Advanced Modeling, Specification Languages, Case of Array Programs

Claude Marché

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

16 janvier 2012

Reminder of the last lecture

- Typed logic language
- Built-in types: boolean and reals
- *Immutable* local variables in expressions and formulas
- *Immutable* variables in statements
- *Mutable* variables in statements
- Procedures and procedure calls
  - Parameters are *immutable*
  - *Contract*
  - Modular proof: proof of a call assumes the contract
  - *Old* label in postcondition
  - Recursive procedures, termination with a *variant*

Exercice: 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

```
val res : ref int

procedure f91(n:int) :
  requires ?
  variant ?
  writes ?
  ensures ?
body:
  if n ≤ 100 then f91(n + 11); f91(!res)
  else res := n − 10
```

Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product types

Sum Types

Inductive predicates

About 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: compromise between

- expressivity 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

Defined logic symbols

Functions defined under the form

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

Predicate defined under the form

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

where τ_i, τ are not reference types.

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

Defined logic symbols: examples

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

predicate prime(x:int) =
$x \geq 2 \wedge$
forall y z:int. $y \geq 0 \wedge z \geq 0 \wedge x = y*z \rightarrow y=1 \ \vee \ z=1$

Outline

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays

Product types

Sum Types

Inductive predicates

Axiomatic definitions

Declarations of the form

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

together with *axioms*

axiom *id* : *formula*

Specifies that *f*, resp. *p*, is *any symbol satisfying the axioms*

Example: division

function div(real,real):real
axiom mul_div: **forall** x,y. y$\neq$0 $\rightarrow$ div(x,y)$*$y = x

Axiomatic Definitions

Example: factorial

function fact(int):int
axiom fact0: fact(0) = 1
axiom factn: **forall** n:int. $n \geq 1 \rightarrow fact(n) = n * fact(n-1)$

- Functions/predicates are typically *underspecified*
- Allows to model a partial function in a logic of total functions
- About soundness:

  caution: axioms may introduce inconsistencies

Axiomatic definitions: example of factorial

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

```
val res : ref int

procedure fact_imp (x:int) :
  requires ?
  writes ?
  ensures ?
body:
  let y = ref 0 in
  res := 1;
  while y < x do y := !y + 1; res := !res * !y done
```

See file `fact.mlw`

Axiomatic type definitions

Declarations of the form

type τ

Example: colors

type color
function blue : color
function red : color
axiom distinct: red $\neq$ blue

Polymorphic types: declarations of the form

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. $\neg$ mem(x,empty)
axiom mem_single: **forall** x y:α. mem(x,single(y)) $\leftrightarrow$ x=y
axiom mem_union:
  forall x:α, y z:set α.
    mem(x,union(y,z)) $\leftrightarrow$ mem(x,y) or 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 $\neq$ j $\rightarrow$ select(store(a,i,v),j) = select(a,j)

- Indexes unbounded
- 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 of the form

  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

Permuts 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 $\wedge$ select(a,1) = 42 $\wedge$ select(a,2) = 64
ensures
select(a,0) = 64 $\wedge$ select(a,1) = 42 $\wedge$ select(a,2) = 13
body swap(0,2)

See file `swap.mlw`

- Specify, implement and prove a procedure which increments by 1 all cells, between given indexes i and j, of an array of reals

See file `incr_array.mlw`

Exercise: Search Algorithms

```
val a: ref(map real)
val idx : ref int

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

1. Formalize postcondition: if v occurs in a, between 0 and $n-1$ then idx is an index where v occurs, otherwise idx set to -1
2. Implement and prove *linear search*:

   for each i from 0 to $n-1$: if $a[i] = v$ then $idx := i$; *exit*

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 formalize *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 =
  $| \; C_1 \tau_{1,1} \cdots \tau_{1,n_1}$
  $| \; \cdots$
  $| \; C_k \tau_{k,1} \cdots \tau_{k,n_k}$
- Pattern-matching with `match ... with ... end`

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 ? 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)

procedure rev(r:list int)
  writes l
  ensures l = rev r
  body ?
```

See file `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

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 clause has 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) $\rightarrow$ r_star(x,y)
  | trans: **forall** x y z:t. r_star(x,y) $\wedge$ 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 $n - 1$:

   finds 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* $<=$ *high*: let *m* be the middle of *low* and *high*
  if $a[m] = v$ then $idx := m$; *exit*
  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 $i - 1$