Blocking Semantics, Weakest Preconditions

First Extensions
(ghost variables, labels, function calls, arrays)

Claude Marché

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

20 décembre 2013

Exercise 1

Consider the following (inefficient) program for computing the sum $a + b$.

```
x := a; y := b;
while y > 0 do
   x := x + 1; y := y - 1
```

- Propose a post-condition stating that the final value of x is the sum of the values of a and b
- Find an appropriate loop invariant
- Prove the program.

Exercise 2

The following program is one of the original examples of Floyd.

```
q := 0; r := x;
while r ≥ y do
   r := r - y; q := q + 1
```

- Propose a formal precondition to express that x is assumed non-negative, y is assumed positive, and a formal post-condition expressing that q and r are respectively the quotient and the remainder of the Euclidean division of x by y.
- Find appropriate loop invariant and prove the correctness of the program.

Exercise 3

Let's assume given in the underlying logic the functions div2(x) and mod2(x) which respectively return the division of x by 2 and its remainder. The following program is supposed to compute, in variable r, the power x^n.

```
r := 1; p := x; e := n;
while e > 0 do
  if mod2(e) ≠ 0 then r := r * p;
  p := p * p;
  e := div2(e);
```

- Assuming that the power function exists in the logic, specify appropriate pre- and post-conditions for this program.
- Find an appropriate loop invariant, and prove the program.

Reminder of the last lecture

- Classical Hoare Logic
  - Very simple programming language
  - Deduction rules for triples $\{Pre\}s\{Post\}$
  - WLP: if $Pre \Rightarrow \mathrm{WLP}(s, Post)$ then $\{Pre\}s\{Post\}$ valid
  - Use of Why3
- Modern programming language, ML-like
  - more data types: int, bool, real, unit
  - *logic variables*: local and immutable
  - statement = expression of type unit
  - Typing rules
  - Formal operational semantics (small steps)
  - *type soundness*: every typed program executes without blocking.

This Lecture's Goals

- Blocking semantics continued:
  - Safety defined by *Blocking Semantics*
  - New WP calculus
  - Soundness = safety of execution
- Extend the language:
  - Ghost variables and Labels
  - Local mutable variables
  - Sub-programs, *modular reasoning*
- Towards complex data structures
  - Axiomatized types and predicates
  - *Arrays*

Outline

Reminder: Syntax

e	$::=$	t	(pure term)
	$\mid$	$e \; op \; e$	(binary operation)
	$\mid$	$x := e$	(assignment)
	$\mid$	$\texttt{let}\ v = e\ \texttt{in}\ e$	(local binding)
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	(conditional)
	$\mid$	$\texttt{assert}\ p$	(*assertion*)
	$\mid$	$\texttt{while}\ e\ \texttt{invariant}\ I\ \texttt{do}\ e$	(*annotated* loop)

- sequence $e_1 ; e_2$: syntactic sugar for

$$\texttt{let}\ v = e_1\ \texttt{in}\ e_2$$

  when e_1 has type `unit` and v not used in e_2
- Addition in the logic language: keyword **`result`** in post-conditions, denotes the value of the expression executed

Reminder: Operational Semantics

- one-step execution has the form

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

- values (i.e. constants) do not reduce
- failed assertions = “run-time errors”

Novelties

- Need for *context rules*
- Precise the order of evaluation: left to right

Blocking Semantics: Modified Rules

$$\frac{[\![P]\!]_{\Sigma,\Pi} \text{ holds}}{\Sigma, \Pi, \texttt{assert}\ P \rightsquigarrow \Sigma, \Pi, ()}$$

$$\frac{[\![I]\!]_{\Sigma,\Pi} \text{ holds}}{\begin{array}{l}\Sigma, \Pi, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e \rightsquigarrow \\ \quad \Sigma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e)\ \texttt{else}\ ()\end{array}}$$

Important

Execution blocks as soon as an invalid annotation is met

Soundness of a program

Definition

Execution of an expression in a given state is *safe* if it does not block: either terminates on a value or runs infinitely.

Definition

A triple $\{P\}e\{Q\}$ is valid if for any state Σ, Π satisfying P, e *executes safely* in Σ, Π, and if it terminates, the final state satisfies Q

Outline

Weakest Preconditions Revisited

Goal:

- construct a new calculus $\mathrm{WP}(e, Q)$
- expected property: in any state satisfying $\mathrm{WP}(e, Q)$, e is guaranteed to execute safely

Remark:

- Stating this for $Q = \mathit{true}$ is enough to ensure safety
- But need to state this for any Q to prove soundness (by induction)

New Weakest Precondition Calculus

- Pure terms:

$$WP(t, Q) = Q[result \leftarrow t]$$

- Let binding:

$$\begin{aligned}&\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, Q) = \\ &\quad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow result])\end{aligned}$$

Weakest Preconditions, continued

- Assignment:

$$\mathrm{WP}(x := e, Q) = \mathrm{WP}(e, Q[\mathit{result} \leftarrow (); x \leftarrow \mathit{result}])$$

- Alternative:

$$\begin{array}{rcl} \mathrm{WP}(x := e, Q) & = & \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ x := v, Q) \\ \mathrm{WP}(x := t, Q) & = & Q[\mathit{result} \leftarrow (); x \leftarrow t]) \end{array}$$

WP: Exercise

$$\mathrm{WP}(\texttt{let}\ v = x\ \texttt{in}\ (x := x + 1; v), x > \mathit{result}) = ?$$

Weakest Preconditions, continued

- Conditional

$$\mathrm{WP}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, Q) = \\ \mathrm{WP}(e_1, \texttt{if } \mathit{result} \texttt{ then } \mathrm{WP}(e_2, Q) \texttt{ else } \mathrm{WP}(e_3, Q))$$

- Alternative with let: (exercise!)

Weakest Preconditions, continued

- Assertion

$$\begin{aligned} \mathrm{WP}(\texttt{assert}\ P, Q) &= P \wedge Q \\ &= P \wedge (P \Rightarrow Q) \end{aligned}$$

(second version useful in practice)

- While loop

$$\begin{aligned} &\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ e, Q) = \\ &\quad I \wedge \\ &\quad \forall \vec{v}, (I \Rightarrow \mathrm{WP}(c, \texttt{if}\ \mathit{result}\ \texttt{then}\ \mathrm{WP}(e, I)\ \texttt{else}\ Q))[w_i \leftarrow v_i] \end{aligned}$$

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

Soundness of WP

Lemma (Preservation by Reduction)

If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *and* $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$ *then* $\Sigma', \Pi' \models \mathrm{WP}(e', Q)$

Proof: predicate induction of $\rightsquigarrow$.

Lemma (Progress)

If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *and* e *is not a value then there exists* Σ', Π, e' *such that* $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$

Proof: structural induction of e.

Corollary (Soundness)

If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ *then* e *executes safely in* Σ, Π.

Outline

Ghost variables

Ghost variables: additional variables, introduced for the specification

Example: Euclid's algorithm, on two globals `x,y`

```
Euclid:
    requires  ?
    ensures   ?
  = while y > 0 do
      let r = mod x y in let q = div x y in
      x := y; y := r
    done;
    x
```

See Why3 file `euclid.mlw`

Labels: motivation

- Using ghost variables becomes quickly painful
- *Label*
  - simple alternative to ghost variables
  - (but not always possible)

Labels: Syntax and Typing

Add in syntax of *terms*:

$$t \quad ::= \quad x@L \quad \text{(labeled variable access)}$$

Add in syntax of *expressions*:

$$e \quad ::= \quad L : e \quad \text{(labeled expressions)}$$

Typing:

- only mutable variables can be accessed through a label
- labels must be declared before use

Implicit labels:

- Here, available in every formula
- Old, available in post-conditions

Toy Examples, Continued

```
{ true }
let v = r in (r := v + 42; v)
{ r = r@Old + 42 ∧ result = r@Old }
```

```
{ true }
let tmp = x in x := y; y := tmp
{ x = y@Old ∧ y = x@Old }
```

SUM revisited:

```
{ y ≥ 0 }
L:
while y > 0 do
  invariant { x + y = x@L + y@L }
  x := x + 1; y := y - 1
{ x = x@Old + y@Old ∧ y = 0 }
```

Labels: Operational Semantics

Program state

- becomes a collection of maps indexed by labels
- value of variable x at label L is denoted $\Sigma(x, L)$

New semantics of variables in terms:

$$\begin{aligned} [\![x]\!]_{\Sigma,\Pi} &= \Sigma(x, Here) \\ [\![x@L]\!]_{\Sigma,\Pi} &= \Sigma(x, L) \end{aligned}$$

The operational semantics of expressions is modified as follows

$$\begin{aligned} \Sigma, \Pi, x := val &\rightsquigarrow \Sigma\{(x, Here) \leftarrow val\}, \Pi, () \\ \Sigma, \Pi, L : e &\rightsquigarrow \Sigma\{(x, L) \leftarrow \Sigma(x, Here) \mid x \text{ any variable}\}, \Pi, e \end{aligned}$$

Syntactic sugar: term $t@L$

- attach label L to any variable of t that does not have an explicit label yet.
- example:$(x + y@K + 2)@L + x$ is $x@L + y@K + 2 + x@Here$.

New rules for WP

New rules for computing WP:

$$\begin{aligned} \mathrm{WP}(x := t, Q) &= Q[x@Here \leftarrow t] \\ \mathrm{WP}(L : e, Q) &= \mathrm{WP}(e, Q)[x@L \leftarrow x@Here \mid x \text{ any variable}] \end{aligned}$$

Exercise:

$$\mathrm{WP}(L : x := x + 42, x@Here > x@L) = ?$$

Exercice: Euclid's algorithm revisited

```
Euclid:
    requires { x ≥ 0 ∧ y ≥ 0 }
    ensures  { result = gcd(x@Old,y@Old) }
  = L:
    while y > 0 do
      invariant { x ≥ 0 ∧ y ≥ 0 }
      invariant { gcd(x,y) = gcd(x@L,y@L) }
      let r = mod x y in let q = div x y in
      x := y; y := r
    done;
    x
```

See file `euclid_lab.mlw`

Outline

Mutable Local Variables

We extend the syntax of expressions with

$$e ::= \texttt{let ref}\ id = e\ \texttt{in}\ e$$

Example: isqrt revisited

```
val x, res : ref int

isqrt:
  res := 0;
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done
```

Operational Semantics

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Π no longer contains just immutable variables.

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let ref } x = e_1 \texttt{ in } e_2 \rightsquigarrow \texttt{let ref } x = e_1' \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let ref } x = v \texttt{ in } e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

Operational Semantics

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

Π no longer contains just immutable variables.

$$\frac{\Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e_1'}{\Sigma, \Pi, \texttt{let ref } x = e_1 \texttt{ in } e_2 \rightsquigarrow \texttt{let ref } x = e_1' \texttt{ in } e_2}$$

$$\frac{}{\Sigma, \Pi, \texttt{let ref } x = v \texttt{ in } e \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

$$\frac{x \text{ local variable}}{\Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, \mathit{Here}) \mapsto v\}, e}$$

And labels too.

Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

$$\mathrm{WP}(\texttt{let ref}\ x = e_1\ \texttt{in}\ e_2, Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \leftarrow \mathrm{result}])$$

$$\mathrm{WP}(x := e, Q) = \mathrm{WP}(e, Q[x \leftarrow \mathit{result}])$$

$$\mathrm{WP}(L : e, Q) = \mathrm{WP}(e, Q)[x@L \leftarrow x@\mathit{Here} \mid x \text{ any variable}]$$

Exercise

- Extend the post-condition of Euclid algorithm to express the Bezout property:

$$\exists a, b, result = x * a + y * b$$

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`

Outline

Functions

Program structure:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{vardecl} \mid \mathit{fundecl} \\
\mathit{vardecl} & ::= & \text{val } \mathit{id} : \text{ref } \mathit{basetype}
\end{array}
$$

Functions

Program structure:

$$
\begin{array}{rcl}
\textit{prog} & ::= & \textit{decl}^* \\
\textit{decl} & ::= & \textit{vardecl} \mid \textit{fundecl} \\
\textit{vardecl} & ::= & \text{val } \textit{id} : \text{ref } \textit{basetype} \\
\textit{fundecl} & ::= & \text{let fun } \textit{id}(\ (\textit{param},)^*\ ) : \textit{basetype} \\
 & & \qquad \textit{contract} \text{ body } e \\
\textit{param} & ::= & \textit{id} : \textit{basetype} \\
\textit{contract} & ::= & \text{requires } t \text{ writes } (\textit{id},)^* \text{ ensures } t
\end{array}
$$

Functions

Program structure:

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{decl}^* \\
\mathit{decl} & ::= & \mathit{vardecl} \mid \mathit{fundecl} \\
\mathit{vardecl} & ::= & \text{val } \mathit{id} : \text{ref } \mathit{basetype} \\
\mathit{fundecl} & ::= & \text{let fun } \mathit{id}(\ (\mathit{param},)^*\ ) : \mathit{basetype} \\
 & & \qquad \mathit{contract} \text{ body } e \\
\mathit{param} & ::= & \mathit{id} : \mathit{basetype} \\
\mathit{contract} & ::= & \text{requires } t \text{ writes } (\mathit{id},)^* \text{ ensures } t
\end{array}
$$

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label *Old* available),
  - assigned variables: clause writes .
- Body: expression.

Example: isqrt

```
let fun isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
```

Example using *Old* label

```
val res: ref int

let fun incr(x:int)
  requires true
  writes res
  ensures res = res@Old + x
body
  res := res + x
```

Typing

Definition d of function f:

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

Typing

Definition d of function f:

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

Well-formed definitions:

$$\frac{\begin{array}{ll} \Gamma' = \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma & \vec{w} \subseteq \Gamma \\ \Gamma' \vdash Pre, Post : formula & \Gamma' \vdash Body : \tau \\ \vec{w}_g \subseteq \vec{w} \text{ for each call } g & y \in \vec{w} \text{ for each assign } y \end{array}}{\Gamma \vdash d : wf}$$

where Γ contains the global declarations.

Typing

Definition d of function f:

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

Well-typed function calls:

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

Note: t_i are immutable expressions.

Operational Semantics

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

$$\frac{\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\} \qquad \Sigma, \Pi' \models Pre}{\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (Old : \texttt{frame}(\Pi', Body, Post))}$$

Operational Semantics of Function Call

`frame` is a dummy operation that keeps track of the local variables of the callee:

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi'', (\texttt{frame}(\Pi, e, P)) \rightsquigarrow \Sigma', \Pi'', (\texttt{frame}(\Pi', e', P))}$$

It also checks that the post-condition holds at the end:

$$\frac{\Sigma, \Pi' \models P[\text{result} \leftarrow v]}{\Sigma, \Pi, (\texttt{frame}(\Pi', v, P)) \rightsquigarrow \Sigma, \Pi, v}$$

WP Rule of Function Call

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

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

Modular proof

When calling function f, only the contract of f is visible, not its body

Example: isqrt(42)

Exercise: prove that $\{true\}isqrt(42)\{\text{result} = 6\}$ holds.

```
val isqrt(x:int): int
  requires x ≥ 0
  writes (nothing)
  ensures result ≥ 0 ∧
          sqr(result) ≤ x < sqr(result + 1)
```

Abstraction of sub-programs

- Keyword **val** introduces a function with a contract but without body
- *writes* clause is mandatory in that case

Example: Incrementation

```
val res: ref int

val incr(x:int):unit
  writes res
  ensures res = res@Old + x
```

Exercise: Prove that $\{res = 6\} incr(36) \{res = 42\}$ holds.

Outline

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
  - logic (i.e. pure) functions, predicates
  - structured types, pattern-matching
- Axiomatizations
- Inductive predicates

Logic Symbols

Logic 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
- Defines 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

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 for this program:

```
let fun 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
```

Modeling Run-time Errors

Error “Division by zero” can be modeled by an abstract function

```
val div_real(x:real,y:real):real
   requires y ≠ 0.0
   ensures  result = div(x,y)
```

Reminder

Execution blocks when an invalid annotations is met

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

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)

let fun test()
  writes a
  ensures select(a,0) = 13  (* 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: array int

let fun swap(i:int,j:int)
  requires 0 ≤ i < length a ∧ 0 ≤ i < length a
  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 program

```
let fun 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 program that increments by 1 all cells, between given indexes *i* and *j*, of an array of reals.

Arrays as Reference on pairs (length,map)

- Goal: model "out-of-bounds" run-time errors
- Array variable: reference on a pair `(length,map` α`)`.
- `a[i]` interpreted as `get(a,i)`
- `a[i] := v` interpreted as `set(a,i,v)`

```
val get(a:array α,i:int):α
  requires 0 ≤ i < fst(a)
  ensures  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) ∧
           snd(a) = store(snd(a@Old),i,v)
```

In Why3: `use import array.Array`
syntax: `(length a)`, `a[i]`, `a[i]<-v`

Example: Swap in why3

```
val a: array int

let swap(i:int,j:int)
  requires { 0 ≤ i < length a ∧ 0 ≤ i < length a }
  writes   { a }
  ensures  { a[i] = old a[j] ∧ a[j] = old a[i]}
  ensures  { forall k:int.
               0 ≤ k < length a ∧ k ≠ i ∧ k ≠ j →
               a[k] = old a[k] }
=
  let tmp = a[i] in a[i] <- a[j]; a[j] <- tmp
```

Exercise: Search Algorithms

```
val a: array real

let 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

Exercise: 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