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,

Reminder 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

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$

Predicate defined as

predicate $p(x_1 : \tau_1, \ldots, x_n : \tau_n) = e$

where τ_i, τ are not reference types.

- No recursion allowed
- No side effects
- Define total functions and predicates

Logic Symbols: Examples

```
function sqr(x:int) = x * x

predicate prime(x:int) =
  x ≥ 2 ∧
  forall y z:int. y ≥ 0 ∧ z ≥ 0 ∧ x = y*z →
    y=1 ∨ z=1
```

Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product Types

Sum Types

Inductive Predicates

Axiomatic Definitions

Function and predicate declarations of the form

function $f(\tau, \ldots, \tau_n) : \tau$
predicate $p(\tau, \ldots, \tau_n)$

together with axioms

axiom id : $formula$

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

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

Example: factorial

```
function fact(int):int
axiom fact0: fact(0) = 1
axiom factn: forall n:int. n ≥ 1 →
  fact(n) = n * fact(n-1)
```

Axiomatic Definitions

- Functions/predicates are typically underspecified.
  $\Rightarrow$ model partial functions in a logic of total functions.

Axiomatic Definitions

- Functions/predicates are typically underspecified.
  $\Rightarrow$ model partial functions in a logic of total functions.
- About soundness: axioms may introduce inconsistencies.

Axiomatic Definitions: Example of Factorial

Exercise: Find appropriate precondition, postcondition, loop invariant, and variant, for this program:

```
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

type τ

Example: colors

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

Axiomatic Type Definitions

Type declarations of the form

type τ

Example: colors

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

Polymorphic types:

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 α:

```
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

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

  is interpreted as

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

Simple Example

```
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*:

```
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

```
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

```
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

Product Types: Example

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 =
  | $C_1\ \tau_{1,1} \cdots \tau_{1,n_1}$
  | $\cdots$
  | $C_k \tau_{k,1} \cdots \tau_{k,n_k}$

Sum Types

- Sum types à la ML:

  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

  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

Sum Types

- Sum types à la ML:

  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

  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.

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

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

  inductive $p(\tau_1, \ldots, \tau_n)$:
  | id_1: $clause_1$
  ...
  | id_k: $clause_k$

  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) = ...

inductive r_star(t,t) =
  | empty: forall x:t. r_star(x,x)
  | single: forall x y:t. r(x,y) → r_star(x,y)
  | trans: forall x y z:t.
    r_star(x,y) ∧ r_star(y,z) → 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:

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

- Insertion Sort:

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