

Why3

A Multi-Prover Platform for Program Verification

Jean-Christophe Filliâtre
CNRS

joint work with
Andrei Paskevich, Claude Marché, and François Bobot

ProVal team, Orsay, France

IFIP WG 1.9/2.14 “Verified Software”
June 2011



INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche **BACLAY** - ÎLE-DE-FRANCE



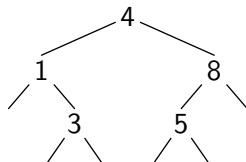
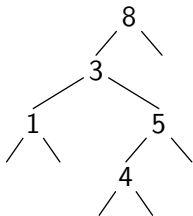
UNIVERSITÉ
PARIS-SUD 11

chapter 1

A Tale of Two Programs

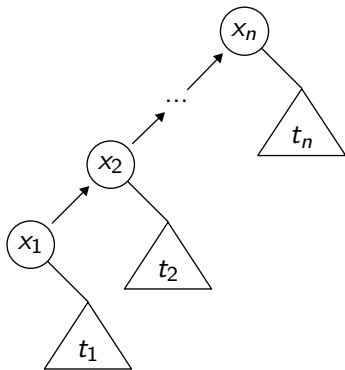
Program 1: Same Fringe

given two binary trees,
do they contain the same elements when traversed in order?



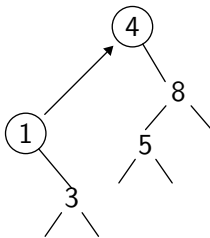
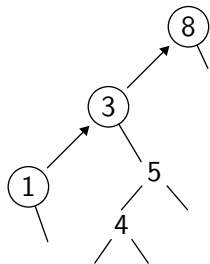
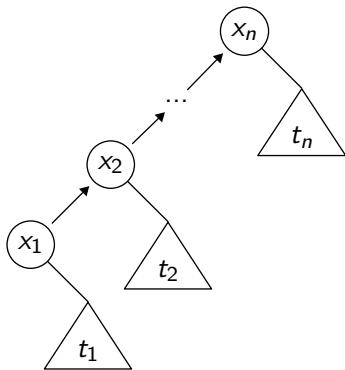
Same Fringe: A Solution

build the left spine as a bottom-up list



Same Fringe: A Solution

build the left spine as a bottom-up list



Same Fringe

a program is contained in a module

```
module SameFringe
```

we introduce an abstract type for elements

```
type elt
```

and an algebraic type for trees

```
type tree =  
  | Empty  
  | Node tree elt tree
```

Same Fringe

the list of elements traversed in order is

```
logic elements (t: tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end
```

so the specification is as simple as

```
let same_fringe t1 t2 =
  { }
  ...
  { result=True ↔ elements t1 = elements t2 }
```

Same Fringe

a datatype for the left spine of a tree, as a bottom-up list

```
type enum =  
  | Done  
  | Next elt tree enum
```

and the list of its elements

```
logic enum_elements (e: enum) : list elt = match e with  
  | Done → Nil  
  | Next x r e → Cons x (elements r ++ enum_elements e)  
end
```

Same Fringe

the left spine of a given tree

```
let rec enum t e =  
  { }  
  match t with  
  | Empty → e  
  | Node l x r → enum l (Next x r e)  
end  
{ enum_elements result = elements t ++ enum_elements e }
```

Same Fringe

idea: comparing enums is easier

```
let rec eq_enum e1 e2 =  
  { }  
  match e1, e2 with  
  | Done, Done →  
    True  
  | Next x1 r1 e1, Next x2 r2 e2 →  
    x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)  
  | _ →  
    False  
end  
{ result=True ↔ enum_elements e1 = enum_elements e2 }
```

Same Fringe

and it degenerates into a solution for the same fringe

```
let same_fringe t1 t2 =  
  { }  
  eq_enum (enum t1 Done) (enum t2 Done)  
  { result=True  $\leftrightarrow$  elements t1 = elements t2 }
```

all VCs are proved automatically

Same Fringe: Termination

additionally, we can prove termination of functions `enum` and `eq_enum`, by providing **variants**

unless specified otherwise,
a variant is a non-negative, strictly decreasing integer quantity

```
let rec enum t e variant { length (elements t) } =  
  ...
```

```
let rec eq_enum e1 e2 variant { length (enum_elements e1) }  
  ...
```

Program 2: Sparse Arrays

from VACID-0

Verification of Ample Correctness of Invariants of Data-structures

Rustan Leino and Michał Moskal (2010)

sparse arrays

array data structure with create, get and set in $O(1)$ time

difficulty

create N d allocates an array of size N with a default value d
but you can't afford $O(N)$ to make the initialization

Sparse Arrays: A Solution

use 3 arrays and keep track of meaningful values

		<i>b</i>		<i>a</i>		<i>c</i>	
val		<i>y</i>		<i>x</i>		<i>z</i>	

idx		1		0		2	
-----	--	---	--	---	--	---	--

	0	1	2				
back	<i>a</i>	<i>b</i>	<i>c</i>				

$n = 3$

Sparse Arrays

we use arrays from the standard library

```
use import module array.Array as A
```

and implement a sparse array as a record

```
type sparse_array 'a = { | val   : array 'a;  
                           idx   : array int;  
                           back  : array int;  
                           mutable card : int;  
                           default : 'a; | }
```

note that

- ▶ field card is mutable
- ▶ fields val, idx, and back contain mutable arrays
- ▶ field default is immutable

Sparse Arrays

the validity of an index:

```
logic is_elt (a: sparse_array 'a) (i: int) =  
  0 ≤ a.idx[i] < a.card ∧ a.back[a.idx[i]] = i
```

the model of a sparse array is a function from integer to values

```
logic value (a: sparse_array 'a) (i: int) : 'a =  
  if is_elt a i then a.val[i] else a.default
```

Sparse Arrays

it is convenient to introduce the length of a sparse array

```
logic length (a: sparse_array 'a) : int = A.length a.val
```

the data structure **invariant**:

```
logic sa_invariant (a: sparse_array 'a) =  
  0 ≤ a.card ≤ length a ≤ maxlen ∧  
  A.length a.val = A.length a.idx = A.length a.back ∧  
  ∀ i: int.  
    0 ≤ i < a.card →  
    0 ≤ a.back[i] < length a ∧ a.idx[a.back[i]] = i
```

Sparse Arrays

creation assumes we can allocate memory

`parameter malloc:`

`n:int → {} array 'a { A.length result = n }`

`let create (sz: int) (d: 'a) =`

`{ 0 ≤ sz ≤ maxlen }`

`{ | val = malloc sz;`

`idx = malloc sz;`

`back = malloc sz;`

`card = 0;`

`default = d | }`

`{ sa_invariant result ∧ result.card = 0 ∧`

`result.default = d ∧ length result = sz }`

Sparse Arrays

we can test the validity of an index

```
let test (a: sparse_array 'a) i =  
  { 0 ≤ i < length a ∧ sa_invariant a }  
  0 ≤ a.idx[i] && a.idx[i] < a.card && a.back[a.idx[i]]  
  { result=True ↔ is_elt a i }
```

(different from is_elt: it does array bounds checking)

accessing an element

```
let get (a: sparse_array 'a) i =  
  { 0 ≤ i < length a ∧ sa_invariant a }  
  if test a i then  
    a.val[i]  
  else  
    a.default  
  { result = value a i }
```

Sparse Arrays

assignment

```
let set (a: sparse_array 'a) (i: int) (v: 'a) =  
  {  $0 \leq i < \text{length } a \wedge \text{sa\_invariant } a$  }  
  a.val[i] <- v;  
  if not (test a i) then begin  
    assert {  $a.\text{card} < \text{length } a$  };  
    a.idx[i] <- a.card;  
    a.back[a.card] <- i;  
    a.card <- a.card + 1  
  end  
  {  $\text{sa\_invariant } a \wedge$   
     $\text{value } a \ i = v \wedge$   
     $\forall j:\text{int}. j \neq i \rightarrow \text{value } a \ j = \text{value } (\text{old } a) \ j$  }
```

Sparse Arrays

one lemma proved with Coq

```
lemma permutation:
```

```
  ∀ a: sparse_array 'a. sa_invariant a →
```

```
    a.card = a.length →
```

```
    ∀ i: int. 0 ≤ i < a.length →
```

```
      0 ≤ a.idx[i] < a.length && a.back[a.idx[i]] = i
```

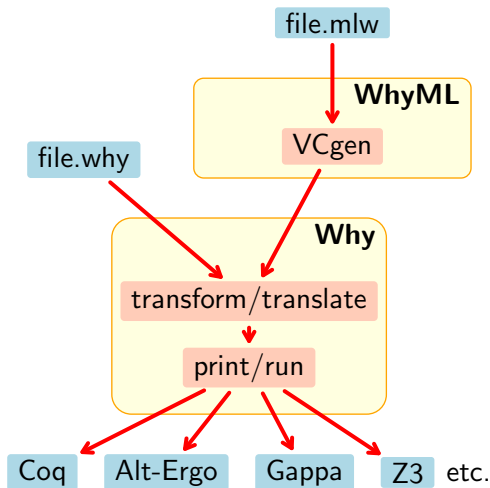
then all VCs are proved automatically

chapter 2

Big Picture

Why3 Overview

started Feb 2010 / authors: JCF, A. Paskevich, C. Marché,
F. Bobot



Logic

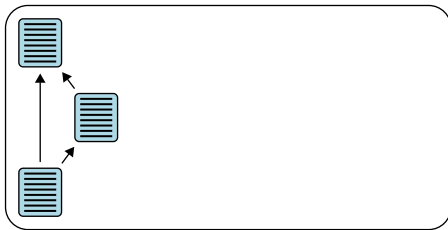
first-order polymorphic logic

- ▶ (mutually) (recursive) algebraic types
- ▶ (mutually) (recursive) functions and predicates
- ▶ (mutually) inductive predicates
- ▶ axioms / lemmas / goals

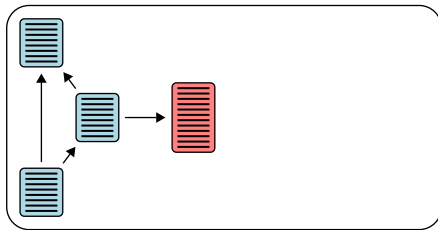
organized in theories

- ▶ a theory may **use** another theory (sharing)
- ▶ a theory may **clone** another theory (copy + substitution)

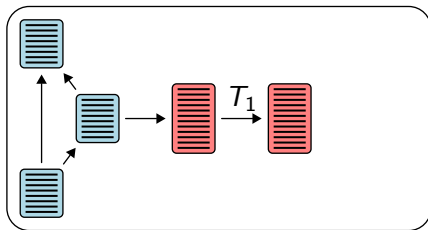
From Theories to Provers



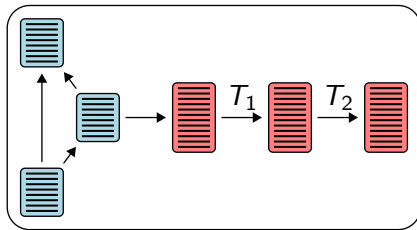
From Theories to Provers



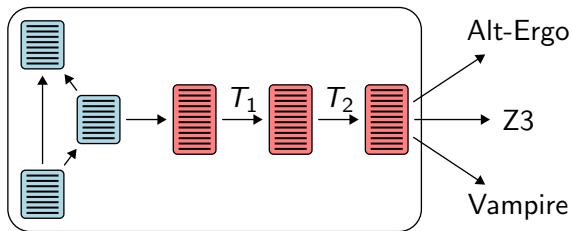
From Theories to Provers



From Theories to Provers



From Theories to Provers



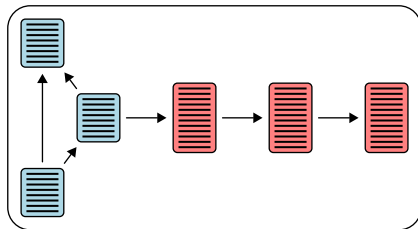
Prover Drivers

- ▶ transformations to apply
- ▶ output format
 - ▶ pretty-printer
 - ▶ built-in symbols
 - ▶ built-in axioms
- ▶ prover's diagnostic messages

Architecture: API and Plug-ins

Your code

Why3 API



Architecture: API and Plug-ins

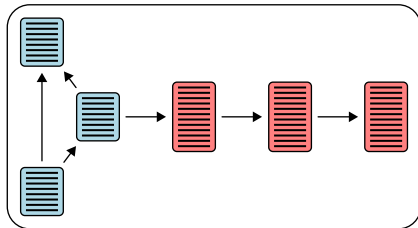
Your code

Why3 API

WhyML

TPTP

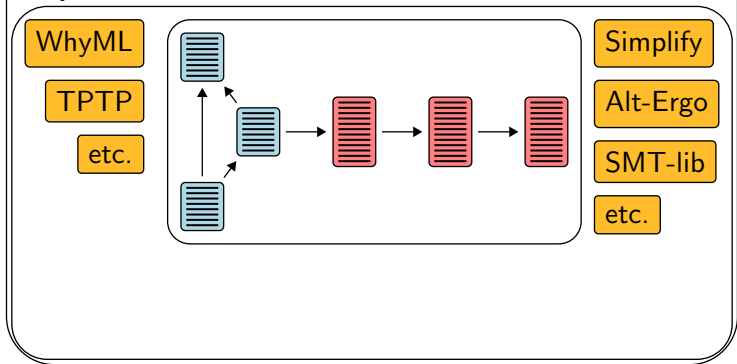
etc.



Architecture: API and Plug-ins

Your code

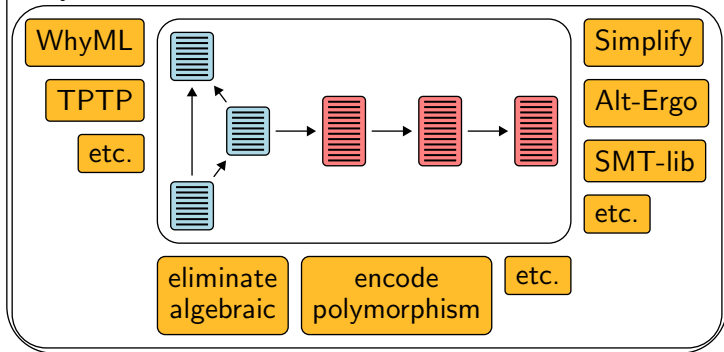
Why3 API



Architecture: API and Plug-ins

Your code

Why3 API



A Programming Language: WhyML

first-order ML

- ▶ purely applicative expressions (logical terms)
- ▶ effects: mutable data structures, exceptions, non-termination
- ▶ annotations (pre/post/assert/loop invariants)
- ▶ a notion of modules, analogous to theories
- ▶ encapsulation

appendix A

Theories

Starting Points

1. keep contexts **small** when using provers
 - ⇒ split specifications into small libraries
 - ⇒ we call them **theories**
2. they must be **reusable** pieces of specification
 - ⇒ they must be parameterized
 - ⇒ we generalize the idea with the notion of **theory cloning**

Theory

a theory groups declarations together

declarations introduce

- ▶ types
- ▶ symbols
- ▶ axioms
- ▶ goals

Example

```
theory List

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

  logic mem (x: 'a) (l: list 'a) = match l with
    | Nil → false
    | Cons y r → x = y ∧ mem x r
  end

  goal G1: mem 2 (Cons 1 (Cons 2 (Cons 3 Nil)))

end
```

Using a Theory

a theory T_2 can **use** another theory T_1

- ▶ the symbols of T_1 are **shared**
- ▶ goals from T_1 are ignored
- ▶ lemmas from T_1 become axioms
- ▶ axioms from T_1 stay axioms

Example

```
theory Length

  use import List
  use import int.Int

  logic length (l: list 'a) : int = match l with
    | Nil          → 0
    | Cons _ r    → 1 + length r
  end

  goal G2: length (Cons 1 (Cons 2 (Cons 3 Nil))) = 3

  lemma length_nonnegative:  $\forall l:\text{list } 'a. \text{length}(l) \geq 0$ 

end
```

Theory Cloning

a theory T_2 can **clone** another theory T_1

it makes a **copy** of T_1 with a possible **instantiation** of some of its abstract types and symbols

$\Rightarrow$ any theory is parameterized w.r.t. any subset of its abstract symbols

Examples

the standard library for integer arithmetic (demo)

Einstein's puzzle (demo)

Cloning a Theory

- ▶ axioms from T_1 remain axioms or become lemmas/goals
- ▶ lemmas from T_1 become axioms
- ▶ goals from T_1 are ignored

the user specifies those axioms to be turned into goals
(optimistic approach)

Example

```
theory Bijection
  use export int.Int
  logic n : int
  logic f int : int
  axiom dom_f :  $\forall i: \text{int}. 0 \leq i < n \rightarrow 0 \leq f\ i < n$ 
  logic g int : int
  axiom gf :  $\forall i: \text{int}. 0 \leq i < n \rightarrow g\ (f\ i) = i$ 
end

theory Test1
  logic id (i: int) : int = i
  clone import Bijection with logic f = id, lemma dom_f
  ...
end
```

note: theory int.Int is used in Test1 (not cloned)

Use and Clone

a given theory can be both **used** and **cloned**

example:

if we **use** Bijection, we are using some abstract n , f , and g such that ...

if we **clone** Bijection, we are getting some **new** n , f , and g such that ...

appendix B

The Essence of Hoare Logic

Hoare Logic

two main ideas

1. logical terms used in programs
2. alias-free programs allow simpler reasoning

$$\frac{}{\{P[x \leftarrow E]\} \ x := E \ \{P\}}$$

these two ideas are pushed **far** in WhyML

Logic in Programs

any logic feature is available for free in programs

- ▶ polymorphism
- ▶ algebraic data structures
- ▶ pattern matching
- ▶ predicates as booleans

reason: weakest preconditions are obviously computed

illustrated above on the “same fringe” example

Alias-free Programs

Why2 was using the traditional concept of Hoare logic

one variable = one memory location

- ▶ easy to compute effects, to exclude aliasing
- ▶ easy to compute weakest preconditions

Weakest Preconditions

include quantifications at key places

$$wp(f\ a, q) \equiv pre(f)(a) \wedge \forall \vec{x}. \forall r. post(f)(a)(r) \Rightarrow q$$

$$wp(\text{while } e \text{ do } s \text{ inv } I, q) \equiv I \wedge \forall \vec{x}. I \Rightarrow \begin{array}{l} e \Rightarrow wq(s, I) \\ \wedge \quad \neg e \Rightarrow q \end{array}$$

Limitation

say we want to model **arrays**

any solution will look like

```
type contents 'a = { | length: int; elts: map int 'a | }  
type array 'a = ref (contents 'a)  
...
```

the length is needed for array bound checking

to account for the length being immutable, we must resort to annotations $\Rightarrow$ more complex verification conditions

Regions

Why3 uses **regions** instead

one region = one memory location

regions are introduced by mutable record fields

Examples

```
type array r1 'a = { |  
    length: int;  
    mutable<r1> elts: map int 'a;  
| }
```

```
type sparse_array r1 r2 r3 r4 'a = { |  
    val : array r1 'a;  
    idx : array r2 int;  
    back : array r3 int;  
    mutable<r4> card : int;  
    default : 'a;  
| }
```

Simplicity

to make it as simple as possible (but no simpler),
regions are **implicit**

the compromise is:

- ▶ regions in arguments are distinct and already allocated
- ▶ regions in results are fresh (newly allocated)

Weakest Preconditions

effects are sets of regions

weakest preconditions quantify over regions,
then **reconstruct** the values of variables involving these regions

Example

```
let f a =  
  while ... do  
    a[i] <- 0  
  done
```

$$\forall a_0. \dots \forall r. \text{let } a_1 = \{\text{length} = a_0.\text{length}; \text{elts} = r\} \Rightarrow \dots$$

Conclusion

Summary

Why3

- ▶ a rich specification logic
- ▶ the whole chain up to provers
- ▶ easy to use and to extend (API, drivers, plug-ins)

see *Why3: Sheperd Your Herd of Provers* (BOOGIE'11)

WhyML, a programming language to be used

- ▶ to verify data structures / algorithms
- ▶ as an intermediate language (similar to Boogie/PL)

Perspectives

► Why3

- better use of TPTP provers
- Coq plug-in to call provers ($\text{Coq} \rightarrow \text{Why3} \rightarrow \text{prover}$, and back)
- improved GUI (make it a real IDE)

► WhyML

- mutable fields containing mutable data types
- ghost code
- code extraction to OCaml

Other Software Verification at ProVal

- ▶ C: collaboration with the Frama-C team (CEA)
- ▶ Java: Krakatoa (C. Marché)
- ▶ probabilistic programs (C. Paulin)
- ▶ floating-point arithmetic (S. Boldo, G. Melquiond)
- ▶ the Alt-Ergo SMT solver (S. Conchon, E. Contejean)