Deductive Program Verification

Jean-Christophe Filliâtre
CNRS

ITP 2018

Oxford, UK
July 12, 2018

joint work with

François Bobot

Claude Marché

Guillaume Melquiond

Andrei Paskevich

a question for programmers

shall I be pure or impure?

a question for program verifiers

shall I be pure or impure?

a question for program verifiers

shall I be pure or impure?

FP ⟷ mutability feast

a question for program verifiers

shall I be pure or impure?

a question for program verifiers

shall I be pure or impure?

FP

the While language

WhyML

mutability feast

WhyML

goal

no model of the heap
to get simpler VCs

solution

records with mutable fields
+
static control of aliases

mutable variables aka references

```
type ref 'a = {
  mutable contents: 'a;
}
```

we can model some data structures

e.g. arrays

```
type array 'a = private {
  mutable ghost elts: int -> 'a;
                length: int;
}
```

we can nest mutable types

e.g. a heap in a resizeable array

```
type heap = {
  mutable       data: array elt;
  mutable       size: int;
  mutable ghost view: bag elt;
}
```

the type checker is powerful enough to let you replace
the `data` field while keeping track of aliases [ESOP 2013]

the key is abstraction

there are mutable DS you cannot implement
(e.g. linked lists, mutable trees)

yet you can model them easily

then you can verify client code, thanks to proof modularity

example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

example: union-find

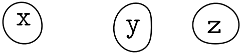
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



example: union-find

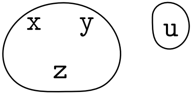
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

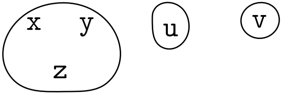
example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



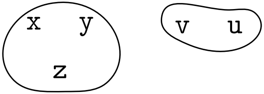
example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



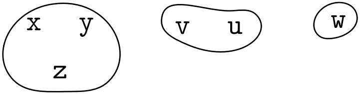
example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



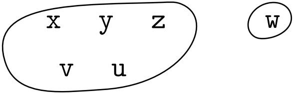
example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



example: union-find

```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



example: union-find

```
type elem
```

```
type uf = {
  mutable dom: set elem;
  mutable rep: elem -> elem;
}
```

```
val ghost create () : uf
val make  (ghost uf: uf) ()            : elem
val union (ghost uf: uf) (x y: elem) : unit
val find  (ghost uf: uf) (x   : elem) : elem
val same  (ghost uf: uf) (x y: elem) : bool
```

what else

WhyML features

- polymorphism
- algebraic data types, pattern matching
- exceptions, `break`, `continue`, `return`

- ghost code and ghost data [CAV 2014]

- contracts, loop and type invariants
- VCGen = either traditional or Flanagan/Saxe style WP

a logic for program verification

goal

rich enough to make your life easier,
simple enough to be sent to ATPs

a logic for program verification

goal

rich enough to make your life easier,
simple enough to be sent to ATPs

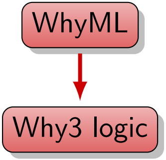
our solution

a total, polymorphic first-order logic with

- algebraic types & pattern matching
- recursive definitions
- (co)inductive predicates
- mapping type $\alpha \to \beta$, λ-notation, application

[FroCos 2011, CADE 2013, VSTTE 2014]

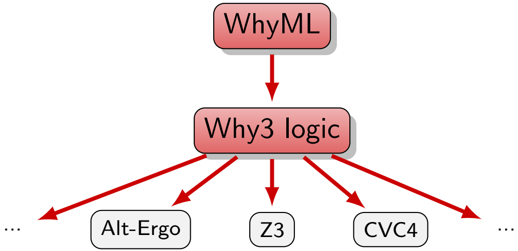
Why3



WhyML

Why3 logic

Why3



WhyML

Why3 logic

…

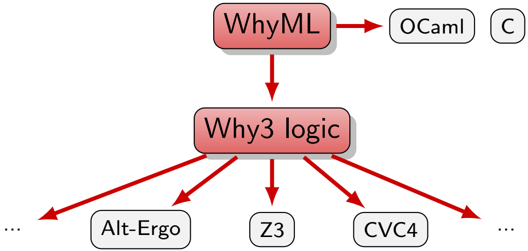
Alt-Ergo

Z3

CVC4

…

Why3



WhyML

OCaml

C

Why3 logic

…

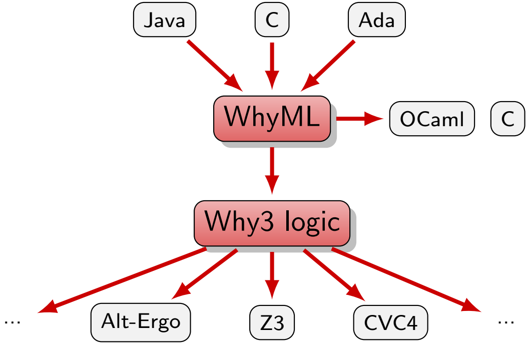
Alt-Ergo

Z3

CVC4

…

Why3



Java

C

Ada

WhyML

OCaml

C

Why3 logic

...

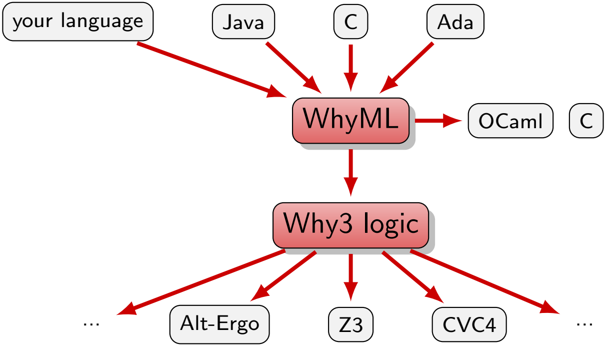
Alt-Ergo

Z3

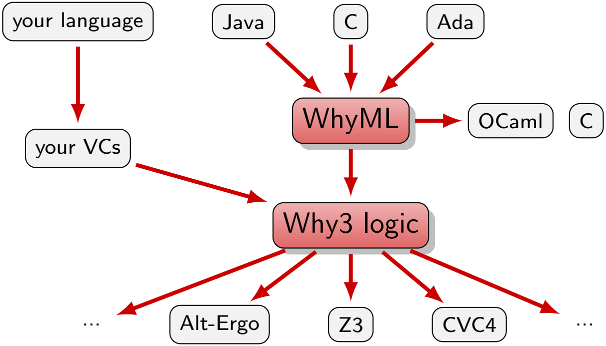
CVC4

...

Why3


your language
Java
C
Ada
WhyML
OCaml
C
Why3 logic
...
Alt-Ergo
Z3
CVC4
...

Why3


your language
your VCs
Java
C
Ada
WhyML
OCaml
C
Why3 logic
…
Alt-Ergo
Z3
CVC4
…

using off-the-shelf provers

Why3 currently supports 25+ ITPs and ATPs

for each prover, a special “driver” file controls [Boogie 2011]

- logical transformations to apply
- input/output format
- predefined symbols, axioms to be removed

example: Z3 driver

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

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic"
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)"
end
...
```

$$\frac{\text{demo}}{\text{union-find}}$$

joint work with S. Melo de Sousa, M. Pereira, and M. Clochard

API

```
type elem
type uf = ...
val ghost create () : uf
val make  (ghost uf: uf) ()           : elem
val union (ghost uf: uf) (x y: elem) : unit
val find  (ghost uf: uf) (x  : elem) : elem
val same  (ghost uf: uf) (x y: elem) : bool
```

specification

```
type elem
```

```
type uf = {
  mutable dom: set elem;
  mutable rep: elem -> elem;
}
invariant { forall x. mem x dom ->
            mem (rep x) dom && rep (rep x) = rep x }
```

```
val ghost create () : uf
  ensures { result.dom = empty }
```

```
val make (ghost uf: uf) () : elem
  writes  { uf.dom, uf.rep }
  ensures { not (mem result (old uf.dom)) }
  ensures { uf.dom = add result (old uf.dom) }
  ensures { uf.rep = (old uf.rep)[result <- result] }
```

```
val find (ghost uf: uf) (x: elem) : elem
  requires { mem x uf.dom }
  ensures  { result = uf.rep x }
```

```
val union (ghost uf: uf) (x y: elem) : ghost elem
  requires { mem x uf.dom }
  requires { mem y uf.dom }
  writes   { uf.rep }
  ensures  { result = old (uf.rep x) ||
             result = old (uf.rep y) }
  ensures  { forall z. mem z uf.dom ->
    uf.rep z = if old (uf.rep z = uf.rep x ||
                       uf.rep z = uf.rep y)
               then result
               else old (uf.rep z) }
```

implementation

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

implementation

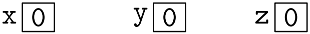
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

implementation

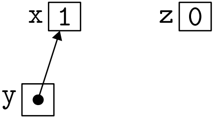
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

implementation

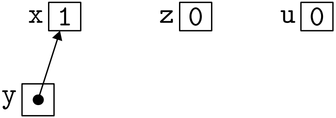
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

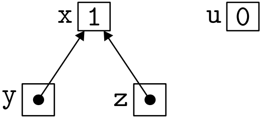


```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



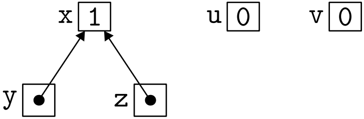
implementation

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



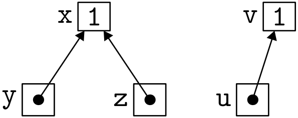
implementation

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

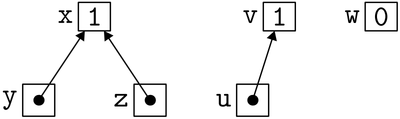


implementation

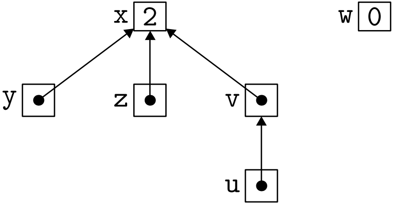
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

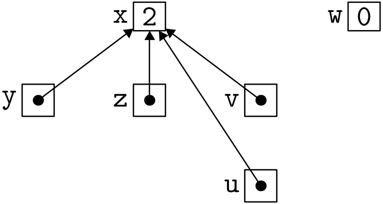


```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



implementation

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



let's verify this with Why3

Why3 implementation

too complex for Why3's type checker; let's model the heap

```
type loc
```

```
type elem =
  content ref
and content =
  | Link of elem
  | Root of int
```

```
type elem =
  loc
type content =
| Link loc
| Root Peano.t
```

```
type heap = {
  ghost mutable
    refs: loc -> option content;
}
```

termination

termination

it would be very tempting to introduce an inductive notion of path

```
inductive path (h: heap) (x y: elem) =
| Path0: forall x y k.
         h.refs x = Some (Root k) ->
         path h x x
| Path1: forall x y z.
         h.refs x = Some (Link y) ->
         path h y z -> path h x z
```

this way, we would have `path heap x (rep x)` as an invariant and this would ensure the termination of `find`

termination

but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

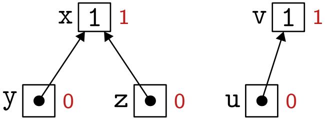
- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

$\texttt{maxd} = 1$

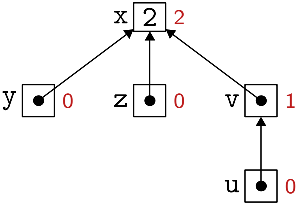


but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

`maxd` $= 2$



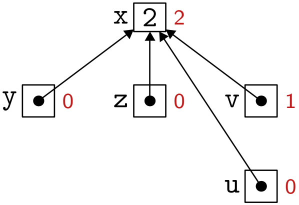
termination

but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

$\texttt{maxd} = 2$



extraction to OCaml

Why3 extraction mechanism

1. removes ghost code
2. maps some Why3 symbols to OCaml symbols

here

- type `Peano.t` is mapped to OCaml's type `int`
- our custom mini-heap is mapped to OCaml's references

related work

Charguéraud & Pottier did a Coq proof of a similar OCaml code, using CFML [ITP 2015, JAR 2017]

- includes a proof of complexity!
- maps OCaml's type `int` to Coq's type `Z` (unsound)
- more than 4k lines

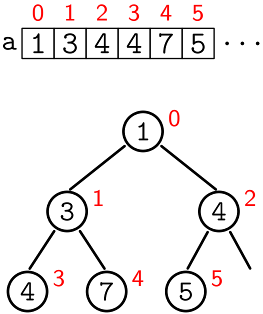
lessons learned

1. modeling the heap can be easy
    - can be local
    - incurs a small TCB

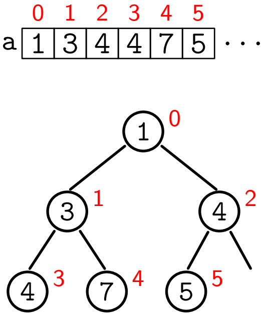
2. avoid recursive/inductive definitions for better automation

   two other examples:
    - heap stored in an array
    - inverting a permutation in-place

heap stored in an array



heap stored in an array



it would be tempting to introduce trees

but a universal, local invariant

$$\forall i.\ \mathtt{a}[i] \le \mathtt{a}[2i+1], \mathtt{a}[2i+2]$$

is all you need

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	0	1	5	2

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	0	1	5	-1

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	-6	1	5	-1

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	5	-1

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	-1	-1

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	-1	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	-1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	-6	-1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	-6	1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	5	1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	5	1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

2	3	5	1	0	4

inverting a permutation in-place

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

2	3	5	1	0	4

again it would tempting to introduce paths, orbits, cycles, etc.

but again a universal, local invariant suffices

many other things about Why3

- Why3+Alt-Ergo in your browser [`http://why3.lri.fr/try/`]
- Python frontend for teaching purposes
- Why3's OCaml API [BOOGIE 2011]
- proof by reflection [VSTTE 2016]
  including imperative programs [IJCAR 2018, next Sunday]
- extraction to C
- logical connectives `by` and `so` to encode proofs
- floating-point arithmetic [ARITH 2007]
- checking the consistency of our library using Coq
- preserving proofs across changes [VSTTE 2013]

visit our gallery

`http://toccata.lri.fr/gallery/why3.en.html`

more than 150 verified programs

- data structures: AVL/red-black trees, Fenwick trees, ropes, skew/binomial/pairing/Braun/leftist heaps, etc.
- algorithms: algorithm I, Tortoise and Hare, sorting, graph, etc.
- solutions to many competitions/challenges (e.g. VerifyThis)