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 $a + b$.

```plaintext
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.

```plaintext
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 $Pre \Rightarrow WLP(s, Post)$ 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

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
Reminder: Syntax

\[
e ::= t \quad \text{(pure term)}
\]

\[
| e \text{ op } e \quad \text{(binary operation)}
\]

\[
| x := e \quad \text{(assignment)}
\]

\[
| \text{let } v = e \text{ in } e \quad \text{(local binding)}
\]

\[
| \text{if } e \text{ then } e \text{ else } e \quad \text{(conditional)}
\]

\[
| \text{assert } p \quad \text{(assertion)}
\]

\[
| \text{while } e \text{ invariant } l \text{ do } e \quad \text{(annotated loop)}
\]

- sequence \(e_1; e_2\): syntactic sugar for
  \[
  \text{let } v = e_1 \text{ in } e_2
  \]
  when \(e_1\) has type \text{unit} and \(v\) not used in \(e_2\)

- Addition in the logic language: keyword \text{result} in post-conditions, denotes the value of the expression executed
Reminder: Operational Semantics

- one-step execution has the form
  \[ \Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \]

- 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

\[ [P]_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{assert } P \overset{\sim}{\Rightarrow} \Sigma, \Pi, () \]

\[ [I]_{\Sigma, \Pi} \text{ holds} \]
\[ \Sigma, \Pi, \text{while } c \text{ invariant } l \text{ do } e \overset{\sim}{\Rightarrow} \]
\[ \Sigma, \Pi, \text{if } c \text{ then } (e; \text{while } c \text{ invariant } l \text{ do } e) \text{ else } () \]

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 $\Sigma, \Pi$ satisfying $P$, $e$ executes safely in $\Sigma, \Pi$, and if it terminates, the final state satisfies $Q$. 
Outline

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
Weakest Preconditions Revisited

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

Remark:
- Stating this for $Q = true$ is enough to ensure safety
- But need to state this for any $Q$ to prove soundness (by induction)
New Weakest Precondition Calculus

- Pure terms:
  \[ WP(t, Q) = Q[\text{result} \leftarrow t] \]

- Let binding:
  \[ WP(\text{let } x = e_1 \text{ in } e_2, Q) = WP(e_1, WP(e_2, Q)[x \leftarrow \text{result}]) \]
Weakest Preconditions, continued

▶ Assignment:

\[
\text{WP}(x := e, Q) = \text{WP}(e, Q[result ← (); x ← result])
\]

▶ Alternative:

\[
\begin{align*}
\text{WP}(x := e, Q) &= \text{WP}\left(\text{let } v := e \text{ in } x := v, Q\right) \\
\text{WP}(x := t, Q) &= Q[result ← (); x ← t]]
\end{align*}
\]
WP: Exercise

\[
\text{WP}(\text{let } v = x \text{ in } (x := x + 1; v), x > \text{ result}) = ?
\]
Weakest Preconditions, continued

- Conditional

\[
WP(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, Q) = WP(e_1, \text{if } result \text{ then } WP(e_2, Q) \text{ else } WP(e_3, Q))
\]

- Alternative with let: (exercise!)
Weakest Preconditions, continued

- **Assertion**
  \[
  WP(\text{assert } P, Q) = P \land Q \\
  = P \land (P \Rightarrow Q)
  \]
  (second version useful in practice)

- **While loop**
  \[
  WP(\text{while } c \text{ invariant } I \text{ do } e, Q) = \\
  I \land \\
  \forall \vec{v}, (I \Rightarrow WP(c, \text{if result then } WP(e, l) \text{ else } Q))[w_i \leftarrow v_i]
  \]

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 \( \Sigma, \Pi \models WP(e, Q) \) and \( \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \) then \( \Sigma', \Pi' \models WP(e', Q) \)

Proof: predicate induction of \( \rightsquigarrow \).

Lemma (Progress)

If \( \Sigma, \Pi \models WP(e, Q) \) and \( e \) is not a value then there exists \( \Sigma', \Pi, e' \) such that \( \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \)

Proof: structural induction of \( e \).

Corollary (Soundness)

If \( \Sigma, \Pi \models WP(e, Q) \) then \( e \) executes safely in \( \Sigma, \Pi \).
Outline

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
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 $r = \text{mod } x \ y$ in let $q = \text{div } x \ y$ 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 \textit{terms}:  
\[
t ::= x@L \quad \text{(labeled variable access)}
\]

Add in syntax of \textit{expressions}:  
\[
e ::= L : e \quad \text{(labeled expressions)}
\]

Typing:
\begin{itemize}
  \item only mutable variables can be accessed through a label
  \item labels must be declared before use
\end{itemize}

Implicit labels:
\begin{itemize}
  \item Here, available in every formula
  \item Old, available in post-conditions
\end{itemize}
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:

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

The operational semantics of expressions is modified as follows

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

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 $x@L + y@K + 2 + x@\text{Here}$. 
New rules for WP

New rules for computing WP:

\[
\begin{align*}
\text{WP}(x := t, Q) & = Q[x@Here \leftarrow t] \\
\text{WP}(L : e, Q) & = \text{WP}(e, Q)[x@L \leftarrow x@Here | x \text{ any variable}]
\end{align*}
\]

Exercise:

\[
\text{WP}(L : x := x + 42, x@Here > x@L) = ?
\]
Euclid:  

requires \{ x \geq 0 \land y \geq 0 \}  

ensures \{ \text{result} = \gcd(x@\text{Old},y@\text{Old}) \}  

= L:  

while y > 0 do  
    invariant \{ x \geq 0 \land y \geq 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

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
Mutable Local Variables

We extend the syntax of expressions with

\[ e ::= \text{let ref } id = e \text{ in } e \]

Example: isqrt revisited

```ocaml
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

\[
\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'
\]

\(\Pi\) no longer contains just immutable variables.

\[
\Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e'_1
\]

\[
\Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \leadsto \text{let ref } x = e'_1 \text{ in } e_2
\]

\[
\Sigma, \Pi, \text{let ref } x = v \text{ in } e \leadsto \Sigma, \Pi\{ (x, \text{Here}) \mapsto v \}, e
\]
Operational Semantics

\[ \Sigma, \Pi, e \leadsto \Sigma', \Pi', e' \]

\( \Pi \) no longer contains just immutable variables.

\[ \Sigma, \Pi, e_1 \leadsto \Sigma', \Pi', e'_1 \]
\[ \Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \leadsto \text{let ref } x = e'_1 \text{ in } e_2 \]

\[ \Sigma, \Pi, \text{let ref } x = v \text{ in } e \leadsto \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

\( x \) local variable
\[ \Sigma, \Pi, x := v \leadsto \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

And labels too.
Mutable Local Variables: WP rules

Rules are exactly the same as for global variables

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

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

\[
\text{WP}(L : e, Q) = \text{WP}(e, Q)[x@L \leftarrow x@\text{Here} | x \text{ any variable}]
\]
Exercise

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

  \[ \exists a, b, \text{result} = x \cdot a + y \cdot b \]

- Prove the program by adding appropriate ghost local variables

Use canvas file `exo_bezout.mlw`
Outline

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
Functions

Program structure:

\[
\begin{align*}
\text{prog} & ::= \text{decl}^* \\
\text{decl} & ::= \text{vardecl} | \text{fundecl} \\
\text{vardecl} & ::= \text{val id : ref basetype}
\end{align*}
\]
Program structure:

\[
\begin{align*}
prog & ::= \ decl* \\
\decl & ::= \ vardecl \mid \ fundecl \\
\vardecl & ::= \ val \ id : \ ref \\ basetype \\
\fundecl & ::= \ let \ fun \ id( (\ param,)* ) : \ basetype \\
& \quad contract \ body \ e \\
\param & ::= \ id : \ basetype \\
\contract & ::= \ requires \ t \ writes \ (id,)* \ ensures \ t
\end{align*}
\]
Functions

Program structure:

\[
prog ::= decl^* \\
\}
\]
decl ::= vardecl | fundecl
vardecl ::= val id : ref basetype
fundecl ::= let fun id( (param,)* ) : basetype

\[
contract \ \text{body} \ e \\
\}
\]
param ::= id : basetype

\[
contract ::= \text{requires } t \ \text{writes } (id,)^* \ \text{ensures } t \\
\}
\]

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label \textit{Old} available),
  - assigned variables: clause \textit{writes}.
- Body: expression.
Example: isqrt

```ocaml
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

```ml
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 $f$:

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

Well-formed definitions:

\[
\begin{align*}
\Gamma' &= \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma \\
\Gamma' &\vdash Pre, Post : \text{formula} \\
\vec{w}_g &\subseteq \vec{w} \text{ for each call } g \\
y &\in \vec{w} \text{ for each assign } y \\
\Gamma &\vdash d : wf
\end{align*}
\]

where $\Gamma$ 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 $Body$

$$\Pi' = \{ x_i \mapsto \llbracket t_i \rrbracket_{\Sigma, \Pi} \} \quad \Sigma, \Pi' \models Pre$$

$$\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (Old : frame(\Pi', Body, Post))$$
frame is a dummy operation that keeps track of the local variables of the callee:

\[
\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'
\]

\[
\Sigma, \Pi'', (\text{frame}(\Pi, e, P)) \rightsquigarrow \Sigma', \Pi'', (\text{frame}(\Pi', e', P))
\]

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

\[
\Sigma, \Pi' \models P[\text{result} \leftarrow v]
\]

\[
\Sigma, \Pi, (\text{frame}(\Pi', v, P)) \rightsquigarrow \Sigma, \Pi, v
\]
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 \( Body \)

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

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)\{result = 6\} holds.

```plaintext
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

```ml
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

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
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 \( \tau_i, \tau \) 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

Blocking Semantics, continued

Weakest Preconditions Revisited

Ghost variables and Labels

Local Mutable Variables

Functions

Advanced Modeling of Programs

Axiomatic Definitions

Programs on Arrays
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

axiom \( id : \text{formula} \)

specify that \( f \) (resp. \( p \)) is any symbol satisfying the axioms.
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 \neq 0 \rightarrow div(x,y) \times y = x

Example: factorial

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)
Axiomatic Definitions

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

- Functions/predicates are typically \textit{underspecified}. \\
  \implies \textit{model partial} functions in a logic of \textit{total} functions.

- About soundness: axioms may introduce \textit{inconsistencies}. 
Exercise: Find appropriate precondition, postcondition, loop invariant for this program:

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

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

Reminder

Execution blocks when an invalid annotations is met
Axiomatic Type Definitions

Type declarations of the form

\texttt{type } \tau

Example: colors

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

Type declarations of the form

\[ \text{type } \tau \]

Example: colors

\[
\begin{align*}
\text{type} & \quad \text{color} \\
\text{function} & \quad \text{blue: color} \\
\text{function} & \quad \text{red: color} \\
\text{axiom} & \quad \text{distinct: red \neq blue}
\end{align*}
\]

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: ∀ x y z: set α. union(union(x,y),z) = union(x,union(y,z)) |
| axiom union_comm: ∀ x y: set α. union(x,y) = union(y,x) |
| predicate mem(α,set α) |
| axiom mem_empty: ∀ x:α. ¬ mem(x,empty) |
| axiom mem_single: ∀ x y:α. mem(x,single(y)) ↔ x=y |
| axiom mem_union: ∀ x:α, y z: set α. mem(x,union(y,z)) ↔ mem(x,y) ∨ mem(x,z) |
```
Outline

Blocking Semantics, continued
Weakest Preconditions Revisited
Ghost variables and Labels
Local Mutable Variables
Functions
Advanced Modeling of Programs
Axiomatic Definitions

Programs on Arrays
Arrays as References on Pure Maps

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

```plaintext
type map $\alpha$
defunction select(map $\alpha$, int): $\alpha$
defunction 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
  
  \[ a := store(a,i,e) \]
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 $a$:

```ocaml
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

```ml
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 \((\text{length}, \text{map } \alpha)\).
- \(a[i]\) interpreted as \(\text{get}(a, i)\)
- \(a[i] := v\) interpreted as \(\text{set}(a, i, v)\)

```why3
val get(a:array \alpha, i:int):\alpha
  requires 0 \leq i < \text{fst}(a)
  ensures result = \text{select}(\text{snd}(a),i)

val set(a:array \alpha, i:int, v:\alpha):unit
  requires 0 \leq i < \text{fst}(a)
  writes a
  ensures \text{fst}(a) = \text{fst}(a@Old) \land \\
    \text{snd}(a) = \text{store}(\text{snd}(a@Old),i,v)
```

In Why3: `use import array.Array` syntax: \((\text{length } a), a[i], a[i]<-v\)
Example: Swap in why3

```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:
   
   \[
   \text{res} := -1; \\
   \text{for each } i \text{ from } 0 \text{ to } n - 1: \text{ if } a[i] = v \text{ then } \text{res} := i; \\
   \text{return res}
   \]
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