

# More data types (lists, trees) Handling Exceptions Computer Arithmetic

Claude Marché

Cours MPRI 2-36-1 "Preuve de Programme"

6 janvier 2021

## Outline

### Reminders, Solutions to Exercises

- Function calls
- Termination
- Axiomatizations, Ghost Code
- Ghost Functions, Lemma Functions
- Programs on Arrays

### 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
- Termination
- Axiomatizations, Ghost Code
- Ghost Functions, Lemma Functions
- Programs on Arrays

### Modeling Continued: Specifying More Data Types

- Sum Types
- Lists

### Exceptions

### Application: Computer Arithmetic

- Handling Machine Integers
- Floating-Point Computations

## Function Calls

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

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

### 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

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

we have

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

## Termination

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

## Home Work 1: McCarthy's 91 Function

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

Find adequate specifications

```
let f91(n:int): int  
  requires ?  
  variant ?  
  writes ?  
  ensures ?  
  body  
  if  $n \leq 100$  then  $f91(f91(n + 11))$  else  $n - 10$ 
```

Use canvas file [mccarthy.mlw](#)

## Why3 Logic Language

- ▶ (First-order) logic, built-in arithmetic (integers and reals)
- ▶ *Definitions* à la ML
  - ▶ logic (i.e. pure) *functions, predicates*
  - ▶ structured types, pattern-matching (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*

## Ghost Code

### Ghost code, ghost variables

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

## Home Work 2

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

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

- ▶ Prove the program by adding appropriate ghost local variables

Use canvas file [exo\\_bezout.mlw](#)

## Axiomatic Definitions

- ▶ logic functions, predicates without body
- ▶ axioms to specify their behavior
- ▶ axiomatic types
- ▶ **Risk of inconsistency**

Example: division

```
function div(real,real):real
axiom mul_div:
  forall x,y. y≠0 → div(x,y)*y = x
```

Error "Division by zero" can be modeled by an abstract function

```
val div_real(x:real,y:real):real
  requires y ≠ 0.0
  ensures result = div(x,y)
```

### Reminder

Execution blocks when an invalid annotation is met

## 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 τ1 ... τn : τ
lemma fspec : ∀x1, ..., xn. Pre → Post[result ← f(x1, ..., xn)]
```

and if  $Body$  is a pure term then

```
lemma fbody : ∀x1, ..., xn. Pre → f(x1, ..., xn) = 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(x1 : τ1, ..., xn : τn) : unit
  requires Pre
  variant var, <
  ensures Post
  body Body
```

then it is a proof of

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

- ▶ If  $f$  is recursive, it simulates a proof by induction

## Home work 3

Prove the helper lemmas stated for the fast exponentiation algorithm

See [power\\_int\\_lemma\\_functions.mlw](#)

## Home Work 4

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](#)

## Programs on Arrays

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

```
type array  $\alpha$  = { length : int; elts : int  $\rightarrow$   $\alpha$  }

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

val set (ref a:array  $\alpha$ ) (i:int) (v: $\alpha$ ) : unit
  requires  $0 \leq i < a.length$ 
  writes a
  ensures a.length = a@old.length  $\wedge$ 
         a.elts = store(a@old.elts,i,v)
```

- ▶  $a[i]$  interpreted as a call to `get(a,i)`
- ▶  $a[i] \leftarrow v$  interpreted as a call to `set(a,i,v)`

## Exercise: Search Algorithms

```
var a: array real

let search(v:real): int
  requires  $0 \leq a.length$ 
  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: `for  $i = e_1$  to  $e_2$  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  $v_2$ )

$$\frac{v_1 > v_2}{\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{v_1 \leq v_2}{\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \rightsquigarrow \Sigma, \Pi, (\text{let } i = v_1 \text{ in } e); (\text{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e)}$$

## Home Work: “for” loops

Propose a Hoare logic rule for the for loop:

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

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  $i$ :

```

{I[i ← v1]}
i ← v1
{I}
e
{I[i ← i + 1]}
i ← i + 1
{I}
e
⋮
{I}
e
{I[i ← i + 1]}
i ← i + 1
(* assuming now i = v2, last iteration *)
{I}(* where i = v2 *)
e
{I[i ← i + 1]}(* and still i=v2, hence *)
{I[i ← v2 + 1]}
    
```

## Home Work: “for” loops

So we deduce the Hoare logic rule

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

### Remark

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

and then a rule for computing the WP:

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

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

[lin\\_search\\_for.mlw](#)

## Outline

### Reminders, Solutions to Exercises

Function calls  
Termination  
Axiomatizations, Ghost Code  
Ghost Functions, Lemma Functions  
Programs on Arrays

### 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:

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

### ▶ 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.

## Sum Types: Example of Lists

```
type list α = Nil | Cons α (list α)  
  
function append(l1:list α,l2:list α): list α =  
  match l1 with  
  | Nil → l2  
  | Cons(x,l) → Cons(x, append(l,l2))  
  end  
  
function length(l:list α): int =  
  match l with  
  | Nil → 0  
  | Cons(_,r) → 1 + length r  
  end  
  
function rev(l:list α): list α =  
  match l with  
  | Nil → Nil  
  | Cons(x,r) → append(rev(r), Cons(x,Nil))  
  end
```

## “In-place” 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  $\alpha$  = Leaf | Node (tree  $\alpha$ )  $\alpha$  (tree  $\alpha$ )
```

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>)

## Outline

### Reminders, Solutions to Exercises

- Function calls
- Termination
- Axiomatizations, Ghost Code
- Ghost Functions, Lemma Functions
- Programs on Arrays

### 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 ::= raise exn
    | try e with exn → e
```

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: 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:

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

Normal execution:

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

Exception handling:

$$\Sigma, \Pi, (\text{try raise } \text{exn} \text{ with } \text{exn} \rightarrow e) \rightsquigarrow \Sigma, \Pi, e$$

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

## WP Rules

Function WP modified to allow **exceptional post-conditions** too:

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

Implicitly,  $R_k = \text{False}$  for any  $\text{exn}_k \notin \{\text{exn}_i\}$ .

Extension of WP for simple expressions:

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

$$\text{WP}(\text{assert } R, Q, \text{exn}_i \rightarrow R_i) = R \wedge Q$$

## WP Rules

Extension of WP for composite expressions:

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

$$\text{WP}(\text{if } t \text{ then } e_1 \text{ else } e_2, Q, \text{exn}_i \rightarrow R_i) = \begin{array}{l} \text{if } t \text{ then } \text{WP}(e_1, Q, \text{exn}_i \rightarrow R_i) \\ \text{else } \text{WP}(e_2, Q, \text{exn}_i \rightarrow R_i) \end{array}$$

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

where  $w_1, \dots, w_k$  is the set of assigned variables in  $e$  and  $v_1, \dots, v_k$  are fresh logic variables.

## WP Rules

Exercise: propose rules for

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

and

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

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

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

$$\text{WP}\left(e_1, Q, \left\{ \begin{array}{l} \text{exn} \rightarrow \text{WP}(e_2, Q, \text{exn}_i \rightarrow R_i) \\ \text{exn}_i \setminus \text{exn} \rightarrow R_i \end{array} \right. \right)$$

## Functions Throwing Exceptions

Generalized contract:

```
val  $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$   
  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:

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

## Verification Conditions for programs

For each function defined with generalized contract

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

we have to check

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

## Example: “Defensive” variant of ISQRT

```
exception NotSquare  
  
let isqrt(x:int): int  
  ensures result  $\geq 0 \wedge \text{sqr}(\text{result}) = x$   
  raises NotSquare  $\rightarrow \text{forall } n:\text{int}. \text{sqr}(n) \neq x$   
  body  
    if  $x < 0$  then raise NotSquare;  
    let ref res = 0 in  
    let ref sum = 1 in  
    while  $\text{sum} \leq x$  do  
      res  $\leftarrow$  res + 1; sum  $\leftarrow$  sum + 2 * res + 1  
    done;  
    if  $\text{sqr}(\text{res}) \neq x$  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

### Reminders, Solutions to Exercises

- Function calls
- Termination
- Axiomatizations, Ghost Code
- Ghost Functions, Lemma Functions
- Programs on Arrays

### 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**
  - ▶  $2147483647 + 1 \rightarrow -2147483648$
  - ▶  $100000^2 \rightarrow 1410065408$
- ▶ **floating-point numbers** (32-, 64-bit):
  - ▶ **overflows**
    - ▶  $2 \times 2 \times \dots \times 2 \rightarrow +inf$
    - ▶  $-1/0 \rightarrow -inf$
    - ▶  $0/0 \rightarrow NaN$
  - ▶ **rounding errors**
    - ▶  $\underbrace{0.1 + 0.1 + \dots + 0.1}_{10\text{times}} = 1.0 \rightarrow \text{false}$   
(because  $0.1 \rightarrow 0.10000001490116119384765625$  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.
- ▶ 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.

## 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:  $-0.08\text{s}$ , 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

## 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 `int32_add(x, y)`.

```
val int32_add(x: int, y: int): int
  requires  $-2^{31} \leq x + y < 2^{31}$ 
  ensures result = x + y
```

Unsatisfactory: range constraints 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.  $-2^{31} \leq \text{to\_int}(x) < 2^{31}$ 
```

```
val int32_add(x: int32, y: int32): int32
  requires  $-2^{31} \leq \text{to\_int}(x) + \text{to\_int}(y) < 2^{31}$ 
  ensures to_int(result) = to_int(x) + to_int(y)
```

## Binary Search with overflow checking

See [bin\\_search\\_int32.mlw](#)

### 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  $\Rightarrow$  **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

- ▶ if  $0 < e' < 2^{w_e} - 1$ , the real  $(-1)^s \cdot \overline{1.m'} \cdot 2^{e'-bias}$ , **normal**
- ▶ if  $e' = 0$ ,
  - ▶  $\pm 0$  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

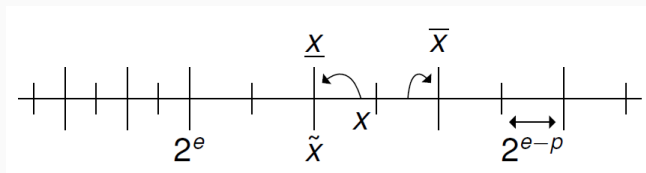
$\boxed{1}$	$\boxed{11000110}$	$\boxed{100100111110000111000000}$
$s$	$e$	$f$
$\downarrow$	$\downarrow$	$\downarrow$
$(-1)^s$	$\times 2^{e-B}$	$\times 1.f$
$(-1)^1$	$\times 2^{198-127}$	$\times 1.100100111110000111000000_2$
		$-2^{54} \times 206727 \approx -3.7 \times 10^{21}$

## 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