Lemma functions
More data types (lists)
Exceptions
Computer Arithmetic

Claude Marché

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

7 janvier 2016

Outline

Outline

Labels, Ghost Variables

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

Home work from the last lecture:

- Extend the post-condition of Euclid algorithm to express the Bezout property:

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

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

Function Call

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

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

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 fun f(x1 : τ1, ..., xn : τn) : τ
  requires Pre
  writes w⃗
  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 *variants*
- *Variants* for (mutually) recursive function

Example: McCarthy's 91 Function

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

Exercise: find adequate specifications.

```
let fun 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

Advanced Modeling of Programs

Direct definitions

- logic functions, predicates with body
- *total* functions, no recursion allowed

Axiomatic definitions

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

Important case: *arrays*

- applicative maps as an axiomatic type
- array = reference to a pair (length, pure map)
- handling of out-of-bounds index check

Home Work: Binary Search

$low = 0$; $high = n - 1$;
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)

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

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

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:

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

Outline

Automated Provers Capabilities

SMT solvers like Alt-Ergo, CVC4, Z3 are the best ones for deductive verification because:

- they understand (typed) first-order logic
- they have built-in support for the equality predicate
- they support integer and real arithmetic
- they allow user definitions and axiomatizations

Weaknesses:

- incompleteness (this logic is too powerful to be decidable)
- weak support for quantifiers (sometimes FO provers like Vampire, Spass, E can be better)
- existential goals are typically hard: provers cannot guess the "witness"
- no support for advanced reasoning like *induction*

Some hints to help provers

- Simplify the goal: inline definitions, compute what can be computed
- Split the goal into subgoals (hint: try to inline definition of the head symbol of the goal)
- help the provers by
  - introduce extra assertions in the code ("local lemmas")
  - introduce extra lemmas before the code
  - prove extra lemmas using *lemma functions*

Lemma functions

- Basic idea: if a program function is *without side effects* and *terminating* :

  `let fun` $f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau$
  `requires` *Pre*
  `variant` *var*, $\prec$
  `ensures` *Post*
  `body` *Body*

  then it is a (constructive) proof of

  $$\forall x_1, \dots, x_n, \exists result, Pre \Rightarrow Post$$

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

Example: power function

```
function power int int : int
axiom power_0 : forall x:int. power x 0 = 1
axiom power_s : forall x n:int. n ≥ 0 →
  power x (n+1) = x * power x n

lemma power_1 : forall x:int. power x 1 = x

lemma sqrt4_256 : exists x:int. power x 4 = 256

lemma power_sum : forall x n m: int. 0 ≤ n ∧ 0 ≤ m →
  power x (n+m) = power x n * power x m
```

See file `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

Outline

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:

  type t =
  $\mid\ C_1\ \tau_{1,1} \cdots \tau_{1,n_1}$
  $\mid\ \vdots$
  $\mid\ C_k \tau_{k,1} \cdots \tau_{k,n_k}$

- Pattern-matching with

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

```
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 l: ref (list int)

let fun 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 fun reverse(r:list int)
  writes l  ensures l = rev r
body ?
```

See `rev.mlw`

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.

(problem 2 of the FoVeOOS verification competition in 2011, http://foveoos2011.cost-ic0701.org/verification-competition)

Outline

Exceptions

We extend the syntax of expressions with

$$\begin{array}{rcl} e & ::= & \texttt{raise}\ exn \\ & | & \texttt{try}\ e\ \texttt{with}\ exn \to e \end{array}$$

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: either constants v or $\texttt{raise}\ exn$

Propagation of thrown exceptions:

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

Reduction of try-with:

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

Normal execution:

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

Exception handling:

$$\Sigma, \Pi, (\texttt{try raise}\ exn\ \texttt{with}\ exn \to e) \rightsquigarrow \Sigma, \Pi, e$$

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

WP Rules

Function WP modified to allow exceptional post-conditions too:

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

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

Extension of WP for simple expressions:

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

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

WP Rules

Extension of WP for composite expressions:

$$\begin{aligned}&\mathrm{WP}(\texttt{let } x = e_1 \texttt{ in } e_2, Q, exn_i \to R_i) = \\ &\qquad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, exn_i \to R_i)[\text{result} \leftarrow x], exn_i \to R_i)\end{aligned}$$

$$\begin{aligned}&\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q, exn_i \to R_i) = \\ &\qquad \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q, exn_i \to R_i) \\ &\qquad\qquad \texttt{else } \mathrm{WP}(e_2, Q, exn_i \to R_i)\end{aligned}$$

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

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

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

and

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

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

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

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

Functions Throwing Exceptions

Generalized contract:

$$\begin{array}{l}\texttt{val } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \\ \quad \texttt{requires } Pre \\ \quad \texttt{writes } \vec{w} \\ \quad \texttt{ensures } Post \\ \quad \texttt{raises } E_1 \to Post_1 \\ \quad \vdots \\ \quad \texttt{raises } E_n \to Post_n\end{array}$$

Extended WP rule for function call:

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

Example: “Defensive” variant of ISQRT

```
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 a immediate exit:

  $low = 0$; $high = n - 1$;
  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

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

Reminder: Advanced Modeling of Programs

About Automated Provers Capabilities

Modeling Continued: Specifying More Data Types
- Product 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*
    - $2 \times 2 \times \cdots \times 2 \rightarrow +inf$
    - $-1/0 \rightarrow -inf$
    - $0/0 \rightarrow$ `NaN`
  - *rounding errors*
    - $\underbrace{0.1 + 0.1 + \cdots + 0.1}_{10 times} = 1.0 \rightarrow$ `false`

      (because $0.1 \rightarrow 0.100000001490116119384765625$ 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 77.1×850 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 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 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 *int*32 coercible to it,
- all operations by abstract functions with preconditions,

and add an axiom about the range of *int*32.

```
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 $\Rightarrow$ 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 $e' = 0$,
  - ± 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

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

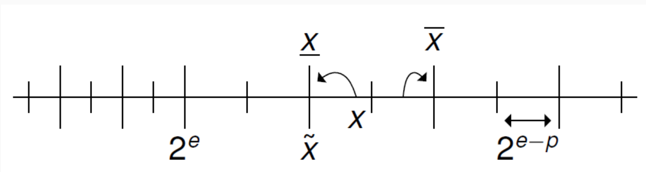
$$-2^{54} \times 206727 \approx -3.7 \times 10^{21}$$

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 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
- Specialized prover: Gappa `http://gappa.gforge.inria.fr/`
- Theory of floats in SMT solvers in the near future

Demo: `clock_drift.c`

Notes on Course Schedule

- Regular lecture on next week
- No lecture on January 21th
- Regular lectures on January 28th and February 4th by Arthur
- February 11th: lab session from 9h30 to 12h: help with the project, same room as usual, bring your laptop
- Regular lectures on February 18th and 25th by Arthur
- Exam: either March 3rd or March 10th.