Deductive Program Verification with WHY3

Andrei Paskevich

LRI, Université Paris-Sud — Toccata, Inria Saclay

http://why3.lri.fr/ejcp-2017

ÉJCP 2017

1. A quick look back

Introduction

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explosion — an erroneous *float-to-int* conversion
- 1997: *Pathfinder* reset loop — priority inversion
- 1999: *Mars Climate Orbiter* explosion — unit error

...

Introduction

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explosion — an erroneous *float-to-int* conversion
- 1997: *Pathfinder* reset loop — priority inversion
- 1999: *Mars Climate Orbiter* explosion — unit error

...

- 2006: *Debian SSH bug* — predictable RNG (fixed in 2008)
- 2012: *Heartbleed* — buffer over-read (fixed in 2014)
- 1989: *Shellshock* — insufficient input control (fixed in 2014)

...

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1960.

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1960.

2006: *Nearly All Binary Searches and Mergesorts are Broken*
(Joshua Bloch, Google, a blog post)

The code in JDK:

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

A simple algorithm: Binary search

Goal: find a value in a sorted array.

First algorithm published in 1946.

First correct algorithm published in 1960.

2006: *Nearly All Binary Searches and Mergesorts are Broken*
(Joshua Bloch, Google, a blog post)

The code in JDK:

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

Bug: addition may exceed $2^{31} - 1$, the maximum `int` in Java.

One possible solution:

```
int mid = low + (high - low) / 2;
```

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

```
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
```

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

First theoretical foundation: Floyd-Hoare logic, 1969

Ensure the absence of bugs

Several approaches exist: model checking, abstract interpretation, etc.

In this lecture: deductive verification

1. provide a program with a specification: a mathematical model
2. build a formal proof showing that the code respects the specification

First proof of a program: Alan Turing, 1949

First theoretical foundation: Floyd-Hoare logic, 1969

First grand success in practice: metro line 14, 1998

tool: Atelier B, proof by refinement

Other major success stories

- Flight control software in A380, 2005

  safety proof: the absence of execution errors
  tool: Astrée, abstract interpretation

  proof of functional properties
  tool: Caveat, deductive verification

- Hyper-V — a native hypervisor, 2008

  tools: VCC + automated prover Z3, deductive verification

- CompCert — certified C compiler, 2009

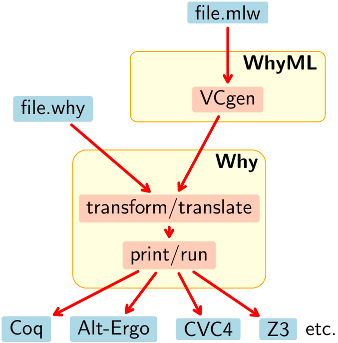
  tool: Coq, generation of the correct-by-construction code

- seL4 — an OS micro-kernel, 2009

  tool: Isabelle/HOL, deductive verification

2. Tool of the day

WHY3 in a nutshell


file.mlw

WhyML

VCgen

file.why

Why

transform/translate

print/run

Coq

Alt-Ergo

CVC4

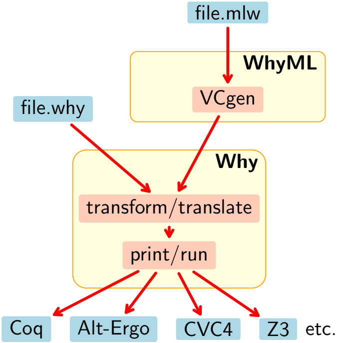
Z3

etc.

WHY3 in a nutshell

WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants



WHY3 in a nutshell

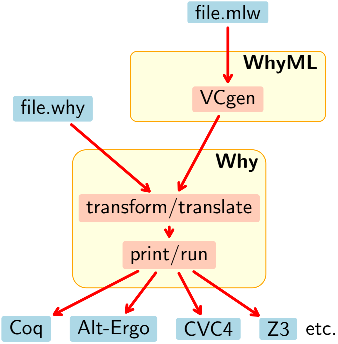
WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WHYML, a specification language

- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)



WHY3 in a nutshell

WHYML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WHY3, a program verification tool

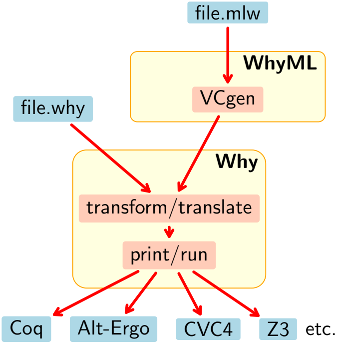
- VC generation using WP or fast WP
- 70+ VC *transformations* ($\approx$ tactics)
- support for 25+ ATP and ITP systems

(Boogie 2011) (ESOP 2013) (VSTTE 2013)

WHYML, a specification language

- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)


file.mlw

WhyML

VCgen

file.why

Why

transform/translate

print/run

Coq

Alt-Ergo

CVC4

Z3

etc.

WHY3 out of a nutshell

Three different ways of using WHY3

- as a logical language
  - a convenient front-end to many theorem provers

- as a programming language to prove algorithms
  - see examples in our gallery
    `http://toccata.lri.fr/gallery/why3.en.html`

- as an intermediate verification language
  - Java programs: Krakatoa (Marché Paulin Urbain)
  - C programs: Frama-C (Marché Moy)
  - Ada programs: SPARK 2014 (Adacore)
  - probabilistic programs: EasyCrypt (Barthe et al.)

Example: maximum subarray problem

```
let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
```

Kadane’s algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |   *)
(*  ......|######## max ########|..............                      *)
(*  ..............................|### cur ####                      *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in



  for i = 0 to length a - 1 do




    cur += a[i];
    if !cur < 0 then       cur := 0;
    if !cur > !max then       max := !cur
  done;
  !max
```

Kadane’s algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                      *)
(*  ..............................|### cur ####                      *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in


  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }


    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then       max := !cur
  done;
  !max
```

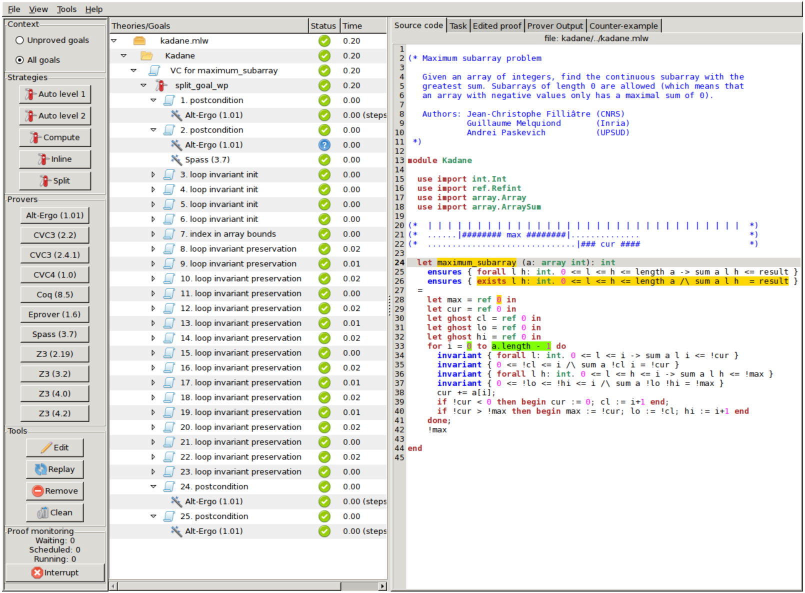
Kadane's algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                      *)
(*  ..............................|### cur ####                      *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Kadane's algorithm

```
use import ref.Refint
use import array.Array
use import array.ArraySum

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Why3 proof session



Waiting: 0
Scheduled: 0
Running: 0

Theories/Goals	Status	Time
kadane.mlw	✓	0.20
Kadane	✓	0.20
VC for maximum_subarray	✓	0.20
split_goal_wp	✓	0.20
1. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps
2. postcondition	✓	0.00
Alt-Ergo (1.01)	?	0.00
Spass (3.7)	✓	0.00
3. loop invariant init	✓	0.00
4. loop invariant init	✓	0.00
5. loop invariant init	✓	0.00
6. loop invariant init	✓	0.00
7. index in array bounds	✓	0.00
8. loop invariant preservation	✓	0.02
9. loop invariant preservation	✓	0.01
10. loop invariant preservation	✓	0.02
11. loop invariant preservation	✓	0.00
12. loop invariant preservation	✓	0.02
13. loop invariant preservation	✓	0.01
14. loop invariant preservation	✓	0.02
15. loop invariant preservation	✓	0.00
16. loop invariant preservation	✓	0.02
17. loop invariant preservation	✓	0.01
18. loop invariant preservation	✓	0.02
19. loop invariant preservation	✓	0.01
20. loop invariant preservation	✓	0.02
21. loop invariant preservation	✓	0.00
22. loop invariant preservation	✓	0.02
23. loop invariant preservation	✓	0.00
24. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps
25. postcondition	✓	0.00
Alt-Ergo (1.01)	✓	0.00 (steps

file: kadane/../kadane.mlw

```

(* Maximum subarray problem

   Given an array of integers, find the continuous subarray with the
   greatest sum. Subarrays of length 0 are allowed (which means that
   an array with negative values only has a maximal sum of 0).

   Authors: Jean-Christophe Filliâtre (CNRS)
            Guillaume Melquiond       (Inria)
            Andrei Paskevich          (UPSUD)
 *)

module Kadane

  use import int.Int
  use import ref.Refint
  use import array.Array
  use import array.ArraySum

(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                     *)
(*  ..............................|### cur ####                     *)

  let maximum_subarray (a: array int): int
    ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
    ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
  =
    let max = ref 0 in
    let cur = ref 0 in
    let ghost cl = ref 0 in
    let ghost lo = ref 0 in
    let ghost hi = ref 0 in
    for i = 0 to a.length - 1 do
      invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
      invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
      invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
      invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
      cur += a[i];
      if !cur < 0 then begin cur := 0; cl := i+1 end;
      if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
    done;
    !max

end

```

3. Program correctness

μML : pure terms

$$
\begin{array}{rcll}
t & ::= & \ldots, -1, 0, 1, \ldots, 42, \ldots & \text{integer constants} \\
 & \mid & \texttt{true} \mid \texttt{false} & \text{Boolean constants} \\
 & \mid & v & \text{immutable variable} \\
 & \mid & x & \text{dereferenced pointer} \\
 & \mid & t \ \mathit{op} \ t & \text{binary operation} \\
 & \mid & \mathit{op} \ t & \text{unary operation} \\
 & & & \\
\mathit{op} & ::= & + \mid - \mid * & \text{arithmetic operations} \\
 & \mid & = \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant & \text{arithmetic comparisons} \\
 & \mid & \wedge \mid \vee \mid \neg & \text{Boolean connectives}
\end{array}
$$

- two data types: mathematical integers and Booleans
- well-typed terms evaluate without errors (no division)
- evaluation of a term does not change the program memory

μML: expressions

e	$::=$	$\mathtt{skip}$	do nothing
	$\mid$	t	pure term
	$\mid$	$x \mathrel{\mathtt{:=}} t$	assignment
	$\mid$	$e \mathbin{\mathtt{;}} e$	sequence
	$\mid$	$\mathtt{let}\ v = e\ \mathtt{in}\ e$	binding
	$\mid$	$\mathtt{let}\ x = \mathtt{ref}\ e\ \mathtt{in}\ e$	allocation
	$\mid$	$\mathtt{if}\ t\ \mathtt{then}\ e\ \mathtt{else}\ e$	conditional
	$\mid$	$\mathtt{while}\ t\ \mathtt{do}\ e\ \mathtt{done}$	loop

- three types: integers, Booleans, and `unit`
- references (pointers) are not first-class values
- expressions can allocate and modify memory
- well-typed expressions evaluate without errors

μML : typed expressions

$\texttt{skip} : \texttt{unit}$

$t_\tau : \tau$

$x_\tau \texttt{ := } t_\tau : \texttt{unit}$

$e_{\texttt{unit}} \texttt{ ; } e_\varsigma : \varsigma$

$\texttt{let } v_\tau = e_\tau \texttt{ in } e_\varsigma : \varsigma$

$\texttt{let } x_\tau = \texttt{ref } e_\tau \texttt{ in } e_\varsigma : \varsigma$

$\texttt{if } t_{\texttt{bool}} \texttt{ then } e_\varsigma \texttt{ else } e_\varsigma : \varsigma$

$\texttt{while } t_{\texttt{bool}} \texttt{ do } e_{\texttt{unit}} \texttt{ done} : \texttt{unit}$

- $\tau ::= \texttt{int} \mid \texttt{bool}$ and $\varsigma ::= \tau \mid \texttt{unit}$
- references (pointers) are not first-class values
- expressions can allocate and modify memory
- well-typed expressions evaluate without errors

μML : syntactic sugar

$$
\begin{array}{rcl}
x \texttt{ := } e & \equiv & \texttt{let } v = e \texttt{ in } x \texttt{ := } v \\
\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 & \equiv & \texttt{let } v = e \texttt{ in if } v \texttt{ then } e_1 \texttt{ else } e_2 \\
\texttt{if } e_1 \texttt{ then } e_2 & \equiv & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else skip} \\
e_1 \mathbin{\&\&} e_2 & \equiv & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else false} \\
e_1 \texttt{ || } e_2 & \equiv & \texttt{if } e_1 \texttt{ then true else } e_2
\end{array}
$$

Example — *ISQRT*

```
let sum = ref 1 in
let count = ref 0 in
while sum ⩽ n do
  count := count + 1;
  sum := sum + 2 ∗ count + 1
done;
count
```

What is the result of this expression for a given n?

Example — *ISQRT*

```
let sum = ref 1 in
let count = ref 0 in
while sum ⩽ n do
  count := count + 1;
  sum := sum + 2 ∗ count + 1
done;
count
```

What is the result of this expression for a given `n`?

Informal specification:

- at the end, `count` contains the truncated square root of `n`
- for instance, given $n = 42$, the returned value is 6

Hoare triples

A statement about program correctness:

$$\{P\}\ e\ \{Q\}$$

P	precondition property
e	expression
Q	postcondition property

What is the meaning of a Hoare triple?

$\{P\}\ e\ \{Q\}$	if we execute e in a state that satisfies P, then the computation either diverges or terminates in a state that satisfies Q

This is partial correctness: we do not prove termination.

Examples

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}\ x \mathrel{\texttt{:=}} x + 2\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2 * y\}$
- $\{\exists v.\, x = 4 * v\}\ x + 42\ \{\exists w.\, \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}\ \texttt{while true do skip done}\ \{\boxed{\boxed{\texttt{false}}}\}$

Examples

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}$ $x := x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\mathtt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \mathtt{result} = 2 * w\}$
- $\{\mathtt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\mathtt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

Examples

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\texttt{false}}}\}$
    - after this loop, *everything* is trivially verified
    - ergo: not proving termination can be fatal

In our square root example:

$$\{?\}\ \mathit{ISQRT}\ \{?\}$$

Examples

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\texttt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

In our square root example:

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{?\}$$

Examples

Examples of valid Hoare triples for partial correctness:

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\mathtt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \mathtt{result} = 2 * w\}$
- $\{\mathtt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\mathtt{false}}}\}$
  - after this loop, *everything* is trivially verified
  - ergo: not proving termination can be fatal

In our square root example:

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{\mathtt{result} * \mathtt{result} \leqslant n < (\mathtt{result} + 1) * (\mathtt{result} + 1)\}$$

4. Weakest precondition calculus

Weakest preconditions

How can we establish the correctness of a program?

One solution: Edsger Dijkstra, 1975

Predicate transformer $\mathrm{WP}(e, Q)$

e	expression
Q	postcondition

computes the weakest precondition P such that $\{P\}\ e\ \{Q\}$

Definition of WP

$$
\begin{array}{rcl}
\mathrm{WP}(\mathtt{skip}, Q) & = & Q \\
\mathrm{WP}(t, Q) & = & Q[\mathtt{result} \mapsto t] \\
\mathrm{WP}(x \mathrel{\mathtt{:=}} t, Q) & = & Q[x \mapsto t] \\
\mathrm{WP}(e_1 \mathbin{\mathtt{;}} e_2, Q) & = & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\
\mathrm{WP}(\mathtt{let}\ v = e_1\ \mathtt{in}\ e_2, Q) & = & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \mapsto \mathtt{result}]) \\
\mathrm{WP}(\mathtt{let}\ x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2, Q) & = & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \mapsto \mathtt{result}]) \\
\mathrm{WP}(\mathtt{if}\ t\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, Q) & = & \quad (t \rightarrow \mathrm{WP}(e_1, Q)) \wedge \\
 & & (\neg t \rightarrow \mathrm{WP}(e_2, Q))
\end{array}
$$

Definition of WP: loops

$\mathrm{WP}(\texttt{while}\ t\ \texttt{do}\ e\ \texttt{done}, Q) =$	
$\quad \exists J : \texttt{Prop}.$	some *invariant property* J
$\qquad J \land$	that holds at the loop entry
$\qquad \forall x_1 \ldots x_k.$	and is preserved
$\qquad\quad (J \land \quad t \to \mathrm{WP}(e, J)) \land$	after a single iteration,
$\qquad\quad (J \land \neg t \to Q)$	is strong enough to prove Q

$x_1 \ldots x_k$ references modified in e

We cannot know the values of the modified references after n iterations

- therefore, we prove preservation and the post for arbitrary values
- the invariant must provide all the needed information about the state

Definition of WP: annotated loops

Finding an appropriate invariant is difficult in the general case

- this is equivalent to constructing a proof of Q by induction

We can ease the task of automated tools by providing annotations:

$$\begin{aligned}
&\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{do}\ e\ \texttt{done}, Q) = && \text{the given invariant } J\\
&\quad J \land && \text{holds at the loop entry,}\\
&\quad \forall x_1 \dots x_k. && \text{is preserved after}\\
&\qquad (J \land t \rightarrow \mathrm{WP}(e, J)) \land && \quad \text{a single iteration,}\\
&\qquad (J \land \neg t \rightarrow Q) && \text{and suffices to prove } Q
\end{aligned}$$

$x_1 \dots x_k$ references modified in e

Examples

$$\mathrm{WP}(x \texttt{ := } x + y, x = 2y) \equiv x + y = 2y$$

Examples

$$\mathrm{WP}(x \texttt{ := } x+y, x=2y) \;\equiv\; x+y=2y$$

$$\mathrm{WP}(\texttt{while } y>0 \texttt{ invariant even } y \texttt{ do } y \texttt{ := } y-2 \texttt{ done}, \texttt{even } y) \;\equiv$$

Examples

$$\mathrm{WP}(x \texttt{ := } x+y, x=2y) \;\equiv\; x+y=2y$$

$$\begin{aligned}
&\mathrm{WP}(\texttt{while } y>0 \texttt{ invariant } \mathsf{even}\ y \texttt{ do } y \texttt{ := } y-2 \texttt{ done}, \mathsf{even}\ y) \;\equiv\\
&\quad \mathsf{even}\ y \wedge\\
&\quad \forall y.(\mathsf{even}\ y \wedge y>0 \rightarrow \mathsf{even}\ (y-2)) \wedge\\
&\quad \forall y.(\mathsf{even}\ y \wedge y \leqslant 0 \rightarrow \mathsf{even}\ y)
\end{aligned}$$

Soundness of WP

Theorem

For any e and Q, the triple $\{\mathrm{WP}(e, Q)\}\ e\ \{Q\}$ is valid.

Can be proved by induction on the structure of the program e w.r.t. some reasonable semantics (axiomatic, operational, etc.)

Corollary

To show that $\{P\}\ e\ \{Q\}$ is valid, it suffices to prove $P \rightarrow \mathrm{WP}(e, Q)$.

This is what WHY3 does.

5. Run-time safety

Run-time errors

Some operations can fail if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

Run-time errors

Some operations can fail if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

A correct program must not fail:

$\{P\}\ e\ \{Q\}$ if we execute e in a state that satisfies P,
then the computation either diverges
or terminates normally in a state that satisfies Q

Assertions

A new kind of expression:

$$\begin{array}{lll} e & ::= & \dots \\ & | & \texttt{assert}\ R \end{array}$$

fail if R does not hold

The corresponding weakest precondition rule:

$$\mathrm{WP}(\texttt{assert}\ R, Q) \;=\; R \wedge Q \;=\; R \wedge (R \rightarrow Q)$$

The second version is useful in practical deductive verification.

Unsafe operations

We could add other partially defined operations to the language:

$$\begin{array}{rcll} e & ::= & \dots & \\ & \mid & t \ \texttt{div} \ t & \text{Euclidean division} \\ & \mid & a\,[\,t\,] & \text{array access} \\ & \mid & \dots & \end{array}$$

and define the WP rules for them:

$$\begin{array}{rcl} \mathrm{WP}(t_1 \ \texttt{div} \ t_2, Q) & = & t_2 \neq 0 \wedge Q[\texttt{result} \mapsto (t_1 \ \texttt{div} \ t_2)] \\ \mathrm{WP}(a[t], Q) & = & 0 \leqslant t < |a| \wedge Q[\texttt{result} \mapsto a[t]] \\ & \dots & \end{array}$$

But we would rather let the programmers do it themselves.

6. Functions and contracts

Subroutines

We may want to delegate some functionality to functions:

$\texttt{let}\ f\ (v_1 : \tau_1) \dots (v_n : \tau_n) : \varsigma\ \mathscr{C} = e$	defined function
$\texttt{val}\ f\ (v_1 : \tau_1) \dots (v_n : \tau_n) : \varsigma\ \mathscr{C}$	abstract function

Function behaviour is specified with a contract:

$\mathscr{C} ::=$	$\texttt{requires}\ P$	precondition
	$\texttt{writes}\ x_1 \dots x_k$	modified global references
	$\texttt{ensures}\ Q$	postcondition

Postcondition Q may refer to the initial value of a global reference: x°

```
let incr_r (v: int): int writes x
  ensures result = x° ∧ x = x° + v
= let u = x in x := u + v ; u
```

Subroutines

We may want to delegate some functionality to functions:

$\texttt{let}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C} = e$	defined function
$\texttt{val}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C}$	abstract function

Function behaviour is specified with a contract:

$\mathscr{C}$ ::=	$\texttt{requires}\ P$	precondition
	$\texttt{writes}\ x_1 \ldots x_k$	modified global references
	$\texttt{ensures}\ Q$	postcondition

Postcondition Q may refer to the initial value of a global reference: x°

Verification condition ($\vec{x}$ are all global references mentioned in f):

$$\mathrm{VC}(\texttt{let}\ f \ldots) \quad = \quad \forall \vec{x} \vec{v}.\, P \rightarrow \mathrm{WP}(e, Q)[\vec{x}^\circ \mapsto \vec{x}]$$

GOSUB

One more expression:

$$\begin{array}{rcll} e & ::= & \dots & \\ & | & f\ \ t\ \dots\ t & \qquad \text{function call} \end{array}$$

and its weakest precondition rule:

$$\begin{aligned} \mathrm{WP}(f\ \ t_1\ \dots\ t_n, Q) \quad = \quad & P_f[\vec{v} \mapsto \vec{t}\,] \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \mathtt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \end{aligned}$$

P_f	precondition of f	$\vec{\boldsymbol{x}}$	references modified in f
Q_f	postcondition of f	$\vec{x}$	references used in f
$\vec{v}$	formal parameters of f	$\vec{w}$	fresh variables

GOSUB

One more expression:

$$\begin{array}{lcll} e & ::= & \ldots & \\ & | & f\ t\ \ldots\ t & \text{function call} \end{array}$$

and its weakest precondition rule:

$$\begin{aligned} \mathrm{WP}(f\ t_1\ \ldots\ t_n, Q) \;=\; & P_f[\vec{v} \mapsto \vec{t}\,] \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \mathtt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \end{aligned}$$

P_f	precondition of f	$\vec{\boldsymbol{x}}$	references modified in f
Q_f	postcondition of f	$\vec{x}$	references used in f
$\vec{v}$	formal parameters of f	$\vec{w}$	fresh variables

Modular proof: when verifying a function call,
we only use the function's contract, not its code.

Examples

```
let max (x y : int) : int
  requires { true }
  ensures  { result >= x /\ result >= y }
  ensures  { result =  x \/ result =  y }
= if x >= y then x else y
```

```
val r : ref int  (* declare a global reference *)

let incr_r (v : int) : int
  writes   { r }
  ensures  { result = old !r /\ !r = old !r + v }
= let u = !r in
  r := u + v;
  u
```

7. Total correctness: termination

Termination

Problem: prove that the program terminates for every initial state that satisfies the precondition.

It suffices to show that

- every loop makes a finite number of iterations
- recursive function calls cannot go on indefinitely

Solution: prove that every loop iteration and every recursive call decreases a certain value, called variant, with respect to some well-founded order.

For example, for signed integers, a practical well-founded order is

$$i \prec j \quad = \quad i < j \land 0 \leqslant j$$

Loop termination

A new annotation:

$$\begin{array}{rcl} e & ::= & \dots \\ & | & \texttt{while } t \texttt{ invariant } J \texttt{ variant } t \cdot \prec \texttt{ do } e \texttt{ done} \end{array}$$

The corresponding weakest precondition rule:

$$\begin{array}{l} \mathrm{WP}(\texttt{while } t \texttt{ invariant } J \texttt{ variant } s \cdot \prec \texttt{ do } e \texttt{ done}, Q) = \\ \quad J \wedge \\ \quad \forall x_1 \dots x_k. \\ \qquad (J \wedge t \rightarrow \mathrm{WP}(e, J \wedge s \prec w)[w \mapsto s]) \wedge \\ \qquad (J \wedge \neg t \rightarrow Q) \end{array}$$

$x_1 \dots x_k$ references modified in e

w a fresh variable (the variant at the start of the iteration)

Termination of recursive functions

A new contract clause:

$$
\begin{array}{l}
\texttt{let rec}\ f\ (v_1 : \tau_1)\ \ldots\ (v_n : \tau_n)\ :\ \varsigma \\
\qquad\qquad \texttt{requires}\ P_f \\
\qquad\qquad \texttt{variant}\ s \cdot \prec \\
\qquad\qquad \texttt{writes}\ \vec{\boldsymbol{x}} \\
\qquad\qquad \texttt{ensures}\ Q_f \\
\qquad\quad = e
\end{array}
$$

For each recursive call of f in e:

$$
\begin{aligned}
\mathrm{WP}(f\ t_1\ \ldots\ t_n, Q) \quad = \quad & P_f[\vec{v} \mapsto \vec{t}\,] \wedge s[\vec{v} \mapsto \vec{t}\,] \prec s[\vec{x} \mapsto \vec{x}^\circ] \wedge \\
& (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^\circ \mapsto \vec{w}] \rightarrow Q)[\vec{w} \mapsto \vec{x}]
\end{aligned}
$$

$s[\vec{v} \mapsto \vec{t}\,]$	variant at the call site	$\vec{x}$	references used in f
$s[\vec{x} \mapsto \vec{x}^\circ]$	variant at the start of f	$\vec{w}$	fresh variables

Mutual recursion

Mutually recursive functions must have

- their own variant terms
- a common well-founded order

Thus, if f calls $g\ t_1\ \ldots\ t_n$, the variant decrease precondition is

$$s_g[\vec{v}_g \mapsto \vec{t}] \prec s[\vec{x} \mapsto \vec{x}^\circ]$$

$\vec{v}_g$	the formal parameters of g
s_g	the variant of g

8. Exceptions

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
    - total correctness ensures against non-termination
- abnormal termination — the computation fails
    - partial correctness ensures against run-time errors

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination
- abnormal termination — the computation fails
  - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
  - partial correctness ensures conformance to the contract

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination
- abnormal termination — the computation fails
  - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination — produces *a different kind of result*

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination
- abnormal termination — the computation fails
  - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination — produces *a different kind of result*
  - the contract should also cover exceptional termination

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination
- abnormal termination — the computation fails
  - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination — produces *a different kind of result*
  - the contract should also cover exceptional termination
  - each potential exception E gets its own postcondition Q_{E}

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
    - total correctness ensures against non-termination
- abnormal termination — the computation fails
    - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
    - partial correctness ensures conformance to the contract
- exceptional termination — produces *a different kind of result*
    - the contract should also cover exceptional termination
    - each potential exception E gets its own postcondition Q_{E}
    - partial correctness: *if* E *is raised, then* Q_{E} *holds*

Exceptions as destinations

Execution of a program can lead to

- divergence — the computation never ends
  - total correctness ensures against non-termination
- abnormal termination — the computation fails
  - partial correctness ensures against run-time errors
- normal termination — the computation produces a result
  - partial correctness ensures conformance to the contract
- exceptional termination — produces *a different kind of result*

```
exception Not_found

let binary_search (a: array int) (v: int) : int
  requires { forall i j. 0 ⩽ i ⩽ j < length a → a[i] ⩽ a[j] }
  ensures  { 0 ⩽ result < length a ∧ a[result] = v }
  raises   { Not_found → forall i. 0 ⩽ i < length a → a[i] ≠ v }
```

Just another semicolon

Our language keeps growing:

e	$::=$	$\dots$	
	$\mid$	`raise E`	raise an exception
	$\mid$	`try` e `with E` $\rightarrow$ e	catch an exception

WP handles two postconditions now:

$$\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) \quad = \quad Q$$

Just another semicolon

Our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E} & \text{raise an exception} \\
 & | & \texttt{try } e \texttt{ with E} \rightarrow e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{array}{rcl}
\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) & = & Q \\
\mathrm{WP}(\texttt{raise E}, Q, Q_{\mathsf{E}}) & = & Q_{\mathsf{E}}
\end{array}
$$

Just another semicolon

Our language keeps growing:

$$
\begin{array}{rcll}
e & ::= & \dots & \\
 & \mid & \texttt{raise}\ \mathsf{E} & \text{raise an exception} \\
 & \mid & \texttt{try}\ e\ \texttt{with}\ \mathsf{E} \rightarrow e & \text{catch an exception}
\end{array}
$$

WP handles two postconditions now:

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) &= Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_{\mathsf{E}}) &= Q_{\mathsf{E}} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_{\mathsf{E}}) &= \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}), Q_{\mathsf{E}})
\end{aligned}
$$

Just another semicolon

Our language keeps growing:

e	$::=$	$\dots$	
	$\mid$	`raise` E	raise an exception
	$\mid$	`try` e `with` E $\rightarrow$ e	catch an exception

WP handles two postconditions now:

$$\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) &= Q \\
\mathrm{WP}(\texttt{raise}\ \mathsf{E}, Q, Q_{\mathsf{E}}) &= Q_{\mathsf{E}} \\
\mathrm{WP}(e_1\ ;\ e_2, Q, Q_{\mathsf{E}}) &= \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}), Q_{\mathsf{E}}) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E} \rightarrow e_2, Q, Q_{\mathsf{E}}) &= \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}}))
\end{aligned}$$

Just another let-in

Exceptions can carry data:

$$\begin{array}{rcll} e & ::= & \dots & \\ & | & \texttt{raise}\ \mathsf{E}\ t & \text{raise an exception} \\ & | & \texttt{try}\ e\ \texttt{with}\ \mathsf{E}\ v \rightarrow e & \text{catch an exception} \end{array}$$

Still, all needed mechanisms are already in WP:

$$\begin{array}{rcl} \mathrm{WP}(t, Q, Q_{\mathsf{E}}) & = & Q[\texttt{result} \mapsto t] \\ \mathrm{WP}(\texttt{raise}\ \mathsf{E}\ t, Q, Q_{\mathsf{E}}) & = & Q_{\mathsf{E}}[\texttt{result} \mapsto t] \\ \mathrm{WP}(\texttt{let}\ v = e_1\ \texttt{in}\ e_2, Q, Q_{\mathsf{E}}) & = & \\ & & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}})[v \mapsto \texttt{result}], Q_{\mathsf{E}}) \\ \mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ \mathsf{E}\ v \rightarrow e_2, Q, Q_{\mathsf{E}}) & = & \\ & & \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_{\mathsf{E}})[v \mapsto \texttt{result}]) \end{array}$$

Functions with exceptions

A new contract clause:

$$\begin{array}{l} \texttt{let}\ f\ (v_1 : \tau_1)\ \ldots\ (v_n : \tau_n)\ :\ \varsigma \\ \qquad \texttt{requires}\ P_f \\ \qquad \texttt{writes}\ \vec{\boldsymbol{x}} \\ \qquad \texttt{ensures}\ Q_f \\ \qquad \texttt{raises}\ \mathsf{E} \rightarrow Q_{\mathsf{E}f} \\ \quad = e \end{array}$$

Verification condition for the function definition:

$$\mathrm{VC}(\texttt{let}\ f\ \ldots) \quad = \quad \forall \vec{x}\,\vec{v}.\, P_f \rightarrow \mathrm{WP}(e, Q_f, Q_{\mathsf{E}f})[\vec{x}^{\circ} \mapsto \vec{x}]$$

Weakest precondition rule for the function call:

$$\begin{aligned} \mathrm{WP}(f\ t_1\ \ldots\ t_n, Q, Q_{\mathsf{E}}) \quad = \quad & P_f[\vec{v} \mapsto \vec{t}\,] \ \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\ Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \rightarrow Q)[\vec{w} \mapsto \vec{x}] \ \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\ Q_{\mathsf{E}f}[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \rightarrow Q_{\mathsf{E}})[\vec{w} \mapsto \vec{x}] \end{aligned}$$

9. WHYML types

WHYML types

WHYML supports most of the OCaml types:

- polymorphic types

```
type set 'a
```

- tuples:

```
type poly_pair 'a = ('a, 'a)
```

- records:

```
type complex = { re : real; im : real }
```

- variants (sum types):

```
type list 'a = Cons 'a (list 'a) | Nil
```

Algebraic types

To handle algebraic types (records, variants):

- access to record fields:

```
let get_real (c : complex) = c.re
let use_imagination (c : complex) = im c
```

- record updates:

```
let conjugate (c : complex) = { c with im = - c.im }
```

- pattern matching (no `when` clauses):

```
let rec length (l : list 'a) : int variant { l } =
  match l with
  | Cons _ ll -> 1 + length ll
  | Nil -> 0
  end
```

Abstract types

Abstract types must be axiomatized:

```
theory Map
  type map 'a 'b

  function ([])   (a: map 'a 'b) (i: 'a): 'b
  function ([<-]) (a: map 'a 'b) (i: 'a) (v: 'b): map 'a 'b

  axiom Select_eq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 = k2  -> m[k1 <- v][k2] = v

  axiom Select_neq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 <> k2 -> m[k1 <- v][k2] = m[k2]
end
```

Abstract types (cont.)

Abstract types must be axiomatized:

```
theory Set
  type set 'a
  predicate mem 'a (set 'a)

  predicate (==) (s1 s2: set 'a) =
    forall x: 'a. mem x s1 <-> mem x s2
  axiom extensionality:
    forall s1 s2: set 'a. s1 == s2 -> s1 = s2

  predicate subset (s1 s2: set 'a) =
    forall x: 'a. mem x s1 -> mem x s2
  lemma subset_refl: forall s: set 'a. subset s s

  constant empty : set 'a
  axiom empty_def: forall x: 'a. not (mem x empty)
  ...
```

Logical language of WHYML

- the same types are available in the logical language
- `match-with-end`, `if-then-else`, `let-in`
  are accepted both in terms and formulas
- functions et predicates can be defined recursively:

```
predicate mem (x: 'a) (l: list 'a) = match l with
  Cons y r -> x = y \/ mem x r | Nil -> false end
```

  no `variant`s, WHY3 requires structural decrease

- inductive predicates (useful for transitive closures):

```
inductive sorted (l: list int) =
  | SortedNil: sorted Nil
  | SortedOne: forall x: int. sorted (Cons x Nil)
  | SortedTwo: forall x y: int, l: list int.
      x <= y -> sorted (Cons y l) ->
        sorted (Cons x (Cons y l))
```

10. Ghost code

Ghost code: example

Compute a Fibonacci number using a recursive function in $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires {                                                }
  ensures  {
                                                            }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Ghost code: example

Compute a Fibonacci number using a recursive function in $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) /\
                                            result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Ghost code: example

Instead of an existential we can use a ghost parameter:

```
let rec aux (a b n: int) (ghost k:  int): int
  requires { 0 <= n }
  requires { 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1) (k+1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n 0
```

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too

The spirit of ghost code

Ghost code is used to facilitate specification and proof

⇒ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
    - if k is ghost, then $(k + 1)$ is ghost, too
- ghost code cannot modify normal data
    - if r is a normal reference, then $r := $ `ghost` k is forbidden

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
    - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify normal data
    - if r is a normal reference, then $r :=$ `ghost` k is forbidden
- ghost code cannot alter the control flow of normal code
    - if c is ghost, then `if c then ...` and `while c do ... done` are ghost

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome

Consequently:

- normal code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify normal data
  - if r is a normal reference, then $r :=$ `ghost` k is forbidden
- ghost code cannot alter the control flow of normal code
  - if c is ghost, then `if c then ...` and `while c do ... done` are ghost
- ghost code cannot diverge
  - we can prove `while true do skip done ; assert { false }`

Ghost code in WHYML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

Ghost code in WHYML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;  (* get from head *)
                  tail: list 'a;  (* add to tail *)
            ghost elts: list 'a;  (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

Ghost code in WHYML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;   (* get from head *)
                  tail: list 'a;   (* add to tail *)
            ghost elts: list 'a;   (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- local variables and functions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

Ghost code in WHYML

Can be declared ghost:

- function parameters

```
val aux (a b n: int) (ghost k: int): int
```

- record fields and variant fields

```
type queue 'a = { head: list 'a;   (* get from head *)
                  tail: list 'a;   (* add to tail *)
            ghost elts: list 'a;   (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- local variables and functions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

- program expressions

```
let x = ghost qu.elts in ...
```

Lemma functions

General idea: a function $f\ \vec{x}$ `requires` P_f `ensures` Q_f that

- returns `unit`
- has no side effects
- terminates

provides a constructive proof of $\forall \vec{x}.P_f \rightarrow Q_f$

$\Rightarrow$ a pure recursive function simulates a proof by induction

Lemma functions

General idea: a function $f\ \vec{x}$ `requires` P_f `ensures` Q_f that

- returns `unit`
- has no side effects
- terminates

provides a constructive proof of $\forall \vec{x}. P_f \rightarrow Q_f$

$\Rightarrow$ a pure recursive function simulates a proof by induction

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
=
  match l with Cons a ll -> length_rev_append ll (Cons a r)
             | Nil -> () end
```

Lemma functions

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
=
  match l with Cons a ll -> length_rev_append ll (Cons a r)
             | Nil -> () end
```

- by the postcondition of the recursive call:

  `length (rev_append ll (Cons a r)) = length ll + length (Cons a r)`

- by definition of `rev_append`:

  `rev_append (Cons a ll) r = rev_append ll (Cons a r)`

- by definition of `length`:

  `length (Cons a ll) + length r = length ll + length (Cons a r)`

11. Mutable data

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`
- cannot be stored in recursive structures: no `list (ref ’a)`

Records with mutable fields

```
module Ref
  type ref 'a = { mutable contents : 'a } (* as in OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- can be passed between functions as arguments and return values
- can be created locally or declared globally
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- can hold ghost data
  - `let ghost r := ref 42 in ... ghost (r := -!r) ...`
- cannot be stored in recursive structures: no `list (ref 'a)`
- cannot be stored under abstract types: no `set (ref 'a)`

The problem of alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* write/write alias *)
  assert { !s = 1 /\ !s = 2 }   (* in fact, !s = 3 *)
```

The problem of alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;    (* write/write alias *)
  assert { !s = 1 /\ !s = 2 }    (* in fact, !s = 3 *)
```

```
val g : ref int

let set_from_g (r: ref int): unit writes {r}
  ensures { !r = !g + 1 }
= r := !g + 1

let wrong () =
  set_from_g g;    (* read/write alias *)
  assert { !g = !g + 1 }    (* contradiction *)
```

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- WHY3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- WHY3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The user must indicate the external dependencies of abstract functions:

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- otherwise the static control of aliases does not have enough information

The problem of alias

The Hoare logic, the WP calculus require the absence of aliases!

- at least for modified values
- Why3 verifies statically the absence of illegal aliases
- any mutable data returned by a function is fresh

The user must indicate the external dependencies of abstract functions:

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- otherwise the static control of aliases does not have enough information

For programs with arbitrary pointers we need more sophisticated tools

- memory models (for example, “address-to-value” arrays)
- handle aliases in the VC: separation logic, dynamic frames, etc.

Abstract specification

Aliasing restrictions in WHYML

⇒ certain structures cannot be implemented

Still, we can specify them and verify the client code

```
type array 'a model { mutable elts: map int 'a;
                              length: int }
invariant { 0 <= self.length }
```

- fields `length` et `elts` can only be used in annotations (model type)
- all access is done via abstract functions
- the type invariant is verified at the boundaries of function calls
  - WHY3 implicitly adds the necessary pre- et postconditions

Abstract specification

```
type array 'a model { mutable elts: map int 'a;
                                length: int }
invariant { 0 <= self.length }

val ([]) (a: array 'a) (i: int): 'a
  requires { 0 <= i < a.length }
  ensures  { result = a.elts[i] }

val ([]<-) (a: array 'a) (i: int) (v: 'a): unit writes {a}
  requires { 0 <= i < a.length }
  ensures  { a.elts = (old a.elts)[i <- v] }

val length (a: array 'a): int ensures { result = a.length }

function get (a: array 'a) (i: int): 'a = a.elts[i]
```

- the immutable fields are preserved — implicit postcondition
- the logical function `get` has no precondition
    - its result outside of the array bounds is undefined

12. Modular programming considered useful

Declarations

- types
  - abstract: `type t`
  - synonym: `type t = list int`
  - variant: `type list 'a = Nil | Cons 'a (list 'a)`

- functions / predicates
  - uninterpreted: `function f int: int`
  - defined: `predicate non_empty (l: list 'a) = l <> Nil`
  - inductive: `inductive path t (list t) t = ...`

- axioms / lemmas / goals
  - `goal G: forall x: int, x >= 0 -> x*x >= 0`

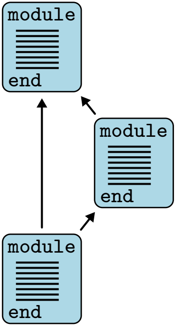
- program functions (*routines*)
  - abstract: `val ([]) (a: array 'a) (i: int): 'a`
  - defined: `let mergesort (a: array elt): unit = ...`

- exceptions
  - `exception Found int`

Modules

Declarations are organized in modules

- purely logical modules are called theories



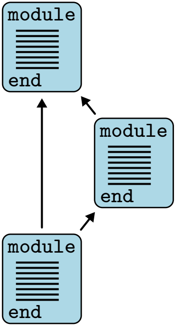
Modules

Declarations are organized in modules

- purely logical modules are called theories

A module M_1 can be

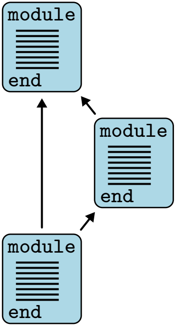
- used (`use`) in a module M_2
  - symbols of M_1 are shared
  - axioms of M_1 remain axioms
  - lemmas of M_1 become axioms
  - goals of M_1 are ignored



Modules

Declarations are organized in modules

- purely logical modules are called theories



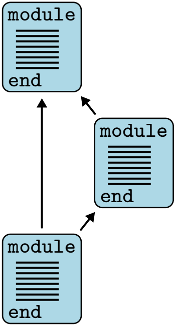
A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2
  - declarations of M_1 are copied or instantiated
  - axioms of M_1 remain axioms or become lemmas
  - lemmas of M_1 become axioms
  - goals of M_1 are ignored

Modules

Declarations are organized in modules

- purely logical modules are called theories



A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2

Cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function
- a `val` with a `let`

Modules

Declarations are organized in modules

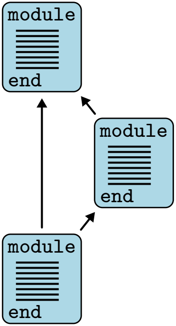
- purely logical modules are called theories

A module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2

Cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function
- a `val` with a `let`



One missing piece coming soon:

- instantiate a used module with another module

Exercises

`http://why3.lri.fr/ejcp-2017`