a = Nil | Cons 'a (list 'a)

function append(l1:list 'a,l2:list 'a): list 'a =
  match l1 with
  | Nil -> l2
  | Cons(x,l) -> Cons(x, append(l,l2))
  end

function length(l:list 'a): int =
  match l with
  | Nil -> 0
  | Cons(_,r) -> 1 + length r
  end

function rev(l:list 'a): list 'a =
  match l with
  | Nil -> Nil
  | Nil -> Nil
  | Cons(x,r) -> append(rev(r), Cons(x,Nil))
  end
“In-place” List Reversal

Exercise: fill the holes below.

```ml
val ref l: list int

let rev_append(r:list int)
  variant ? writes ? ensures ?
body
  match r with
  | Nil -> ()
  | Cons(x,r) -> l <- Cons(x,l); rev_append(r)
end

let reverse(r:list int)
  writes l ensures l = rev r
body ?
```

See rev.mlw
Binary Trees

```ocaml
type tree 'a = Leaf | Node (tree 'a) 'a (tree 'a)
```

Home work: specify, implement, and prove a procedure returning the maximum of a tree of integers.

Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
Exceptions

We extend the syntax of expressions with

\[ e ::= \text{raise } exn \]
\[ \text{try } e \text{ with } exn \rightarrow e \]

with \( exn \) a set of exception identifiers, declared as

\text{exception } exn <\text{type}>

Remark: \(<\text{type}>\) can be omitted if it is \text{unit}
Example: linear search revisited in \text{lin_search_exc.mlw}
Operational Semantics

- Values: either constants $v$ or $\text{raise } exn$

Propagation of thrown exceptions:

$$\Sigma, \pi, (\text{let } x = \text{raise } exn \text{ in } e) \leadsto \Sigma, \pi, \text{raise } exn$$

Exception handling:

$$\Sigma, \pi, (\text{try raise } exn \text{ with } exn \rightarrow e) \leadsto \Sigma, \pi, \text{raise } exn$$
Operational Semantics

- Values: either constants $v$ or $\text{raise } exn$

Propagation of thrown exceptions:

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

Reduction of try-with:

$$\Sigma, \pi, e \rightsquigarrow \Sigma', \pi', e'$$

$$\Sigma, \pi, (\text{try } e \text{ with } exn \rightarrow e'') \rightsquigarrow \Sigma', \pi', (\text{try } e' \text{ with } exn \rightarrow e'')$$
Operational Semantics

- Values: either constants \( v \) or \( \text{raise } \text{exn} \)

Propagation of thrown exceptions:

\[
\Sigma, \pi, (\text{let } x = \text{raise } \text{exn} \text{ in } e) \rightsquigarrow \Sigma, \pi, \text{raise } \text{exn}
\]

Reduction of try-with:

\[
\Sigma, \pi, e \rightsquigarrow \Sigma', \pi', e' \\
\Sigma, \pi, (\text{try } e \text{ with } \text{exn } \rightarrow e'') \rightsquigarrow \Sigma', \pi', (\text{try } e' \text{ with } \text{exn } \rightarrow e'')
\]

Normal execution:

\[
\Sigma, \pi, (\text{try } v \text{ with } \text{exn } \rightarrow e') \rightsquigarrow \Sigma, \pi, v
\]
Operational Semantics

- Values: either constants \( v \) or \( \text{raise } exn \)

Propagation of thrown exceptions:

\[
\Sigma, \pi, (\text{let } x = \text{raise } exn \text{ in } e) \leadsto \Sigma, \pi, \text{raise } exn
\]

Reduction of try-with:

\[
\Sigma, \pi, e \leadsto \Sigma', \pi', e' \\
\Sigma, \pi, (\text{try } e \text{ with } exn \rightarrow e'') \leadsto \Sigma', \pi', (\text{try } e' \text{ with } exn \rightarrow e'')
\]

Normal execution:

\[
\Sigma, \pi, (\text{try } v \text{ with } exn \rightarrow e') \leadsto \Sigma, \pi, v
\]

Exception handling:

\[
\Sigma, \pi, (\text{try raise } exn \text{ with } exn' \rightarrow e) \leadsto \Sigma, \pi, e \\
\text{exn} \neq \text{exn}' \\
\Sigma, \pi, (\text{try raise } exn \text{ with } exn' \rightarrow e) \leadsto \Sigma, \pi, \text{raise } exn
\]
Function WP modified to allow exceptional post-conditions too:

\[
WP(e, Q, exn_i \rightarrow R_i)
\]

Implicitly, \( R_k = False \) for any \( exn_k \not\in \{exn_i\} \).
Function \textit{WP} modified to allow \textit{exceptional post-conditions} too:

\[ \text{WP}(e, Q, \text{exn}_i \rightarrow R_i) \]

Implicitly, \( R_k = False \) for any \( \text{exn}_k \not\in \{ \text{exn}_i \} \).

Extension of \textit{WP} for simple expressions:

\[ \text{WP}(x \leftarrow t, Q, \text{exn}_i \rightarrow R_i) = Q[\text{result} \leftarrow (), x \leftarrow t] \]

\[ \text{WP}(\text{assert } R, Q, \text{exn}_i \rightarrow R_i) = R \land Q \]
WP Rules

Extension of WP for composite expressions:

\[
\text{WP}\left(\text{let } x = e_1 \text{ in } e_2, Q, \text{exn}_i \rightarrow R_i\right) = \\
\text{WP}(e_1, \text{WP}(e_2, Q, \text{exn}_i \rightarrow R_i)[\text{result} \leftarrow x], \text{exn}_i \rightarrow R_i)
\]

\[
\text{WP}\left(\text{if } t \text{ then } e_1 \text{ else } e_2, Q, \text{exn}_i \rightarrow R_i\right) = \\
\text{if } t \text{ then } \text{WP}(e_1, Q, \text{exn}_i \rightarrow R_i) \text{ else } \text{WP}(e_2, Q, \text{exn}_i \rightarrow R_i)
\]

\[
\text{WP}\left(\begin{array}{l}
\text{while } c \text{ invariant } I \\
\text{do } e
\end{array}\right) = I \land \forall \vec{v}, \\
(l \rightarrow \text{if } c \text{ then } \text{WP}(e, l, \text{exn}_i \rightarrow R_i) \text{ else } Q)[w_i \leftarrow v_i]
\]

where \( w_1, \ldots, w_k \) is the set of assigned variables in \( e \) and \( v_1, \ldots, v_k \) are fresh logic variables.
WP Rules

Exercise: propose rules for

$$WP(\text{raise } \text{exn}, Q, \text{exn}_i \rightarrow R_i)$$

and

$$WP(\text{try } e_1 \text{ with } \text{exn} \rightarrow e_2, Q, \text{exn}_i \rightarrow R_i)$$
WP Rules

\[
\begin{align*}
\text{WP}(\text{raise } exn_k, Q, exn_i \rightarrow R_i) &= R_k \\
\text{WP}((\text{try } e_1 \text{ with } exn \rightarrow e_2), Q, exn_i \rightarrow R_i) &= \\
&\quad \text{WP} \left( e_1, Q, \left\{ \begin{array}{l}
exn \rightarrow \text{WP}(e_2, Q, exn_i \rightarrow R_i) \\
\text{exn}_i \setminus \text{exn} \rightarrow R_i
\end{array} \right. \right)
\end{align*}
\]
Functions Throwing Exceptions

Generalized contract:

\[
\text{val } f(x_1: \tau_1, \ldots, x_n: \tau_n): \tau \\
\quad \text{requires } Pre \\
\quad \text{writes } \vec{w} \\
\quad \text{ensures } Post \\
\quad \text{raises } E_1 \rightarrow Post_1 \\
\vdots \\
\quad \text{raises } E_n \rightarrow Post_n
\]

Extended WP rule for function call:

\[
WP(f(t_1, \ldots, t_n), Q, E_k \rightarrow R_k) = Pre[x_i \leftarrow t_i] \land \forall \vec{v}, \\
(Post[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow Q[w_j \leftarrow v_j]) \land \\
\land_k (Post_k[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow R_k[w_j \leftarrow v_j])
\]
Verification Conditions for programs

For each function defined with generalized contract

\[
\text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\quad \text{requires } Pre \\
\quad \text{writes } \vec{w} \\
\quad \text{ensures } Post \\
\quad \text{raises } E_1 \rightarrow Post_1 \\
\vdots \\
\quad \text{raises } E_n \rightarrow Post_n \\
\text{body } Body
\]

we have to check

- Variables assigned in \text{Body} belong to $\vec{w}$
- $Pre \rightarrow WP(\text{Body}, Post, E_k \rightarrow Post_k)[w_i@Old \leftarrow w_i]$ holds
Example: “Defensive” variant of ISQRT

```ocaml
exception NotSquare

let isqrt(x:int): int
  ensures result >= 0 /
  sqr(result) = x
  raises NotSquare -> for all 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 sqr(res) <> x then raise NotSquare;
  res
```

See Why3 version in `isqrt_exc.mlw`
▶ Implement and prove binary search using also a immediate exit:

\[
\text{low } = 0; \text{ high } = a.\text{length} - 1;
\]
while \( \text{low} \leq \text{high} \):
  let \( m \) be the middle of \( \text{low} \) and \( \text{high} \)
  if \( a[m] = v \) then return \( m \)
  if \( a[m] < v \) then continue search between \( m \) and \( \text{high} \)
  if \( a[m] > v \) then continue search between \( \text{low} \) and \( m \)

(see \text{bin\_search\_exc.mlw})
Outline

Reminders, Solutions to Exercises
  Function calls
  Programs on Arrays

Termination, Variants

Specification Language and Ghost Code
  Ghost code
  Ghost Functions
  Lemma functions

Modeling Continued: Specifying More Data Types
  Sum Types
  Lists

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - $2^{31} + 1 \rightarrow -2^{31}$
  - $100000^2 \rightarrow 1410065408$

See also arith.c
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$

- floating-point numbers (32-, 64-bit):
  - overflows
    - $2 \times 2 \times \cdots \times 2 \rightarrow +inf$
    - $-1/0 \rightarrow -inf$
    - $0/0 \rightarrow NaN$
Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - $2147483647 + 1 \rightarrow -2147483648$
  - $100000^2 \rightarrow 1410065408$

- floating-point numbers (32-, 64-bit):
  - overflows
    - $2 \times 2 \times \cdots \times 2 \rightarrow +\text{inf}$
    - $-1/0 \rightarrow -\text{inf}$
    - $0/0 \rightarrow \text{NaN}$
  - rounding errors
    - $0.1 + 0.1 + \cdots + 0.1 = 1.0 \rightarrow \text{false}$
      (because $0.1 \rightarrow 0.100000001490116119384765625$ in 32-bit)

See also arith.c
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.

▶ 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
▶ 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
▶ 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
Some Numerical Failures

(see more at http://catless.ncl.ac.uk/php/risks/search.php?query=rounding)

▶ 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.
▶ 1992, Green Party of Schleswig-Holstein seats in Parliament for a few hours, until a rounding error is discovered.
▶ 1995, Ariane 5 explodes during its maiden flight due to an overflow: insurance cost is $500M.
▶ 2007, Excel displays 77.1 × 850 as 100000.
Some Numerical Failures

- 1991, during Gulf War 1, a Patriot system fails to intercept a Scud missile: 28 casualties.

  Internal clock ticks every 0.1 second.
  Time is tracked by fixed-point arith.: $0.1 \approx 209715 \cdot 2^{-24}$.
  Cumulated skew after 24h: $-0.08s$, distance: 160m.
  System was supposed to be rebooted periodically.

- 2007, Excel displays $77.1 \times 850$ as 100000.

  Bug in binary/decimal conversion.
  Failing inputs: 12 FP numbers.
  Probability to uncover them by random testing: $10^{-18}$. 
Integer overflow: example of Binary Search

Google “Read All About It: Nearly All Binary Searches and Mergesorts are Broken”

```
let ref l = 0 in
let ref u = a.length - 1 in
while l <= u do
  let m = (l + u) / 2 in
  ...
```

$l + u$ may overflow with large arrays!

Goal
prove that a program is safe with respect to overflows
32-bit signed integers in two-complement representation: integers between $-2^{31}$ and $2^{31} - 1$.

If the mathematical result of an operation fits in that range, that is the computed result.

Otherwise, an overflow occurs. Behavior depends on language and environment: modulo arith, saturated arith, abrupt termination, etc.

A program is safe if no overflow occurs.
Idea: replace all arithmetic operations by abstract functions with preconditions. \( x + y \) becomes \( \text{int32_add}(x, y) \).

```java
val int32_add(x: int, y: int): int
    requires -2^31 <= x + y < 2^31
    ensures result = x + y
```

 Unsatisfactory: range contraints of integer must be added explicitly everywhere
Safety Checking, Second Attempt

Idea:
- replace type \textit{int} with an abstract type \textit{int32}
- introduce a \textit{projection} from \textit{int32} to \textit{int}
- axiom about the \textit{range} of projections of \textit{int32} elements
- replace all operations by abstract functions with preconditions

\textbf{type} \textit{int32}

\textbf{function} \textit{to\_int}(x: \textit{int32}): \textit{int}

\textbf{axiom} \textit{bounded\_int32}:
  \texttt{forall x: int32. -2^31 <= to\_int(x) < 2^31}

\textbf{val} \textit{int32\_add}(x: \textit{int32}, y: \textit{int32}): \textit{int32}
  \textbf{requires} -2^31 <= to\_int(x) + to\_int(y) < 2^31
  \textbf{ensures} to\_int(result) = to\_int(x) + to\_int(y)
Binary Search with overflow checking

See `bin_search_int32.mlw`
Binary Search with overflow checking

See `bin_search_int32.mlw`

**Application**

Used for translating mainstream programming language into Why3:

- From C to Why3: Frama-C, Jessie plug-in
  See `bin_search.c`
- From Java to Why3: Krakatoa
- From Ada to Why3: Spark2014
Floating-Point Arithmetic

- Limited range ⇒ exceptional behaviors.
- Limited precision ⇒ inaccurate results.
Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.
Width: $1 + w_e + w_m = 32$, or 64, or 128.
Bias: $2^{w_e - 1} - 1$. Precision: $p = w_m + 1$.

A floating-point datum

<table>
<thead>
<tr>
<th>sign $s$</th>
<th>biased exponent $e'$ ($w_e$ bits)</th>
<th>mantissa $m$ ($w_m$ bits)</th>
</tr>
</thead>
</table>

represents
Floating-Point Data

IEEE-754 Binary Floating-Point Arithmetic.
Width: \(1 + w_e + w_m = 32\), or 64, or 128.
Bias: \(2^{w_e-1} - 1\). Precision: \(p = w_m + 1\).

A floating-point datum

<table>
<thead>
<tr>
<th>sign (s)</th>
<th>biased exponent (e') ((w_e) bits)</th>
<th>mantissa (m) ((w_m) bits)</th>
</tr>
</thead>
</table>

represents

- if \(0 < e' < 2^{w_e} - 1\), the real \((-1)^s \cdot 1.m' \cdot 2^{e' - \text{bias}}\), normal
- if \(e' = 0\),
  - \(\pm 0\) if \(m' = 0\), zeros
  - the real \((-1)^s \cdot 0.m' \cdot 2^{-\text{bias}+1}\) otherwise, subnormal
- if \(e' = 2^{w_e} - 1\),
  - \((-1)^s \cdot \infty\) if \(m' = 0\), infinity
  - \textit{Not-a-Number} otherwise, NaN
Floating-Point Data

\[ \begin{align*}
(-1)^s & \times 2^{e-B} \times 1.f \\
(-1)^1 & \times 2^{198-127} \times 1.10010011110000111000000_2 \\
-2^{54} \times 206727 & \approx -3.7 \times 10^{21}
\end{align*} \]
Semantics for the Finite Case

IEEE-754 standard

A floating-point operator shall behave as if it was first computing the infinitely-precise value and then rounding it so that it fits in the destination floating-point format.

Rounding of a real number $x$:

Overflows are not considered when defining rounding: exponents are supposed to have no upper bound!
Specifications, main ideas

Same as with integers, we specify FP operations so that no overflow occurs.

constant max : real = 0x1.FFFFFEp127
predicate in_float32 (x:real) = abs x <= max
type float32
function to_real(x: float32): real
axiom float32_range: forall x: float32. in_float32 (to_real x)

function round32(x: real): real
(* ... axioms about round32 ... *)

function float32_add(x: float32, y: float32): float32
  requires in_float32(round32(to_real x + to_real y))
  ensures to_real result = round32 (to_real x + to_real y)
Specifications in practice

- Several possible rounding modes
- many axioms for `round32`, but incomplete anyway

Demo: `clock_drift.c`
Deductive verification nowadays

More native support in SMT solvers:

- *bitvectors* supported by CVC4, Z3, others
- *theory of floats* supported by Z3, CVC4, MathSAT

Using such a support for deductive program verification remains an open research topic

- Issues when bitvectors/fasts are mixed with other features: conversions, arrays, quantification

**Fumex et al. (2016)** C. Fumex, C. Dross, J. Gerlach, C. Marché. Specification and proof of high-level functional properties of bit-level programs. 8th NASA Formal Methods Symposium, LNCS 9690 Science