Ghost Code, Lemma Functions
More Data Types (lists, trees)
Handling Exceptions
Computer Arithmetic

Claude Marché

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

December 22nd, 2023
Outline

Reminders, Solutions to Exercises
  Reminder: Function Calls
  Reminder: Termination
  Reminder: Programs on Arrays

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
  Reminder: Function Calls
  Reminder: Termination
  Reminder: Programs on Arrays

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

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

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

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 \( \vec{w} \),
- \( \models Pre \rightarrow WP(\text{Body}, Post)[\text{Old} \leftarrow w_i] \) holds,

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

Remark: (mutually) recursive functions are allowed
Termination

- Loop *variant*
- *Variants* for (mutually) recursive function(s)
Home Work: McCarthy’s 91 Function

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

Find adequate specifications

```ml
let f91(n:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n <= 100 then f91(f91(n + 11)) else n - 10
```

Use canvas file `mccarthy.mlw`
Programs on Arrays

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

```
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)`
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
Home Work: Binary Search

\[ \text{low} = 0; \text{high} = a.length - 1; \]
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 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 \( v_2 \))

\[
\frac{v_1 > v_2}{\Sigma, \pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \leadsto \Sigma, \pi, ()}
\]

\[
\frac{v_1 \leq v_2}{\Sigma, \pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \leadsto \Sigma, \pi, (\text{let } i = v_1 \text{ in } e)\}; (\text{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e)}
\]
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 $i$:

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

So we deduce the Hoare logic rule

\[
\{ I \land v_1 \leq i \leq v_2 \} e \{ [i \leftarrow i + 1] \}
\]

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

Remark

Some rule should be stated for case \( v_1 > v_2 \), left as exercise

and then a rule for computing the WP:

\[
\begin{align*}
WP (& \text{for } i = v_1 \text{ to } v_2 \text{ invariant } I \text{ do } e, Q) = \\
v_1 \leq v_2 \land & \ [i \leftarrow v_1] \land \\
\forall \vec{v}, ( & \ (\forall i, I \land v_1 \leq i \leq v_2 \rightarrow WP(e, [i \leftarrow i + 1])) \land \\
& \ ([i \leftarrow v_2 + 1] \rightarrow Q))[w_j \leftarrow v_j]
\end{align*}
\]

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

`lin_search_for.mlw`
Outline

Reminders, Solutions to Exercises
   Reminder: Function Calls
   Reminder: Termination
   Reminder: Programs on Arrays

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:

```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:

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

(See Why3 file euclidean_rem.mlw)
Introducing Ghost Code

Example: Euclidean division / just compute the remainder:

\[
q \gets 0; r \gets x;
\]

\[
\text{while } r \geq y \text{ do }
\]

\[
\text{invariant } \{ x = q \cdot y + r \}
\]

\[
r \gets r - y; \quad q \gets q + 1
\]
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
```

Ghost code, ghost variables

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

See also euclidean_rem_with_ghost.mlw
Home Work: Bézout coefficients

- Extend the post-condition of Euclid’s algorithm for GCD to express the Bézout property:

  \[ \exists a, b, \text{result} = x \times a + y \times b \]

- Prove the program by adding appropriate ghost local variables

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

Let $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$

requires $Pre$

variant $var$, $\prec$

ensures $Post$

body $Body$

If the program $f$ is

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

then there exists a logic function:

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

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

and if $Body$ is a pure term then

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

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)
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 *without side effects* and *terminating*:
  
  \[
  \text{let } f(x_1: \tau_1, \ldots, x_n: \tau_n): \text{unit} \\
  \text{requires } Pre \\
  \text{variant } var, \prec \\
  \text{ensures } Post \\
  \text{body } Body
  \]
  
  then it is a proof of

  \[
  \forall x_1, \ldots, x_n. Pre \to Post
  \]

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

```ml
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

```
let rec lemma sum_of_odd_numbers_any (n:int)
  requires { n >= 0 }
  variant { n }
  ensures { sum_of_odd_numbers n = n * n }
  = if n > 0 then sum_of_odd_numbers_any (n-1)
```
Home work

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
  Reminder: Function Calls
  Reminder: Termination
  Reminder: Programs on Arrays

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 =
| C_1 \tau_{1,1} \cdots \tau_{1,n_1}
| \vdots
| C_k \tau_{k,1} \cdots \tau_{k,n_k}
```

- Pattern-matching with `match` and `with`:

```ocaml
match e with
| C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1
| \ldots
| C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k
end
```

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

- Sum types à la ML:
  ```ml
  type t =
    | C₁ τ₁₁ · · · τ₁ₙ₁
    | · · ·
    | Cₖ τₖ₁ · · · τₖₙₖ
  ```

- Pattern-matching with
  ```ml
  match e with
  | C₁(p₁, · · · , pₙ₁) → e₁
  | · · ·
  | Cₖ(p₁, · · · , pₙₖ) → eₖ
  end
  ```
Sum Types

- Sum types à la ML:
  ```ocaml
type t =
  | C_1 \tau_{1,1} \cdots \tau_{1,n_1}
  | : 
  | C_k \tau_{k,1} \cdots \tau_{k,n_k}
  ```

- Pattern-matching with
  ```ocaml
match e with
  | C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1
  | : 
  | C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k
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.
Sum Types: Example of Lists

```ocaml
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
  | Cons(x, r) -> append(rev(r), Cons(x, Nil))
end
```
Example: Efficient 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

```
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
  Reminder: Function Calls
  Reminder: Termination
  Reminder: Programs on Arrays

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 \textit{exn} a set of exception identifiers, declared as

\textbf{exception} \textit{exn} <\textbf{type}>

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

- Values (i.e. expressions that do not reduce): now either constants \( v \) or \( \text{raise } exn \)
- Context rules
  Assuming that sub-expressions are introduced with “let”, e.g. \( e_1 + e_2 \) written as

\[
\text{let } v_1 = e_1 \text{ in let } v_2 = e_2 \text{ in } v_1 + v_2
\]

then context rules are essentially given by the propagation of thrown exceptions inside “let”:

\[
\Sigma, \pi, (\text{let } x = \text{raise } exn \text{ in } e) \leadsto \Sigma, \pi, \text{raise } exn
\]
Operational Semantics: main rules

- Reduction of try-with:

\[
\begin{align*}
\Sigma, \pi, e & \leadsto \Sigma', \pi', \ e' \\
\Sigma, \pi, (\text{try } e \text{ with } \text{exn } \rightarrow e'') & \leadsto \Sigma', \pi', (\text{try } e' \text{ with } \text{exn } \rightarrow e'')
\end{align*}
\]
Operational Semantics: main rules

▶ 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:

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

▶ Exception handling:

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

\[
\text{exn} \neq \text{exn}' \\ 
\Sigma, \pi, (\text{try raise } \text{exn} \text{ with } \text{exn}' \rightarrow e) \rightsquigarrow \Sigma, \pi, \text{raise exn}
\]
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 WP modified to allow 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 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:

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

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

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

where \( w_1, \ldots, w_k \) is the set of assigned variables in \( e \) and \( v_1, \ldots, v_k \) are fresh logic variables.
Exercise: propose rules for

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

and

\[ \text{WP}(\text{try \, e}_1 \, \text{with \, exn} \rightarrow e_2, \, Q, \, exn_i \rightarrow R_i) \]
WP Rules

\[ WP(\text{raise } \text{exn}_k, Q, \text{exn}_i \rightarrow R_i) = R_k \]

\[ WP((\text{try } \text{e}_1 \text{ with } \text{exn} \rightarrow \text{e}_2), Q, \text{exn}_i \rightarrow R_i) = \]

\[ WP(\text{e}_1, Q, \left\{ \begin{array}{l} \text{exn} \rightarrow WP(\text{e}_2, Q, \text{exn}_i \rightarrow R_i) \\ \text{exn}_i \setminus \text{exn} \rightarrow R_i \end{array} \right. ) \]
Functions Throwing Exceptions

Generalized contract:

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

Extended WP rule for function call:

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

For each function defined with generalized contract

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

we have to check

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

```ocaml
exception NotSquare

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

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

\[ \text{low} = 0; \text{high} = a.length - 1; \]
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 bin_search_exc.mlw)
Outline

Reminders, Solutions to Exercises
  Reminder: Function Calls
  Reminder: Termination
  Reminder: Programs on Arrays

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

Floating-point numbers (32-, 64-bit):
- overflows
  - $2 \times 2 \times \cdots \times 2 \rightarrow +\text{inf}$
  - $-1/0 \rightarrow -\text{inf}$
  - $0/0 \rightarrow \text{NaN}$
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
  - \(0.1 + 0.1 + \cdots + 0.1 = 1.0 \rightarrow \text{false}\) \(10\text{times}\)
    (because \(0.1 \rightarrow 0.100000001490116119384765625\) in 32-bit)

See also arith.c
Some Numerical Failures

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

- 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

- 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

- 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 $77.1 \times 850$ as 100000.
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: $10^{-18}$. 
Google “Read All About It: Nearly All Binary Searches and Mergesorts are Broken”

```ocaml
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 `int32_add(x, y)`. 

```plaintext
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 `int` with an abstract type `int32`
- introduce a projection from `int32` to `int`
- axiom about the range of projections of `int32` elements
- replace all operations by abstract functions with preconditions

```plaintext
type int32
function to_int(x: int32): int
axiom bounded_int32:
    forall x: int32. -2^31 <= to_int(x) < 2^31

val int32_add(x: int32, y: int32): int32
    requires -2^31 <= to_int(x) + to_int(y) < 2^31
    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
- From Rust to Why3: Creusot
Floating-Point Arithmetic

- Limited range $\Rightarrow$ exceptional behaviors.
- Limited precision $\Rightarrow$ inaccurate results.
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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

represents

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

\[
\begin{align*}
(−1)^s × 2^{198−127} × 1.10010011110000111000000_2 \\
−2^{54} × 206727 \approx −3.7 × 10^{21}
\end{align*}
\]
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.

```plaintext
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/floats 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

That’s all for today, Merry Christmas!

- Next lecture on January 14th
- Several home work exercises to do
- Project text available on the web page, to be returned before February 8th, 2024