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

Outline

Function Calls

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

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

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

let $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  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 formula Q and any expression e,
if $\Sigma, \pi \models \mathrm{WP}(e, Q)$ then execution of Σ, π, e is *safe*

Remark: (mutually) recursive functions are allowed

Termination

- Loop *variant*
- *Variants* for (mutually) recursive function(s)

Home Work: McCarthy's 91 Function

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

Find adequate specifications

```
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)`

Home Work: 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*:

   $res \leftarrow -1$;
   for each i from 0 to $a.length - 1$: if $a[i] = v$ then $res \leftarrow i$;
   return res

See file `lin_search.mlw`

Home Work: Binary Search

$low = 0$; $high = a.length - 1$;
while $low \leq high$:
    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: $\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e$

Typing:

- i visible only in e, and is immutable
- e_1 and e_2 must be of type `int`, e must be of type `unit`

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

$$\frac{v_1 > v_2}{\Sigma, \pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \leadsto \Sigma, \pi, ()}$$

$$\frac{v_1 \leq v_2}{\Sigma, \pi, \texttt{for}\ i = v_1\ \texttt{to}\ v_2\ \texttt{do}\ e \leadsto \Sigma, \pi, \begin{array}{l}(\texttt{let}\ i = v_1\ \texttt{in}\ e);\\ (\texttt{for}\ i = v_1 + 1\ \texttt{to}\ v_2\ \texttt{do}\ e)\end{array}}$$

Home Work: “for” loops

Propose a Hoare logic rule for the `for` loop:

$$\frac{\{?\}e\{?\}}{\{?\}\texttt{for } i = v_1 \texttt{ to } v_2 \texttt{ do } e\{?\}}$$

Propose a rule for computing the WP:

$$\mathrm{WP}(\texttt{for } i = v_1 \texttt{ to } v_2 \texttt{ invariant } I \texttt{ do } e, Q) = ?$$

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:

$$
\begin{array}{l}
\{I[i \leftarrow v1]\} \\
i \leftarrow v1 \\
\{I\} \\
e \\
\{I[i \leftarrow i+1]\} \\
i \leftarrow i+1 \\
\{I\} \\
e \\
\vdots \\
\{I\} \\
e \\
\{I[i \leftarrow i+1]\} \\
i \leftarrow i+1 \\
\text{(* assuming now } i = v2\text{, last iteration *)} \\
\{I\}\text{(* where } i = v2 \text{ *)} \\
e \\
\{I[i \leftarrow i+1]\}\text{(* and still i=v2, hence *)} \\
\{I[i \leftarrow v2+1]\}
\end{array}
$$

Home Work: "for" loops

So we deduce the Hoare logic rule

$$\frac{\{I \wedge v_1 \leq i \leq v_2\}e\{I[i \leftarrow i+1]\}}{\{I[i \leftarrow v_1] \wedge v_1 \leq v_2\}\texttt{for } i = v_1 \texttt{ to } v_2 \texttt{ do } e\{I[i \leftarrow v_2+1]\}}$$

Remark

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

and then a rule for computing the WP:

$$\begin{array}{l}
\mathrm{WP}(\texttt{for } i = v_1 \texttt{ to } v_2 \texttt{ invariant } I \texttt{ do } e, Q) = \\
v_1 \leq v_2 \wedge I[i \leftarrow v_1] \wedge \\
\quad \forall \vec{v}, (\\
\qquad (\forall i, I \wedge v_1 \leq i \leq v_2 \rightarrow \mathrm{WP}(e, I[i \leftarrow i+1])) \wedge \\
\qquad (I[i \leftarrow v_2+1] \rightarrow Q))[w_j \leftarrow v_j]
\end{array}$$

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

`lin_search_for.mlw`

Outline

(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:

```
        r <- x;
while r >= y do
  invariant { exists q. x = q * y + r }
  r <- r - y;
```

(See Why3 file `euclidean_rem.mlw`)

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:

```
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:

$$\exists a, b, result = x * a + y * b$$

- 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*

```
let f(x1 : τ1, ..., xn : τn) : τ
  requires Pre
  variant var, ≺
  ensures Post
  body Body
```

then there exists a logic function:

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

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

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)
```

Example of Factorial

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

```
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*:

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

  then it is a proof of

$$\forall x_1, \ldots, x_n . \mathit{Pre} \rightarrow \mathit{Post}$$

- 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

```
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`

Home Work

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

Sum Types

- Sum types à la ML:

  type t =

  $\mid\ C_1\ \tau_{1,1} \cdots \tau_{1,n_1}$

  $\mid\ \vdots$

  $\mid\ C_k \tau_{k,1} \cdots \tau_{k,n_k}$

Sum Types

- Sum types à la ML:

  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 e with
  $| \, C_1(p_1, \cdots, p_{n_1}) \to e_1$
  $| \, \vdots$
  $| \, C_k(p_1, \cdots, p_{n_k}) \to e_k$
  end

Sum Types

- Sum types à la ML:

  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 e with
  $|\ C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1$
  $|\ \vdots$
  $|\ C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k$
  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

```
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.

```
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.

(problem 2 of the FoVeOOS verification competition in 2011, http://foveoos2011.cost-ic0701.org/verification-competition, continued nowadays as the yearly VerifyThis competition, https://www.pm.inf.ethz.ch/research/verifythis.html)

Outline

Exceptions

We extend the syntax of expressions with

$$
\begin{array}{rcl}
e & ::= & \texttt{raise}\ \mathit{exn} \\
  & \mid & \texttt{try}\ e\ \texttt{with}\ \mathit{exn} \rightarrow e
\end{array}
$$

with *exn* a set of exception identifiers, declared as

```
exception exn <type>
```

Remark: `<type>` can be omitted if it is `unit`
Example: linear search revisited in `lin_search_exc.mlw`

Operational Semantics

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

  $$\texttt{let}\ v_1 = e_1\ \texttt{in}\ \texttt{let}\ v_2 = e_2\ \texttt{in}\ v_1 + v_2$$

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

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

Operational Semantics: main rules

- Reduction of try-with:

$$\frac{\Sigma, \pi, e \leadsto \Sigma', \pi', e'}{\Sigma, \pi, (\texttt{try}\ e\ \texttt{with}\ exn \to e'') \leadsto \Sigma', \pi', (\texttt{try}\ e'\ \texttt{with}\ exn \to e'')}$$

Operational Semantics: main rules

- 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:

$$\Sigma, \pi, (\texttt{try raise}\ exn\ \texttt{with}\ exn \rightarrow e) \rightsquigarrow \Sigma, \pi, e$$

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

WP Rules

Function WP modified to allow exceptional post-conditions too:

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

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

WP Rules

Function WP modified to allow exceptional post-conditions too:

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

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

Extension of WP for simple expressions:

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

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

WP Rules

Extension of WP for composite expressions:

$$\mathrm{WP}(\texttt{let } x = e_1 \texttt{ in } e_2, Q, exn_i \rightarrow R_i) =$$
$$\mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, exn_i \rightarrow R_i)[\text{result} \leftarrow x], exn_i \rightarrow R_i)$$

$$\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q, exn_i \rightarrow R_i) =$$
$$\texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q, exn_i \rightarrow R_i)$$
$$\texttt{else } \mathrm{WP}(e_2, Q, exn_i \rightarrow R_i)$$

$$\mathrm{WP}\left(\begin{array}{c}\texttt{while } c \texttt{ invariant } I \\ \texttt{do } e\end{array}, Q, exn_i \rightarrow R_i\right) = I \wedge \forall \vec{v},$$
$$(I \rightarrow \texttt{if } c \texttt{ then } \mathrm{WP}(e, I, exn_i \rightarrow R_i) \texttt{ else } Q)[w_i \leftarrow v_i]$$

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

$$\mathrm{WP}(\texttt{raise}\ exn, Q, exn_i \rightarrow R_i)$$

and

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

WP Rules

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

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

$$\mathrm{WP}\left(e_1, Q, \left\{\begin{array}{l} exn \to \mathrm{WP}(e_2, Q, exn_i \to R_i) \\ exn_i \backslash exn \to R_i \end{array}\right.\right)$$

Functions Throwing Exceptions

Generalized contract:

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

Extended WP rule for function call:

$$\begin{aligned}\mathrm{WP}(f(t_1, \ldots, t_n), Q, E_k \rightarrow R_k) = {} & Pre[x_i \leftarrow t_i] \wedge \forall \vec{v}, \\ & (Post[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow Q[w_j \leftarrow v_j]) \wedge \\ & \bigwedge_k (Post_k[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow R_k[w_j \leftarrow v_j])\end{aligned}$$

Verification Conditions for programs

For each function defined with generalized contract

```
let f(x_1 : τ_1, ..., x_n : τ_n) : τ
  requires Pre
  writes w⃗
  ensures Post
  raises E_1 → Post_1
  ⋮
  raises E_n → Post_n
  body Body
```

we have to check

- Variables assigned in *Body* belong to $\vec{w}$
- $Pre \rightarrow \mathrm{WP}(Body, Post, E_k \rightarrow Post_k)[w_i@Old \leftarrow w_i]$ holds

Example: “Defensive” variant of ISQRT

```
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

Home Work

- Implement and prove binary search using also a immediate exit:

  $low = 0$; $high = a.length - 1$;
  while $low \leq high$:
  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 `bin_search_exc.mlw`)

Outline

Computers and Number Representations

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

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*
    - $2 \times 2 \times \cdots \times 2 \rightarrow +inf$
    - $-1/0 \rightarrow -inf$
    - $0/0 \rightarrow \texttt{NaN}$

Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may *overflow*
  - $2147483647 + 1 \to -2147483648$
  - $100000^2 \to 1410065408$
- floating-point numbers (32-, 64-bit):
  - *overflows*
    - $2 \times 2 \times \cdots \times 2 \to +inf$
    - $-1/0 \to -inf$
    - $0/0 \to$ `NaN`
  - *rounding errors*
    - $\underbrace{0.1 + 0.1 + \cdots + 0.1}_{10 times} = 1.0 \to$ `false`

      (because $0.1 \to 0.100000001490116119384765625$ 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 77.1×850 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.: $0.1 \simeq 209715 \cdot 2^{-24}$.
  Cumulated skew after 24h: -0.08s, distance: 160m.
  System was supposed to be rebooted periodically.

- 2007, Excel displays 77.1×850 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

Target Type: int32

- 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.

Safety Checking

Idea: replace all arithmetic operations by abstract functions with preconditions. $x + y$ becomes $\text{int32\_add}(x, y)$.

```
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

```
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.

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

sign s	biased exponent e' (w_e bits)	mantissa m (w_m bits)

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

sign s	biased exponent e' (w_e bits)	mantissa m (w_m bits)

represents

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

Floating-Point Data

$$
\begin{array}{ccccc}
\boxed{1} & & \boxed{11000110} & & \boxed{10010011110000111000000} \\
s & & e & & f \\
\downarrow & & \downarrow & & \downarrow \\
(-1)^s & \times & 2^{e-B} & \times & 1.f \\
 & & & & \\
(-1)^1 & \times & 2^{198-127} & \times & 1.10010011110000111000000_2
\end{array}
$$

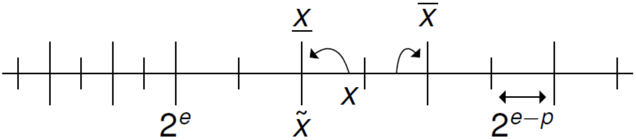
$$-2^{54} \times 206727 \approx -3.7 \times 10^{21}$$

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
- Specialized prover: Gappa http://gappa.gforge.inria.fr/

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

Boldo, Marché (2011) S. Boldo, C. Marché. Formal verification of numerical programs: from C annotated programs to mechanical proofs. Mathematics in Computer Science, 5:377–393

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