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:
  \[ \exists a, b, \text{result} = x * a + y * b \]

- 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

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

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

we have

- variables assigned in Body belong to \( \vec{w} \),
- \( \models Pre \Rightarrow WP(Body, Post)[w_i@Old \leftarrow w_i] \) 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 variants
- Variants for (mutually) recursive function

Example: McCarthy’s 91 Function

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

Exercise: find adequate specifications.

```ml
let fun 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

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
### 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 ≤ 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 = e₁ to e₂ do e`

**Typing:**
- `i` visible only in `e`, and is immutable
- `e₁` and `e₂` must be of type `int`, `e` must be of type `unit`

**Operational semantics:**
(assuming `e₁` and `e₂` are values `v₁` and `v₂`)

\[
\begin{align*}
\Delta, \Pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e &\Rightarrow \Delta, \Pi, () \\
\end{align*}
\]

\[
\begin{align*}
\Delta, \Pi, \text{for } i = v₁ \text{ to } v₂ \text{ do } e &\Rightarrow \Delta, \Pi, (\text{let } i = v₁ \text{ in } e); \\
&\quad (\text{for } i = v₁ + 1 \text{ to } v₂ \text{ do } e)
\end{align*}
\]

### Home Work: “for” loops

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

```
{?} e(?)
{?} for i = v₁ to v₂ do e(?)
```

Propose a rule for computing the WP:

```
WP(for i = v₁ to v₂ invariant I do e, Q) =?
```
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

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

  then it is a (constructive) proof of

  \[ \forall x_1, \ldots, x_n. \exists result. Pre \Rightarrow Post \]

- 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 \geq 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 \leq n \land 0 \leq 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

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

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:
  \[ \text{type} \ t = \]
  \[ | \ C_1 \tau_{1,1} \cdots \tau_{1,n_1} \]
  \[ | \vdots \]
  \[ | C_k \tau_{k,1} \cdots \tau_{k,n_k} \]

- Pattern-matching with
  \[ \text{match} \ e \ 	ext{with} \]
  \[ | C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1 \]
  \[ | \vdots \]
  \[ | C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k \]
end

- Extended pattern-matching, wildcard: _

Recursive Sum Types

- Sum types can be recursive.
- Recursive definitions of functions or predicates 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.


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

Exceptions

We extend the syntax of expressions with

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

with \textit{exn} a set of exception identifiers, declared as

\text{exception} \ exn \ <\text{type}>

Remark: \textit{<type>} can be omitted if it is \textit{unit}

Example: linear search revisited in \texttt{lin_search_exc.mlw}

Operational Semantics

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

Propagation of thrown exceptions:

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

Reduction of try-with:

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

Normal execution:

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

Exception handling:

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

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

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

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

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

\[
\Sigma, \Pi, (\text{try } \text{raise exn with } exn' \rightarrow e) \leadsto \Sigma, \Pi, \text{raise exn}
\]
WP Rules

Function WP modified to allow exceptional post-conditions too:

$$ WP(e, Q, exn_i \rightarrow R_i) $$

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

Extension of WP for simple expressions:

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

$$ WP(\text{assert } R, Q, exn_i \rightarrow R_i) = R \land Q $$

WP Rules

Extension of WP for composite expressions:

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

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

$$ \text{if } t \text{ then } WP(e_1, Q, exn_i \rightarrow R_i) \quad \text{else } WP(e_2, Q, exn_i \rightarrow R_i) $$

$$ WP(\text{while } c \text{ invariant } I \text{ do } e, Q, exn_i \rightarrow R_i) = $$

$$ I \land \forall \vec{v}, (l \Rightarrow \text{if } c \text{ then } WP(e, I, exn_i \rightarrow R_i) \text{ else } Q)[w_i \leftarrow v_i] $$

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

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

and

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

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

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

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

WP Rules

Extension of WP for composite expressions:

$$ WP(\text{let } x = e_1 \text{ in } e_2, Q, exn_i \rightarrow R_i) = $$

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

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

$$ \text{if } t \text{ then } WP(e_1, Q, exn_i \rightarrow R_i) \quad \text{else } WP(e_2, Q, exn_i \rightarrow R_i) $$

$$ WP(\text{while } c \text{ invariant } I \text{ do } e, Q, exn_i \rightarrow R_i) = $$

$$ I \land \forall \vec{v}, (l \Rightarrow \text{if } c \text{ then } WP(e, I, exn_i \rightarrow R_i) \text{ else } Q)[w_i \leftarrow v_i] $$

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

Functions Throwing Exceptions

Generalized contract:

$$ \text{val } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau $$

$$ \text{requires } Pre $$

$$ \text{writes } \vec{w} $$

$$ \text{ensures } Post $$

$$ \text{raises } E_1 \rightarrow Post_1 $$

$$ \vdots $$

$$ \text{raises } E_n \rightarrow Post_n $$

Extended WP rule for function call:

$$ WP(f(t_1, \ldots, t_n), Q, E_k \rightarrow R_k) = \text{Pre}[x_i \leftarrow t_i] \land \forall \vec{v}, (\text{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j] \Rightarrow Q[w_j \leftarrow v_j]) \land \land_k (\text{Post}_k[x_i \leftarrow t_i, w_j \leftarrow v_j] \Rightarrow R_k[w_j \leftarrow v_j]) $$
Example: “Defensive” variant of ISQRT

```ocaml
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_exc.mlw`)

### 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
    - \(0.1 + 0.1 + \cdots + 0.1 = 1.0 \rightarrow false\)
      (because \(0.1 \rightarrow 0.10000001490116119384765625\) in 32-bit)
  See also `arith.c`

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
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”

```plaintext
let l = ref 0 in
let u = ref (a.length - 1) in
while l \leq 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

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: $-0.08$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: $10^{-18}$.

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 \( \text{int32}_\text{add}(x, y) \).

```plaintext
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 \textit{int} with an abstract type \textit{int32} coercible to it,
- all operations by abstract functions with preconditions, and add an axiom about the range of \textit{int32}.

```plaintext
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 \texttt{bin\_search\_int32.mlw}

Floating-Point Arithmetic

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

Application

Used for translating mainstream programming language into Why3:

- From C to Why3: Frama-C, Jessie plug-in
  See \texttt{bin\_search.c}
- From Java to Why3: Krakatoa
- From Ada to Why3: Spark2014
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

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

represents

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

Specifications, main ideas

Same as with integers, we specify FP operations so that no overflow occurs.

<table>
<thead>
<tr>
<th>constant max : real</th>
<th>= 0x1.FFFFEp127</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate in_float32 (x:real)</td>
<td>(= ) abs (x \leq) max</td>
</tr>
<tr>
<td>type float32</td>
<td></td>
</tr>
<tr>
<td>function of_float32(x: float32): real</td>
<td></td>
</tr>
<tr>
<td>axiom float32.range: forall x: float32. in_float32 (of_float32 x)</td>
<td></td>
</tr>
<tr>
<td>function round32(x: real): real</td>
<td></td>
</tr>
<tr>
<td>(* ... axioms about round32 ...)</td>
<td></td>
</tr>
<tr>
<td>function float32.add(x: float32, y: float32): float32</td>
<td></td>
</tr>
<tr>
<td>requires in_float32(round32(of_float32 x + of_float32 y))</td>
<td></td>
</tr>
<tr>
<td>ensures of_float32 result = round32 (of_float32 x + of_float32 y)</td>
<td></td>
</tr>
</tbody>
</table>
Specifications in practice

- Several possible rounding modes
- Many axioms for `round32`, but incomplete anyway
- 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.