Deductive Program Verification with Why3

Jean-Christophe Filliâtre
CNRS

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

ProVal team, Orsay, France

ETERNAL workshop
September 28, 2011

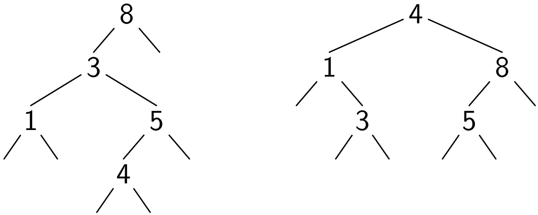
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche SACLAY - ÎLE-DE-FRANCE
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
UNIVERSITÉ PARIS-SUD 11

chapter 1

A Tale of Two Programs

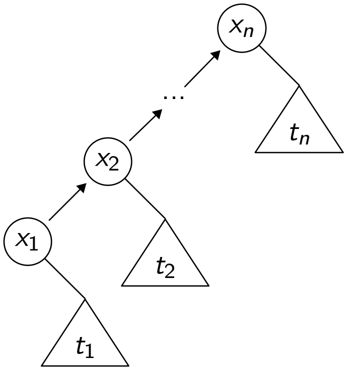
Program 1: Same Fringe

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



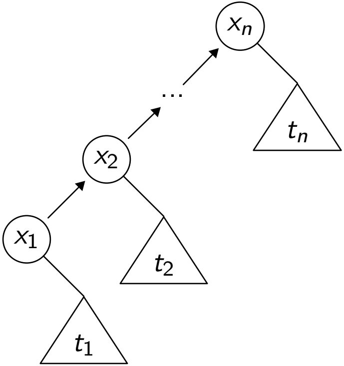
Same Fringe: A Solution

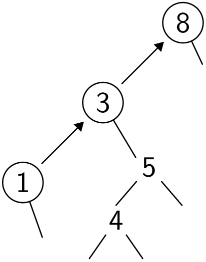
build the left spine as a bottom-up list

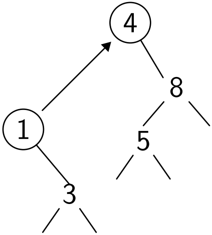


Same Fringe: A Solution

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

```
function 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

```
function enum_elements (e: enum) : list elt = match e wit
  | 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 ↔ 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

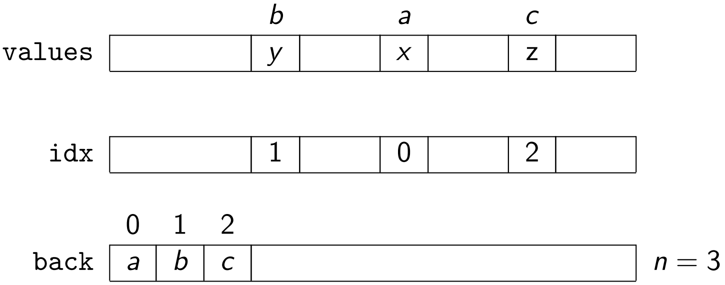
from VACID-0
Verification of Ample Correctness of Invariants of Data-structures
Rustan Leino and Michał Moskal (2010)

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



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 α = {| values  : array α;
                          idx     : array int;
                          back    : array int;
                  mutable card    : int;
                        default   : α; |}
```

note that

- field `card` is mutable
- fields `values`, `idx`, and `back` contain mutable arrays
- field `default` is immutable

Sparse Arrays

the validity of an index:

```
predicate is_elt (a: sparse_array α) (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

```
function value (a: sparse_array α) (i: int) : α =
  if is_elt a i then a.values[i] else a.default
```

Sparse Arrays

it is convenient to introduce the length of a sparse array

```
function length (a: sparse_array α) : int = A.length a.va
```

the data structure invariant:

```
predicate sa_invariant (a: sparse_array α) =
  0 ≤ a.card ≤ length a ≤ maxlen ∧
  A.length a.values = 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 α { A.length result = n }

let create (sz: int) (d: α) =
  { 0 ≤ sz ≤ maxlen }
  {| values = 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 α) 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 α) i =
  { 0 ≤ i < length a ∧ sa_invariant a }
  if test a i then
    a.values[i]
  else
    a.default
  { result = value a i }
```

Sparse Arrays

assignment

```
let set (a: sparse_array α) (i: int) (v: α) =
  { 0 ≤ i < length a ∧ sa_invariant a }
  a.values[i] ← v;
  if not (test a i) then begin
    assert { a.card < length a };
    a.idx[i] ← a.card;
    a.back[a.card] ← i;
    a.card ← a.card + 1
  end
  { sa_invariant a ∧
    value a i = v ∧
    ∀ j:int. j ≠ i → value a j = value (old a) j }
```

Sparse Arrays

one lemma proved with Coq

```
lemma permutation:
  ∀ a: sparse_array α. 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

Where Programs Meet Provers

model a program behavior in a pure logical setting (Hoare logic)

use ATP (SMT+TPTP) whenever possible, recur to ITP otherwise

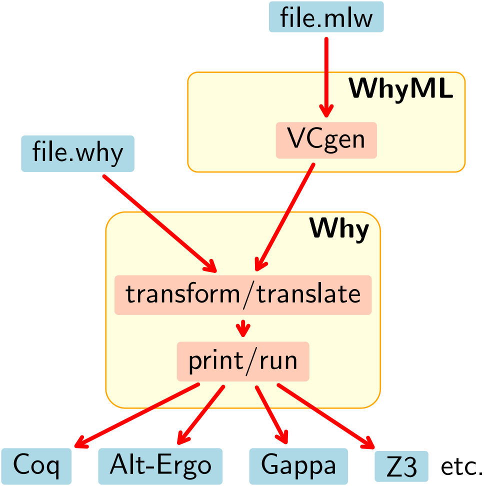
between the language of programs and that of provers lies a gap

rich type system	sorts
data structures: algebraic types, records, arrays	built-in types: integers, reals, booleans, arrays
modules and functions	flat lists of premises
higher-order	first-order

logical language as rich as possible without hindering proof search

Why3 Overview

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



chapter 3

Logical Language

Logical Language

first-order logic with polymorphic types

- ((mutually) recursive) algebraic types
- ((mutually) recursive) functions and predicates
- (mutually) inductive predicates
- let-in, match-with, if-then-else expressions
- axioms / lemmas / goals

organized in "theories"

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

Modular Specification with Theories

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 α = Nil | Cons α (list α)

  logic mem (x: α) (l: list α) = 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
- axioms from T_1 remain axioms
- lemmas from T_1 become axioms
- goals from T_1 are ignored

Example

```
theory Length

  use import List
  use import int.Int

  logic length (l: list α) : 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: ∀ l:list α. length(l) ≥ 0

end
```

Theory Cloning

a theory T_2 can clone another theory T_1

this makes a copy of T_1 with a possible instantiation
of some of its abstract symbols: types, functions, predicates

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

(similar to PVS's *theory interpretation*)

Example

```
theory OrderedUnitaryCommutativeRing
  clone export UnitaryCommutativeRing
  predicate (≤) t t
  predicate (≥) (x y : t) = y ≤ x
  clone export relations.TotalOrder with
    type t = t, predicate rel = (≤)
  axiom CompatAdd : ∀ x y z : t. x ≤ y → x + z ≤ y + z
  axiom CompatMult :
    ∀ x y z : t. x ≤ y → zero ≤ z → x * z ≤ y * z
end
theory Int
  function zero : int = 0
  function one : int = 1
  predicate (<) int int
  predicate (>) (x y : int) = y < x
  predicate (≤) (x y : int) = x < y ∨ x = y
  clone export algebra.OrderedUnitaryCommutativeRing with
    type t = int, function zero = zero,
    function one = one, predicate (≤) = (≤)
end
```

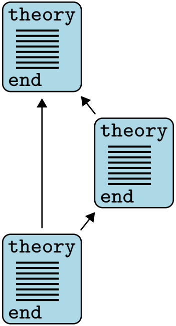
Theory Cloning

a theory T_2 can clone another theory T_1

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

a cloned axiom becomes a lemma/goal when we implement an abstract symbol and want to verify our implementation

From Theories to Provers

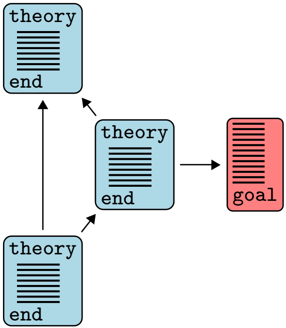


Alt-Ergo

Z3

Vampire

From Theories to Provers

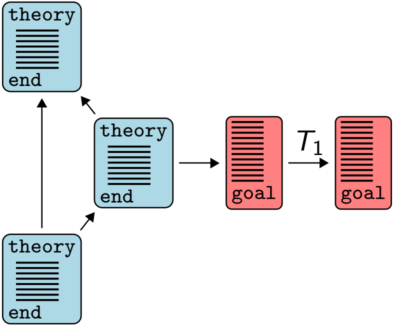


Alt-Ergo

Z3

Vampire

From Theories to Provers

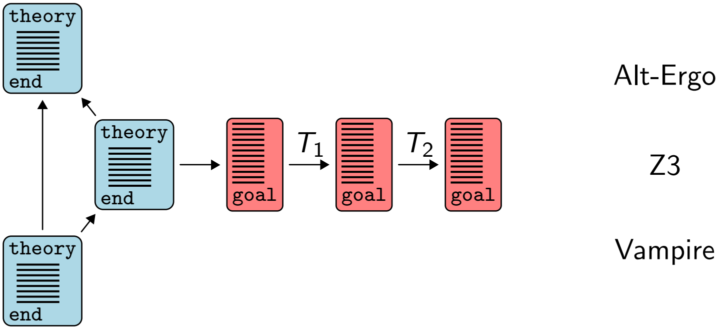


Alt-Ergo

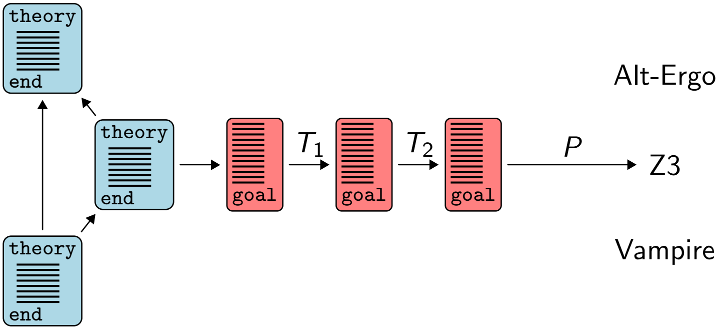
Z3

Vampire

From Theories to Provers



From Theories to Provers



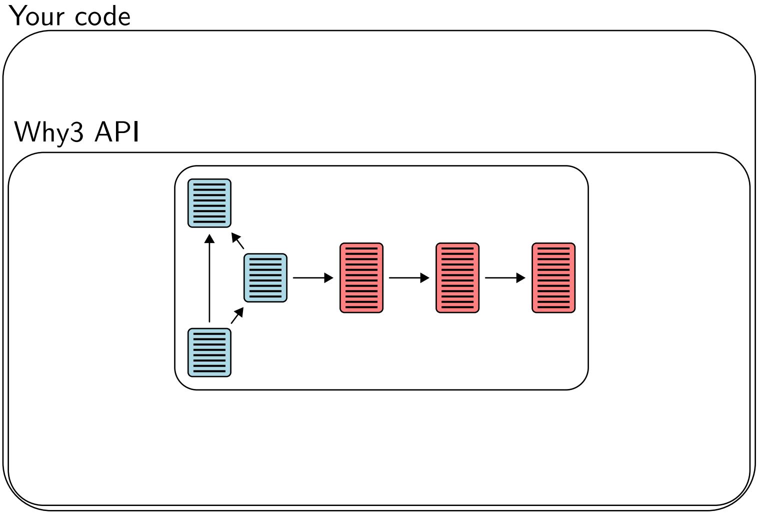
Prover Drivers

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

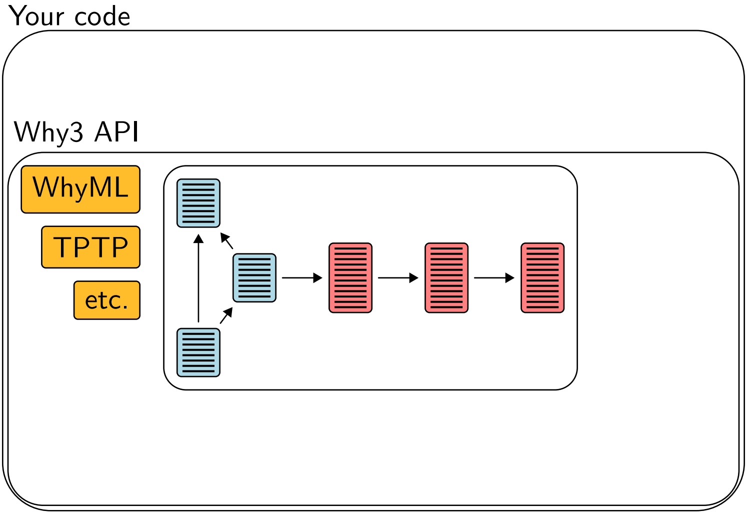
Example: Z3 driver (excerpt)

```
printer "smtv2"
valid "^unsat"
invalid "^sat"

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic_smt"
transformation "simplify_formula"
transformation "discriminate"
transformation "encoding_smt"

prelude "(set-logic AUFNIRA)"

theory BuiltIn
   syntax type int "Int"
   syntax type real "Real"
   syntax predicate (=) "(= %1 %2)"

   meta "encoding : kept" type int
end
```

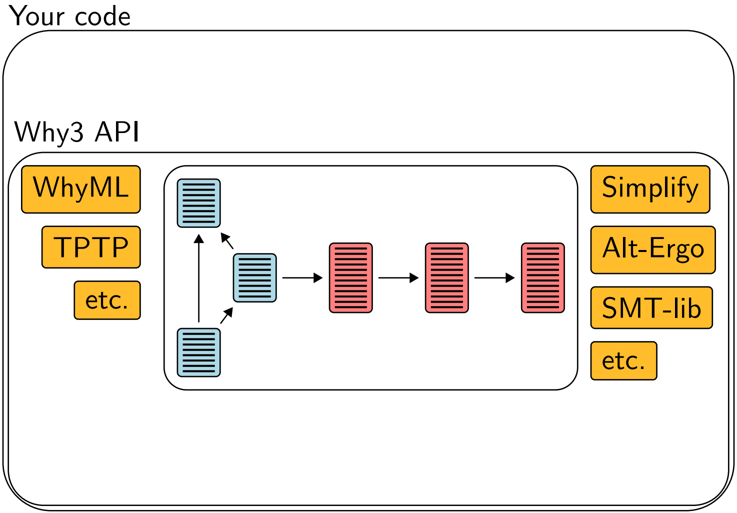
Architecture: API and Plug-ins



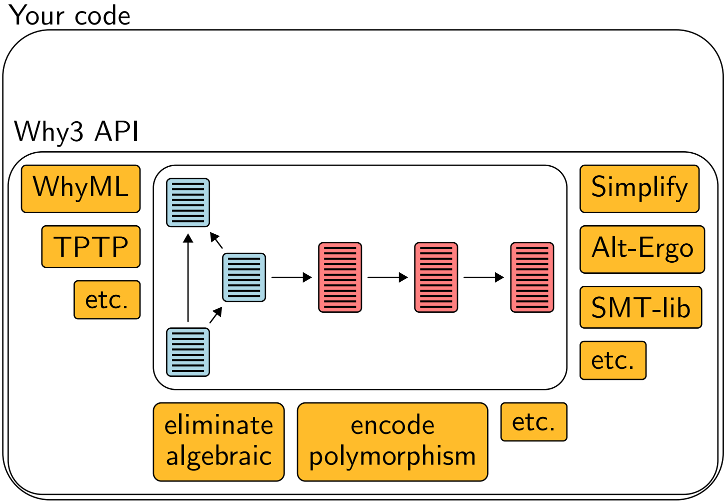
Architecture: API and Plug-ins



Architecture: API and Plug-ins



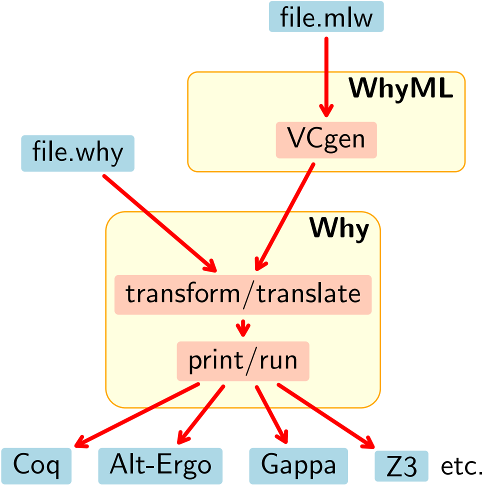
Architecture: API and Plug-ins



chapter 4

A Programming Language: WhyML

Why3 Overview



A Programming Language: WhyML

- ML-inspired (syntax, polymorphism, algebraic data types, local functions), yet first-order
- effects: mutable data structures, exceptions, non-termination
- annotations (pre/post/assert/loop invariants)
- a notion of modules, analogous to theories
- encapsulation

```
type int32 model int

type array α model {| length : int;
                       mutable elts : map int α |}
```

Two Ideas from Hoare Logic

1. logical types and terms used in programs
2. alias-free programs, to allow simpler reasoning

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

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

$$\text{one variable} = \text{one memory location}$$

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

Weakest Preconditions

include quantifications at key places

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

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

Limitation

say we want to model arrays

any solution will look like

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

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 α = {|
            length: int;
  mutable<r1> elts: map int α;
|}

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

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
```

$$\forall a_0. \ldots . \forall r. \texttt{let } a_1 = \{\texttt{length} = a_0.\texttt{length}; \texttt{elts} = r\} \Rightarrow \ldots$$

Conclusion

Summary

Why3

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

Why3: Sheperd Your Herd of Provers (BOOGIE'11)
Expressing Polymorphic Types in a Many-Sorted Language (FroCos'11)

WhyML, a programming language to be used

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

`http://why3.lri.fr`

Perspectives

- Why3
  - better use of TPTP provers
  - Coq plug-in to call provers (Coq $\rightarrow$ Why3 $\rightarrow$ prover, and back)
- WhyML
  - data type invariants
  - ghost code
  - module cloning
  - 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)