Soundness Theorem for a Complete Program

Assuming that for each function defined as
\[ \text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \]
requires \( \text{Pre} \)
writes \( \vec{w} \)
ensures \( \text{Post} \)
body \( \text{Body} \)

we have
- \( \triangleright \) variables assigned in \( \text{Body} \) belong to \( \vec{w} \),
- \( \triangleright \) \( \models \text{Pre} \rightarrow \text{WP(Body, Post)}[w@\text{Old} \leftarrow w] \) holds,
then for any formula \( Q \) and any expression \( e \), if \( \Sigma, \pi \models \text{WP(e, Q)} \) then execution of \( \Sigma, \pi, e \) is \textit{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

\[
\begin{align*}
\text{type} \ a &= \{ \text{length : int; elts : int -> 'a} \} \\
\text{val} \ \text{get} \ (\text{ref } a) \ (i : \text{int}) &= 'a \\
\text{requires} \ &0 <= i < a.\text{length} \\
\text{ensures} \ &\text{result} = \text{select}(a.\text{elts}, i) \\
\text{val} \ \text{set} \ (\text{ref } a) \ (i : \text{int}) \ (v : 'a) &= \text{unit} \\
\text{requires} \ &0 <= i < a.\text{length} \\
\text{writes} \ &a \\
\text{ensures} \ &a.\text{length} = a@\text{Old}.\text{length} \land \\
& a.\text{elts} = \text{store}(a@\text{Old}.\text{elts}, i, v) \\
\end{align*}
\]

- \( a[i] \) interpreted as a call to \( \text{get}(a, i) \)
- \( a[i] \leftarrow v \) interpreted as a call to \( \text{set}(a, i, v) \)

Exercise: Search Algorithms

\[
\begin{align*}
\text{var} \ a &: \text{array} \ \text{int} \\
\text{let} \ \text{search}(v : \text{int}) &: \text{int} \\
\text{requires} \ &0 <= a.\text{length} \\
\text{ensures} \ &\{ \ ? \ \} \\
\end{align*}
\]

1. Formalize postcondition: if \( v \) occurs in \( a \), between 0 and \( a.\text{length} - 1 \), then \( \text{result} \) is an index where \( v \) occurs, otherwise \( \text{result} \) is set to \(-1\)
2. Implement and prove \textit{linear search}:
   \[
   \begin{align*}
   \text{res} &\leftarrow -1; \\
   \text{for each } i \text{ from } 0 \text{ to } a.\text{length} - 1: \text{if } a[i] = v \text{ then } \text{res} \leftarrow i; \\
   \text{return } \text{res}
   \end{align*}
   \]

See file \texttt{lin_search.mlw}

Home Work: Binary Search

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

See file \texttt{bin_search.mlw}
Home Work: “for” loops

Syntax: for i = e₁ to e₂ do e

Typing:
- i visible only in e, and is immutable
- e₁ and e₂ must be of type \textit{int}, e must be of type \textit{unit}

Operational semantics:
(assuming e₁ and e₂ are values v₁ and v₂)
\[
\begin{align*}
\Sigma, \pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e & \leadsto \Sigma, \pi, () \\
\Sigma, \pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e & \leadsto \Sigma, \pi, (\text{let } i = v₁ \text{ in } e); \ (\text{for } i = v₁ + 1 \text{ to } v₂ \text{ do } e)
\end{align*}
\]

Home Work: “for” loops

Propose a Hoare logic rule for the \textit{for} loop:
\[
\{?\} e (? \{?\} \text{ for } i = v₁ \text{ to } v₂ \text{ do } e (?)
\]

Propose a rule for computing the WP:
\[
\text{WP}(\text{for } i = v₁ \text{ to } v₂ \text{ invariant } I \text{ do } e, Q) = ?
\]

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:
\[
\begin{align*}
\{[i \leftarrow v₁]\} & \ i \leftarrow v₁ \\
& \ \{i\} \\
& \ e \\
& \ \{[i \leftarrow i + 1]\} \\
& \ i \leftarrow i + 1 \\
& \ \{i\} \\
& \ eq \\
& \ \{[i \leftarrow i + 1]\} \\
& \ i \leftarrow i + 1 \\
& \ (* \ assuming \ now \ i = v₂, \ last \ iteration *) \\
& \ \{[i]\} (* \ where \ i = v₂ *) \\
& \ e \\
& \ \{[i \leftarrow i + 1]\} (* \ and \ still \ i = v₂, \ hence *) \\
& \ \{i \leftarrow v₂ + 1\}
\end{align*}
\]

Remark
Some rule should be stated for case v₁ > v₂, left as exercise

and then a rule for computing the WP:
\[
\text{WP}(\text{for } i = v₁ \text{ to } v₂ \text{ invariant } I \text{ do } e, Q) = \]
\[
\forall \overline{v}, (\forall i, I \land v₁ \leq i \leq v₂ \rightarrow \text{WP}(e, [i \leftarrow i + 1]) \land (i \leftarrow v₂ + 1 \rightarrow Q)) | w_j \leftarrow v_j)
\]

Additional exercise: use a for loop in the linear search example
\texttt{lin\_search\_for.mlw}
Case of loops

Solution: annotate loops with \textit{loop variants}

- a term that \textit{decreases at each iteration}
- for some \textit{well-founded ordering} $\prec$ (i.e. there is no infinite sequence $\text{val}_1 \prec \text{val}_2 \prec \text{val}_3 \prec \cdots$
- A typical ordering on integers:

$$ x \prec y \iff x < y \land 0 \leq y $$
Operational semantics

$$\left[\text{while } c \text{ invariant } / \text{ variant } t, \prec \text{ do } e \mapsto \Sigma, \pi \right]$$

$$\Sigma, \pi, L : \text{if } c$$
$$\text{then } (e; \text{assert } t \prec t@L)$$
$$\text{while } c \text{ invariant } / \text{ variant } t, \prec \text{ do } e$$
$$\text{else } ()$$

(new parts shown in red)

Weakest Precondition

$$\text{WP} (\text{while } c \text{ invariant } / \text{ variant } t, \prec \text{ do } e, Q) =$$

$$\Sigma, \pi, L : \text{if } c$$
$$\text{then } (e; \text{assert } t \prec t@L;$$
$$\text{while } c \text{ invariant } / \text{ variant } t, \prec \text{ do } e)$$
$$\text{else } ()$$

In practice with Why3

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

Examples

Exercise: find adequate variants

```latex
i \leftarrow 0;
\text{while } i \leq 100
\text{ variant ? }
\text{ do } i \leftarrow i+1
\text{ done; }
```

```latex
\text{while } \text{sum} \leq x
\text{ variant ? }
\text{ do }
\text{ res } \leftarrow \text{ res } + 1; \text{ sum } \leftarrow \text{ sum } + 2 * \text{ res } + 1
\text{ 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, ..., x_n : \tau_n) : \tau
\text{ requires } Pre
\text{ variant } var, \prec
\text{ writes } w
\text{ ensures } Post
\text{ body } Body
```

WP for function call:

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

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

Solution:

```ocaml
variant x
```

Case of mutual recursion

Assume two functions $f(x)$ and $g(y)$ 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 $g(i)$ the WP should include

$$v_g[y \leftarrow i] \prec v_f@Old$$

and symmetrically when $g$ calls $f$

Home Work 1: 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

```ocaml
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`

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:

```plaintext
r <- x;
while r >= y do
  invariant { exists q. x = q * y + r }
  r <- r - y;
  q <- q + 1
```

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:

```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 Why3 file `euclidean_rem.mlw`)

Home Work 2

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

```plaintext
∃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

then there exists a logic function:

```plaintext
let f(x_1: τ_1, ..., x_n: τ_n): τ
  requires Pre
  variant var, <
  ensures Post
  body Body
```

and if \( Body \) is a pure term then

```plaintext
lemma f_body: \forall x_1, ..., x_n. Pre \rightarrow f(x_1, ..., x_n) = Body
```

Example: axiom-free specification of factorial

```plaintext
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

```plaintext
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:

```ocaml
let fact_imp (x:int): int
  requires ?
  ensures ?
body
let ref y = 0
let ref res = 1
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**:

  ```ocaml
  let f(x_1 : \tau_1, ..., x_n : \tau_n) : unit
    requires Pre
    variant var, <
    ensures Post
body Body
  then it is a proof of

  \forall x_1, ..., x_n. Pre \rightarrow Post
  ```

- If \( f \) is recursive, it simulates a proof by induction

Example: sum of odds

```ocaml
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

```ocaml
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 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

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:

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

- Pattern-matching with

  \[
  \text{match } e \text{ with } 
  \begin{cases} 
  C_1(p_1, \ldots, p_{n_1}) \rightarrow e_1 \\
  \vdots \\
  C_k(p_1, \ldots, p_{n_k}) \rightarrow e_k
  \end{cases}
  \]

  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

```ml
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
```

“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

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

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
```

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

We extend the syntax of expressions with
\[
\begin{align*}
e & ::= \text{raise } \text{exn} \\
& | \text{try } e \text{ with } \text{exn} \rightarrow e
\end{align*}
\]
with \text{exn} a set of exception identifiers, declared as
\[
\text{exception } \text{exn} < \text{type}\]

Remark: \text{<type>} can be omitted if it is \text{unit}
Example: linear search revisited in \text{lin.search_exc.mlw}

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

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

Extension of WP for composite expressions:

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

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

\[
\text{WP}(\text{while } c \text{ invariant } I \text{ do } e, Q, \text{exn} \rightarrow R_i) = I \land \forall \bar{v}, \\
(l \rightarrow \text{if } c \text{ then } \text{WP}(e, I, \text{exn} \rightarrow R_i) \text{ else } Q)[w_i \leftarrow v_i]
\]

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

Functions Throwing Exceptions

Generalized contract:

\[
\text{val } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{requires } \text{Pre} \\
\text{writes } \bar{w} \\
\text{ensures } \text{Post} \\
\text{raises } E_1 \rightarrow \text{Post}_1 \\
\vdots \\
\text{raises } E_n \rightarrow \text{Post}_n
\]

Extended WP rule for function call:

\[
\text{WP}(f(t_1, \ldots, t_n), Q, E_k \rightarrow R_k) = P_{\text{Pre}}[x_i \leftarrow t_i] \land \forall \bar{v}, \\
(P_{\text{Post}}[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow Q[w_j \leftarrow v_j]) \land \\
\lor_k (P_{\text{Post}}[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow R_k[w_j \leftarrow v_j])
\]

Verification Conditions for programs

For each function defined with generalized contract

\[
\text{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\
\text{requires } \text{Pre} \\
\text{writes } \bar{w} \\
\text{ensures } \text{Post} \\
\text{raises } E_1 \rightarrow \text{Post}_1 \\
\vdots \\
\text{raises } E_n \rightarrow \text{Post}_n \\
\text{body } \text{Body}
\]

we have to check

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

```ml
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:

  ```ml```
  ```
  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 bin_search_exc.mlw)
  ```

Computers and Number Representations

- 32-, 64-bit signed `integers` in two-complement: may
  `overflow`
  ```
  ▶ 2147483647 + 1 → -2147483648
  ▶ 100000² → 1410065408
  ```
- Floating-point numbers (32-, 64-bit):
  ```
  ▶ overflows
  ▶ 2 * 2 * ... * 2 → +inf
  ▶ -1/0 → -inf
  ▶ 0/0 → NaN
  ▶ rounding errors
  ▶ 0.1 + 0.1 + ... + 0.1 = 1.0 → false
  ```
  (because `0.1 → 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.
- 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.

**Integer overflow: example of Binary Search**

- 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

**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 \texttt{int32\_add}(x, y).

```latex
val \texttt{int32\_add}(x: \texttt{int}, y: \texttt{int}): \texttt{int}
  \texttt{requires} -2^{31} \leq x + y < 2^{31}
  \texttt{ensures} \texttt{result} = x + y
```

Unsatisfactory: range contraints of integer must be added explicitly everywhere.

Safety Checking, Second Attempt

Idea:

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

```latex
type \texttt{int32}

function \texttt{to\_int}(x: \texttt{int32}): \texttt{int}
axiom \texttt{bounded\_int32}:
  \forall x: \texttt{int32}. -2^{31} \leq \texttt{to\_int}(x) < 2^{31}

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

Binary Search with overflow checking

See \texttt{bin\_search\_int32.mlw}

Floating-Point Arithmetic

- Limited range \(\Rightarrow\) exceptional behaviors.
- Limited precision \(\Rightarrow\) inaccurate results.
Floating-Point Data

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

A floating-point datum

\[
\begin{array}{c|c|c}
\text{sign} & \text{biased exponent } e' (w_e \text{ bits}) & \text{mantissa } m (w_m \text{ bits}) \\
\hline
\end{array}
\]

represents

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

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 \(\text{max} : \text{real} = 0x1.FFFFFEp127\)
predicate \(\text{in\_float32 (x:real)} = \text{abs} \ x \leq \text{max}\)
type \(\text{float32}\)
function \(\text{to\_real (x: float32)}: \text{real}\)
axiom \(\text{float32\_range: for all x: float32. in\_float32 (to\_real x)}\)
function \(\text{round32 (x: real)}: \text{real}\)
(* ... axioms about round32 ... *)
function \(\text{float32\_add (x: float32, y: float32)}: \text{float32}\)
requires \(\text{in\_float32 (round32 (to\_real x + to\_real y))}\)
ensures \(\text{to\_real result} = \text{round32 (to\_real x + to\_real y)}\)
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
