Modeling, Specification Languages, Array Programs

Guillaume Melquiond

MPRI 2-36-1 “Preuve de Programme”

January 16th, 2012
Reminder of Previous Lectures

- ML-like programs:
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
Remainder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
  - recursive functions.
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
  - recursive **functions**.

- Program verification:
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
  - recursive functions.

- **Program verification:**
  - Hoare logic: safety, validity, termination,
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
  - recursive **functions**.

- **Program verification:**
  - Hoare logic: safety, validity, termination,
  - weakest precondition computations,
Reminder of Previous Lectures

- **ML-like programs:**
  - mutable variables,
  - expressions with side effects,
  - exceptions,
  - recursive **functions**.

- **Program verification:**
  - Hoare logic: safety, validity, termination,
  - weakest precondition computations,
  - **modular** verification: function **contract**.
Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates
About Specification Languages

Specification languages:

- Algebraic Specifications: CASL, Larch
- Set theory: VDM, Z notation, Atelier B
- Higher-Order Logic: PVS, Isabelle/HOL, HOL4, Coq
- Object-Oriented: Eiffel, JML, OCL
- ...
About Specification Languages

Specification languages:

- Algebraic Specifications: CASL, Larch
- Set theory: VDM, Z notation, Atelier B
- Higher-Order Logic: PVS, Isabelle/HOL, HOL4, Coq
- Object-Oriented: Eiffel, JML, OCL
- …

Case of Why3, ACSL, Dafny: trade-off between

- expressiveness of specifications,
- support by automated provers.
Why3 Logic Language

- First-order logic, with type polymorphism à la ML
- Built-in arithmetic (integers and reals)
- Definitions à la ML
  - Functions, predicates
  - Structured types, pattern-matching
- Axiomatizations
- Inductive predicates
Logic Symbols

Functions defined as

\[
\text{function } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e
\]

Predicate defined as

\[
\text{predicate } p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e
\]

where \( \tau_i, \tau \) are not reference types.

- No recursion allowed
- No side effects
- Define total functions and predicates
function $sqr(x:\text{int}) = x \times x$

predicate prime($x:\text{int}$) =

\[
x \geq 2 \land \\
\text{forall } y, z:\text{int. } y \geq 0 \land z \geq 0 \land x = y \times z \rightarrow \\
y = 1 \lor z = 1
\]
Axiomatic Definitions

Function and predicate declarations of the form

\[
\text{function } f(\tau, \ldots, \tau_n) : \tau \\
\text{predicate } p(\tau, \ldots, \tau_n)
\]

together with axioms

\[
\text{axiom } id : \text{formula}
\]

specify that \( f \) (resp. \( p \)) is any symbol satisfying the axioms.
Axiomatic Definitions

Example: division

```
function div(real,real):real
axiom mul_div: forall x,y. y≠0 →
    div(x,y)*y = x
```
Axiomatic Definitions

Example: division

\begin{verbatim}
function div(real, real): real
axiom mul_div: forall x, y. y \neq 0 \rightarrow
div(x, y) \times y = x
\end{verbatim}

Example: factorial

\begin{verbatim}
function fact(int): int
axiom fact0: fact(0) = 1
axiom factn: forall n: int. n \geq 1 \rightarrow
fact(n) = n \times fact(n-1)
\end{verbatim}
Functions/predicates are typically underspecified. 
⇒ model partial functions in a logic of total functions.
Axiomatic Definitions

▶ Functions/predicates are typically **underspecified**.  
⇒ model *partial* functions in a logic of total functions.

▶ About soundness: axioms may introduce **inconsistencies**.
Exercise: Find appropriate precondition, postcondition, loop invariant, and variant, for this program:

```plaintext
function fact_imp (x:int): int
    requires ?
    ensures ?
body
    let ref y = 0 in
    let ref res = 1 in
    while y < x do
        y := y + 1;
        res := res * y
    done;
res
```
Axiomatic Type Definitions

Type declarations of the form

\texttt{type } \tau

Example: colors

\texttt{type color}
\texttt{function blue: color}
\texttt{function red: color}
\texttt{axiom distinct: red \neq blue}
Axiomatic Type Definitions

Type declarations of the form

\[
\text{type } \tau
\]

Example: colors


| type color |
| function blue: color |
| function red: color |
| axiom distinct: red \(\neq\) blue |

Polymorphic types:

\[
\text{type } \tau \alpha_1 \cdots \alpha_k
\]

where \(\alpha_1 \cdots \alpha_k\) are type parameters.
Example: Sets

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)
Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates
Arrays as References on Pure Maps

Axiomatization of maps from int to some type $\alpha$:

```
type map $\alpha$
function select (map $\alpha$, int): $\alpha$
function store (map $\alpha$, int, $\alpha$): map $\alpha$
axiom select_store_eq:
  forall a:map $\alpha$, i:int, v:$\alpha$.
  select(store(a,i,v),i) = v
axiom select_store_neq:
  forall a:map $\alpha$, i j:int, v:$\alpha$.
  i $\neq$ j $\rightarrow$ select(store(a,i,v),j) = select(a,j)
```

- Unbounded indexes.
- $\text{select}(a,i)$ models the usual notation $a[i]$.
- $\text{store}$ 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
  \[ a[i] := e \]
  is interpreted as
  \[ a := \text{store}(a,i,e) \]
**Simple Example**

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

procedure test()
  writes a
  ensures select(a,0) = 13

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

Exercise: prove this program.
```
Example: Swap

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

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

procedure 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 *)
```
Exercises on Arrays

- Prove Swap using WP.
- Prove the procedure

```plaintext
procedure 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 procedure that increments by 1 all cells, between given indexes $i$ and $j$, of an array of reals.
Exercise: Search Algorithms

```haskell
val a: ref (map real)

function search (n:int, v:real): int
  requires 0 ≤ n
  ensures ?
body ?
```

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:
   for each \( i \) from 0 to \( n - 1 \): if \( a[i] = v \) then return \( i \)
Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates
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
  ```
A possible approach to formalizing bounded arrays is

```
type array α = { length:int; contents:map α }
```

Drawback of this approach: needs to specify that length does not change all along computations
Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates
Sum Types

- Sum types à la ML:

  
  type t =
  | \( C_1 \tau_{1,1} \ldots \tau_{1,n_1} \)
  | \[ \ldots \]
  | \( C_k \tau_{k,1} \ldots \tau_{k,n_k} \)
Sum Types

- Sum types à la ML:
  ```plaintext
type t =
  | C₁ τ₁₁ ⋯ τ₁ₙ₁
  | ⋮
  | Cₖ τₖ₁ ⋯ τₖₙₖ
  ```

- Pattern-matching with
  ```plaintext
match e with
  | C₁(p₁, ⋯, pₙ₁) → e₁
  | ⋮
  | Cₖ(p₁, ⋯, pₙₖ) → eₖ
end
  ```
Sum Types

- Sum types à la ML:

```ocaml
type t =
  | C_1 \tau_{1,1} \cdots \tau_{1,n_1}
  | \cdots
  | C_k \tau_{k,1} \cdots \tau_{k,n_k}
```

- Pattern-matching with

```ocaml
match e with
  | C_1(p_1, \cdots, p_{n_1}) \rightarrow e_1
  | \cdots
  | C_k(p_1, \cdots, p_{n_k}) \rightarrow e_k
end
```

- Extended pattern-matching.
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(x,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)

procedure rev_append(r:list int)
  variant
  writes l
  ensures

body
  match r with
  | Nil → skip
  | Cons(x,r) → l := Cons(x,l); rev_append(r)
end

procedure 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.

Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates
Inductive Predicates

- Definition à la Prolog, also in Coq, PVS, etc.
- An **inductive definition** of a predicate has the form
  \[
  \text{inductive } p(\tau_1, \ldots, \tau_n):
  \]
  \[
  | id_1 : \text{clause}_1
  \]
  \[
  \ldots
  \]
  \[
  | id_k : \text{clause}_k
  \]
  where clauses have the form
  \[
  \forall \vec{x}. \ hyp \Rightarrow p(e_1, \ldots, e_n)
  \]
  and \( p \) occurs only positively in \( hyp \) (Horn clause).
- Always one smallest fix-point:
  predicate satisfying the clauses that is true the less often.
Inductive Predicates: Example

Classical example: transitive closure.

**predicate** \( r(x:t,y:t) = \ldots \)

**inductive** \( r\_star(t,t) = \)

| empty: | forall \( x:t. \ r\_star(x,x) \) |
| single: | forall \( x \ y:t. \ r(x,y) \rightarrow r\_star(x,y) \) |
| trans: | forall \( x \ y \ z:t. \) |
|        | \( r\_star(x,y) \land r\_star(y,z) \rightarrow r\_star(x,z) \) |
Exercise: Selection Sort

```
val a: ref(map real)

procedure sort(n:int):
  requires 0 ≤ n
  writes a
  ensures 

body ?
```

1. Formalize postconditions:
   - array in increasing order between 0 and $n - 1$,
   - array at exit is a permutation of the array at entrance.

2. Implement and prove selection sort algorithm:
   for each $i$ from 0 to $n - 1$:
     find index $idx$ of the min element between $i$ and $n - 1$
     swap elements at indexes $i$ and $idx$
Extra Exercises

- **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 \( m \)
Extra Exercises

▶ Binary Search:

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

▶ Insertion Sort:

for each \( i \) from 1 to \( n - 1 \):
    insert element at index \( i \) at the right place
    between indexes 0 and \( i - 1 \)