Outline

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

Reminder: Advanced Modeling of Programs

Reminder: Programs on Arrays

Modeling Continued: Specifying More Data Types
- Product Types
- Sum Types
- Lists

Exceptions

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

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

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`
**Function Call**

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

**Soundness Theorem for a Complete Program**

Assuming that for each function defined as

```ml
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}`,
- \( \models Pre \Rightarrow WP(\text{Body, Post})[w_i@Old \leftarrow w_i] \) 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

```ml
f_91(n) = if n \leq 100 then f_91(f_91(n + 11)) 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

Reminder: Programs on Arrays

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

Lemma functions
- When automated provers fail: Write a program to construct a proof
- Example: construct witnesses for existential quantification
- Example: proof by induction using recursive functions

Home Work 3

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

Reminder: Programs on Arrays

Programming Continued: Specifying More Data Types
Product Types
Sum Types
Lists

Exceptions

Application: Computer Arithmetic
Handling Machine Integers
Floating-Point Computations

Programs on Arrays

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

```plaintext
val get(a:array α,i:int):α
requires 0 ≤ i < fst(a)
enforces 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) \land
snd(a) = store(snd(a@Old),i,v)
```

- \( a[i] \) interpreted as a call to \( get(a,i) \)
- \( a[i] := v \) interpreted as a call to \( set(a,i,v) \)
Exercise: Search Algorithms

```ml
var 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*:
   
   ```
   res := -1;
   for each i from 0 to n - 1: if a[i] = v then res := i;
   return res
   ```

See file `lin_search.mlw`

Home Work: Binary Search

```ml
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 = 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 \( v_2 \))

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \leadsto & \Sigma, \Pi, () \\
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Pi, \text{for } i = v_1 \text{ to } v_2 \text{ do } e \leadsto & \Sigma, \Pi, (\text{let } i = v_1 \text{ in } e); \\
& (\text{for } i = v_1 + 1 \text{ to } v_2 \text{ do } e)
\end{align*}
\]

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

\[
\begin{align*}
{?} & e(?) \\
\{?\} & \text{for } i = v_1 \text{ to } v_2 \text{ do } e(?)
\end{align*}
\]

Propose a rule for computing the WP:

\[
\text{WP}(\text{for } i = v_1 \text{ to } v_2 \text{ invariant } l \text{ do } e, Q) = ?
\]

Additional exercise: use a for loop in the linear search example
Outline
Reminder: labels and ghost variables, function calls and modularity, termination
Reminder: Advanced Modeling of Programs
Reminder: Programs on Arrays
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:
    type t =
    | C_1 \tau_1,1 \cdots \tau_1,n_1
    | :
    | C_k \tau_{k,1} \cdots \tau_{k,n_k}
  Pattern-matching with
    match e with
    | C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1
    | :
    | 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
  Must termination (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 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.

```ml
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.ml
```

**Binary Trees**

```ml
type tree α = Leaf | Node (tree α) α (tree α)
```

Home work: 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

Reminder: Programs on Arrays

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

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

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 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:

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

Normal execution:

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

Exception handling:

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

\[
\text{exn} \neq \text{exn'}
\]

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

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:

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

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

\[
\text{WP} \left( \text{while } c \text{ invariant } I \right. \\
\text{do } e \\
\left. \right) = I \land \forall v_i
\]

\[
(l \Rightarrow \text{if } c \text{ then } \text{WP}(e, l, \text{exn}_i \rightarrow R_i) \text{ else } Q)[w_i \leftarrow v_i]
\]

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

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

and

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

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

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

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. 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

Functions Throwing Exceptions

Generalized contract:

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

Extended WP rule for function call:

\[\text{WP}(f(t_1, \ldots, t_n), Q, E_k \rightarrow R_k) = \text{Pre}[x_i ← t_i] \land \forall \vec{w}, (\text{Post}[x_i ← t_i, w_j ← v_j] \Rightarrow Q[w_j ← v_j]) ∧ \land_k (\text{Post}_k[x_i ← t_i, w_j ← v_j] ⇒ R_k[w_j ← v_j])\]

Home Work

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

(see bin_search_exc.mlw)
Outline

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

Reminder: Advanced Modeling of Programs

Reminder: Programs on Arrays

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
    - $2 \times 2 \times \cdots \times 2 \rightarrow \inf$
    - $-1/0 \rightarrow -\inf$
    - $0/0 \rightarrow \text{NaN}$
  - rounding errors
    - $0.1 + 0.1 + \ldots + 0.1 = 1.0 \rightarrow false$ (10 times)
      (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.: $0.1 \approx 209715 \cdot 2^{-24}$.
  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”

```
let l = ref 0 in
let u = ref (Array.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 `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 ≤ to_int(x) < 2^31

val int32_add(x: int32, y: int32): int32
  requires -2^31 ≤ to_int(x) + 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 ⇒ 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

```
| sign s | biased exponent \( e' \) (\( w_e \) bits) | mantissa \( m \) (\( w_m \) bits) |
```

represents

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

Floating-Point Data

```
\[
\begin{array}{cccc}
1 & 11000110 & 10010011110000111000000 \\
\downarrow & \downarrow & \downarrow \\
(-1)^s & 2^{e-B} & 1.f \\
(-1)^l & 2^{198-127} & 1.10010011110000111000000000000002 \\
\end{array}
\]
```

\(-2^{54} \times 206727 \approx -3.7 \times 10^{21}\)
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 $x$ $\leq$ 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
(function float32_add(x: float32, y: float32): float32
(* ... axioms about round32 ... *)
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, 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
