Complex data types (lists, arrays)

Exceptions

Computer Arithmetic

Claude Marché

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

15 janvier 2015

Outline

Reminder: labels, ghost, function calls, modularity, termination

Reminder: Advanced Modeling of Programs

Modeling Continued: Specifying More Data Types

Product Types

Sum Types

Axiomatic Type Definitions

Application: Programs on Arrays

Exceptions

Application: Computer Arithmetic

Handling Machine Integers

Floating-Point Computations

WP Rule of Function Call

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

requires $Pre$

writes $\bar{w}$

ensures $Post$

body $Body$

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

Modular proof

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

Assuming that for each function defined as

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

we have

- variables assigned in \text{Body} belong to \vec{w},
- \( \preceq \text{Pre} \Rightarrow \text{WP(Body, Post)[wi@Old ← wi]} \)

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

Remark: (mutually) recursive functions are allowed

Abstraction of sub-programs

- Keyword \text{val} introduces a function with a contract but without body
- \text{writes} clause is mandatory in that case

Example: Incrementation

```ml
val res: ref int
val incr(x:int):unit
  \text{writes } res \\
  \text{ ensures } res = res@Old + x
```

Modular proof
Program is proved correct under the assumption that the "val" are implementable.

Labels, Ghost Variables

- Labels and ghost variables are handy to refer to past program states in specifications

Exercise from last lecture:
- Extend the post-condition of Euclid algorithm to express the Bezout property:

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

- Prove the program by adding appropriate ghost local variables

Use canvas file \text{exo_bezout.mlw}

Termination

- Loop \text{variants}
- \text{Variants} for (mutually) recursive function

Example: McCarthy's 91 Function

\[
f91(n) = \begin{cases} 
  f91(f91(n + 11)) & \text{if } n \leq 100 \\
  n - 10 & \text{else}
\end{cases}
\]

Exercise: find adequate specifications.

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

Use canvas file \text{mccarthy.mlw}
Outline

Reminder: labels, ghost, function calls, modularity, termination

Reminder: Advanced Modeling of Programs

Modeling Continued: Specifying More Data Types
- Product Types
- Sum Types
- Axiomatic Type Definitions
- Application: Programs on Arrays

Exceptions

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

Advanced Modeling of Programs

- Direct definitions
- Axiomatic definitions
- Lemma functions to help provers

Exercise: Prove Fermat's little theorem for case $p = 3$:
$$\forall x, \exists y. x^3 - x = 3y$$

using a lemma function

Product Types

- Tuples types are built-in:
  ```
  type pair = (int, int)
  ```
- Record types can be defined:
  ```
  type point = { x:real; y:real }
  ```
- Fields are immutable.
- We allow let with pattern, e.g.
  ```
  let (a,b) = some pair in ...
  let { x = a; y = b } = some point in
  ```
- Dot notation for records fields, e.g.
  ```
  point.x + point.y
  ```
Sum Types

- Sum types à la ML:
  
  ```ml
  type t =
    | C1 \tau_1 \cdots \tau_{n_1}
    | 
    | \cdots
    | Ck \tau_k \cdots \tau_{nk}
  ```

- Pattern-matching with
  ```ml
  match e with
    | C1(p_1, \cdots, p_{n_1}) \rightarrow e_1
    | 
    | \cdots
    | Ck(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 allowed if recursive calls are on structurally smaller arguments.

Sum Types: Example of Lists

```ml
type list \alpha = Nil | Cons \alpha (list \alpha)

function append(l1:list \alpha, l2:list \alpha): list \alpha =
  match l1 with
    | Nil \rightarrow l2
    | Cons(x, l) \rightarrow Cons(x, append(l, l2))
  end

function length(l:list \alpha): int =
  match l with
    | Nil \rightarrow 0
    | Cons(_, r) \rightarrow 1 + length r
  end

function rev(l:list \alpha): list \alpha =
  match l with
    | Nil \rightarrow Nil
    | Cons(x, r) \rightarrow append(rev(r), Cons(x, Nil))
  end
```

“In-place” List Reversal

Exercise: fill the holes below.

```ml
val l: ref (list int)
let fun rev_append(r:list int)
variant ? writes ? ensures ?
body
  match r with
    | Nil \rightarrow ()
    | Cons(x, r) \rightarrow l := Cons(x, l); rev_append(r)
  end
let fun rev(r:list int)
  writes l ensures l = rev r
body ?
```
### Binary Trees

**type** tree α = Leaf | Node (tree α) α (tree α)

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


### Axiomatic Type Definitions

Type declarations of the form

```plaintext
type τ
```

Example: colors

```plaintext
type color
function blue: color
function red: color
axiom distinct: red ≠ blue
```

Polymorphic types:

```plaintext
type τ α₁ ··· αₖ
```

where α₁ ··· αₖ are type parameters.

### Example: Sets

```plaintext
type set α
function empty: set α
function single(α): set α
function union(set α, set α): set α
axiom union_assoc: forall x y z: set α.
  union(union(x, y), z) = union(x, union(y, z))
axiom union_comm: forall x y: set α.
  union(x, y) = union(y, x)
predicate mem(α, set α)
axiom mem_empty: forall x: α.
  ¬ mem(x, empty)
axiom mem_single: forall x y: α.
  mem(x, single(y)) ⇔ x = y
axiom mem_union: forall x: α, y z: set α.
  mem(x, union(y, z)) ⇔ mem(x, y) ∨ mem(x, z)
```

### Arrays as References on Pure Maps

Axiomatization of maps from int to some type α:

```plaintext
type map α
function select(map α, int): α
function store (map α, int, α): map α
axiom select_store_eq:
  forall a: map α, i: int, v: α.
  select(store(a, i, v), i) = v
axiom select_store_neq:
  forall a: map α, i j: int, v: α.
  i ≠ j → select(store(a, i, v), j) = select(a, j)
```

▶ Unbounded indexes.
▶ select(a, i) models the usual notation a[i].
▶ store denotes the functional update of a map.
Arrays as Reference on Maps

▶ Array variable: variable of type `ref (map α)`.
▶ In a program, the standard assignment operation

```plaintext
a[i] := e
```

is interpreted as

```plaintext
a := store(a,i,e)
```

Example: Swap

Permute the contents of cells `i` and `j` in an array `a`:

```plaintext
val a: array int

let fun swap(i:int,j:int) writes a
  ensures select(a,i) = select(a@Old,j) ∧
  select(a,j) = select(a@Old,i) ∧
  forall k:int. k ≠ i ∧ k ≠ j →
  select(a,k) = select(a@Old,k)

body
let tmp = select(a,i) in (* tmp :=a[i]*)
a := store(a,i,select(a,j)); (* a[i]:=a[j]*)
a := store(a,j,tmp) (* a[j]:=tmp *)
```

Simple Example

```plaintext
val a: ref (map int)

let fun test()
  writes a
  ensures select(a,0) = 13 (* a[0] = 13 *)

body
  a := store(a,0,13); (* a[0] := 13 *)
a := store(a,1,42) (* a[1] := 42 *)
```

Exercise: prove this program.

Exercises on Arrays

▶ Prove Swap using WP.
▶ Prove the program

```plaintext
let fun test()
  requires
  select(a,0) = 13 ∧ select(a,1) = 42 ∧
  select(a,2) = 64
  ensures
  select(a,0) = 64 ∧ select(a,1) = 42 ∧
  select(a,2) = 13

body swap(0,2)
```

▶ Specify, implement, and prove a program that increments by 1 all cells, between given indexes `i` and `j`, of an array of reals.
Arrays as Reference on pairs (length,map)

- Arrays have a **finite length**
- Goal: model "out-of-bounds" run-time errors
- Array variable: reference to a pair (length,pure map)
  - \( a[i] \) interpreted as \( \text{get}(a,i) \)
  - \( a[i] := v \) interpreted as \( \text{set}(a,i,v) \)

```why3
val get(a:array α,i:int):α
  requires 0 ≤ i < fst(a)
  ensures result = select(snd(a),i)

val set(a:array α,i:int,v:α):unit
  requires 0 ≤ i < fst(a)
  writes a
  ensures fst(a) = fst(a@Old) ∧
         snd(a) = store(snd(a@Old),i,v)
```

In Why3: use import array.Array
Syntax: \( a.length, a[i], a[i]<-v \)

Example: Swap in why3

```why3
val a: array int

let fun swap(i:int,j:int)
  requires { 0 ≤ i < a.length ∧ 0 ≤ j < a.length }
  writes { a }
  ensures { a[i] = old a[j] ∧ a[j] = old a[i] }
  ensures { forall k:int.
             0 ≤ k < a.length ∧ k ≠ i ∧ k ≠ j →
             a[k] = old a[k] }
  =
      let tmp = a[i] in
      a[i] <- a[j]; a[j] <- tmp
```

Exercise: Search Algorithms

```why3
val a: array real

let fun search (n:int, v:real): int
  requires 0 ≤ n
  ensures { ? }
  = ?
```

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

See `lin_search.mlw`

Outline

- Reminder: labels, ghost, function calls, modularity, termination
- Reminder: Advanced Modeling of Programs
- Modeling Continued: Specifying More Data Types
  - Product Types
  - Sum Types
  - Axiomatic Type Definitions
- Application: Programs on Arrays
- Exceptions
  - Application: Computer Arithmetic
    - Handling Machine Integers
    - Floating-Point Computations
Exceptions

We extend the syntax of expressions with

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

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}

Operational Semantics

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

Propagation of thrown exceptions:

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

Reduction of try-with:

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

Normal execution:

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

Exception handling:

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

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:

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

WP Rules

Extension of WP for composite expressions:

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

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

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

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

Exercise: propose rules for

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

and

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

\[
\text{WP}(\text{try } e_1 \text{ with } \text{exn} \rightarrow e_2, Q, \text{exn}_i \rightarrow R_i) = \text{WP}\left(e_1, Q, \left\{ \text{exn} \rightarrow \text{WP}(e_2, Q, \text{exn}_i \rightarrow R_i) \right\} \right)
\]

Functions Throwing Exceptions

Generalized contract:

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

Extended WP rule for function call:

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

Example: “Defensive” variant of ISQRT

```why3
exception NotSquare

let fun isqrt(x:int): int
  ensures result ≥ 0 ∧ sqr(result) = x
  raises NotSquare → forall n:int. n * 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 res * res ≠ x then raise NotSquare;
  res

See Why3 version in isqrt_exc.mlw
```

Exercises

- Re-implement and prove linear search in an array, using an exception to exit immediately when an element is found. (see lin_search_exc.mlw)
- Implement and prove binary search using also an immediate exit:
  
  - low = 0; high = n − 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.mlw)
Outline

Reminder: labels, ghost, function calls, modularity, termination

Reminder: Advanced Modeling of Programs

Modeling Continued: Specifying More Data Types
  Product Types
  Sum Types
  Axiomatic Type Definitions
  Application: Programs on Arrays

Exceptions

Application: Computer Arithmetic
  Handling Machine Integers
  Floating-Point Computations

Computers and Number Representations

- 32-, 64-bit signed integers in two-complement: may overflow
  - \( 2^{147483647 + 1} \rightarrow -2147483648 \)
  - \( 100000^2 \rightarrow 1410065408 \)

- floating-point numbers (32-, 64-bit):
  - overflows
    - \( 2 \times 2 \times \cdots \times 2 \rightarrow +\inf \)
    - \( -1/0 \rightarrow -\inf \)
    - \( 0/0 \rightarrow \text{NaN} \)
  - rounding errors
    - \( 0.1 + 0.1 + \cdots + 0.1 = 1.0 \rightarrow \text{false} \)
      (because \( 0.1 \rightarrow 0.10000001490116119384765625 \) in 32-bit)

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.: \( 0.1 \approx 209715 \cdot 2^{-24} \).
- Cumulated skew after 24h: \(-0.08s\), distance: 160m.
- System was supposed to be rebooted periodically.

- 2007, Excel displays 77.1 \( \times \) 850 as 100000.
- Bug in binary/decimal conversion.
- Failing inputs: 12 FP numbers.
- Probability to uncover them by random testing: \( 10^{-18} \).
Integer overflow: example of Binary Search

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

```ocaml
let l = ref 0 in
let u = ref (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 $\text{int32\_add}(x, y)$.

```ocaml
val int32_add(x: int, y: int): int
  requires -2^{31} ≤ 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 $\text{int32}$ coercible to it,
- all operations by abstract functions with preconditions,
- and add an axiom about the range of $\text{int32}$.

```ocaml
type int32
function of_int32(x: int32): int
axiom bounded_int32:
  forall x: int32. -2^{31} ≤ of_int32(x) < 2^{31}
val int32_add(x: int32, y: int32): int32
  requires -2^{31} ≤ of_int32(x) + of_int32(y) < 2^{31}
  ensures of_int32(result) = of_int32(x) + of_int32(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 ⇒ exceptional behaviors.
- Limited precision ⇒ inaccurate results.

Floating-Point Data

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

A floating-point datum

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

represents

- if \(0 < e' < 2^{w_e} - 1\), the real \((-1)^s \cdot 1.m' \cdot 2^{e' - \text{bias}}\), normal

- if \(e' = 0\):
  - ±0 if \(m' = 0\), zeros
  - the real \((-1)^s \cdot 0.m' \cdot 2^{-\text{bias}+1}\) otherwise, subnormal

- if \(e' = 2^{w_e} - 1\):
  - \((-1)^s \cdot \infty\) if \(m' = 0\), infinity
  - *Not-a-Number* otherwise. NaN
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 of_float32(x: float32): real
axiom float32_range: forall x: float32. in_float32 (of_float32 x)

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

function float32_add(x: float32, y: float32): float32
requires in_float32(round32(of_float32 x + of_float32 y))
ensures of_float32 result = round32 (of_float32 x + of_float32 y)
```

Specifications in practice

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

Demo: clock_drift.c