Why3
where programs meet provers

Jean-Christophe Filliâtre
CNRS

KeY Symposium 2017

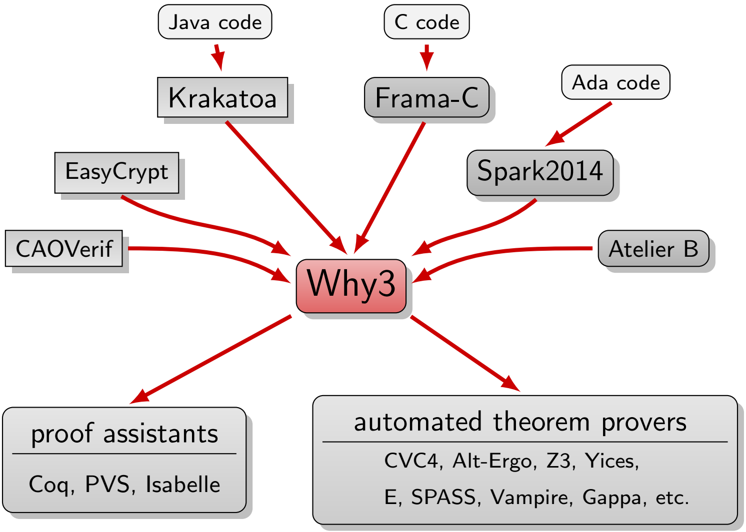
Rastatt, Germany
October 5, 2017

history

started in 2001, as an intermediate language in the process of verifying C and Java programs ($\sim$ Boogie)

today, joint work with

- François Bobot (CEA)
- Claude Marché (Inria)
- Guillaume Melquiond (Inria)
- Andrei Paskevich (Univ Paris Sud)


Java code
C code
Krakatoa
Frama-C
Ada code
EasyCrypt
Spark2014
CAOVerif
Why3
Atelier B
proof assistants
Coq, PVS, Isabelle
automated theorem provers
CVC4, Alt-Ergo, Z3, Yices,
E, SPASS, Vampire, Gappa, etc.

and a language of its own

Why3 features a full-fledged programming language

programs can be translated to OCaml automatically

contributions

Why3 uses well-known techniques (e.g. weakest preconditions) and off-the-shelf provers

most our R&D is focused on the design of a logic, a programming language, and a library dedicated to program verification

designing a logic

designing a logic

a total, polymorphic first-order logic, with

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

[FroCos 2011, CADE 2013, VSTTE 2014]

- polymorphic types

```
type set 'a
```

- tuples

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

- records

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

- sums

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

- access to record fields

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

- record reconstruction

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

pattern matching and recursive definition

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

termination is checked automatically

```
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))
```

(the smallest predicate satisfying these three axioms)

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

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

  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

  ...
```

designing a programming language

designing a programming language

$\sim$ small subset of OCaml

- polymorphism
- pattern matching
- exceptions
- mutable data with controlled aliasing [ESOP 2013]
- ghost code and ghost data [CAV 2014]
- contracts, loop and type invariants

```
let rec f91 (n: int) : int
  ensures { result = if n <= 100 then 91 else n - 10 }
  variant { 101 - n }
= if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

ghost code/data eases the specification and the proof

ghost code should not interfere with regular code

- regular code cannot see ghost data
- ghost code cannot mutate regular data
- ghost code cannot raise exceptions
- ghost code must terminate

the system checks the non-interference [CAV 2014]

- function parameters

```
let f (a b n: int) (ghost k: int): int = ...
```

- record fields

```
type queue 'a = { head: list 'a;
                  tail: list 'a;
            ghost elts: list 'a; }
invariant { elts = head ++ reverse tail }
```

- variables and functions

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

- program expressions

```
let x = ghost q.head in ...
```

idea: a ghost function

$$\texttt{f}\ \vec{x}\ \texttt{requires}\ P\ \texttt{ensures}\ Q$$

with no side effect and terminating
is a constructive proof of

$$\forall \vec{x}.\, P \Rightarrow Q$$

you have defined

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

and you want to prove

$\forall$ `l r. length (rev_append l r) = length l + length r`

this requires induction

solution: a recursive lemma function

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

- you prove it correct, as with any function
- the lemma ∀`l r`... is added to the context
- still available as a ghost function, to be called explicitly

record fields can be declared mutable

e.g. OCaml's mutable variables, aka references

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

mutable records

- can be passed to functions and returned
- can be created locally and declared globally
  - `let r = ref 0 in while !r < 42 do ...`
  - `val gr: ref int`
- can store ghost data
  - `let ghost r = ref 42 in ...`
- can be nested
  but are disallowed in recursive types (i.e. no `list (ref 'a)`)
  and abstract types (i.e. no `set (ref 'a)`)

aliases

Why3 keeps track of all aliases, statically,
using a type system with effects

motivation: keep using traditional WP,
without resorting to a memory model

consequence: some programs are rejected by the type checker

designing a library

includes

- integers, real numbers, sets, maps, sequences
- option type, lists, binary trees
- higher-order operators, e.g.

$$\texttt{sum f a b} \;=\; \sum_{\texttt{a} \le i < \texttt{b}} \texttt{f}(i)$$

$$\texttt{numof p a b} \;=\; \mathrm{card}\{i \mid \texttt{a} \le i < \texttt{b} \wedge \texttt{p}(i)\}$$

a programming library

includes

- references, arrays, stacks, queues
- floating-point arithmetic [ARITH 2007]
- machine integers
  - how to prove the absence of overflows [VSTTE 2015]

demo

maximum subarray problem

given an array of integers,
find the contiguous subarray with the largest sum

example:

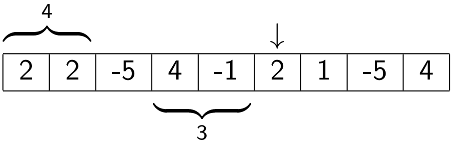
2	2	-5	4	-1	2	1	-5	4

6

note: zero-length subarrays are allowed,
and maximal when all elements are negative

there is a nice, linear time, constant space solution due to Jay Kadane (1977)
(the whole story can be found in Jon Bentley's *Programming Pearls*)

idea: scan the array, maintaining both the maximum so far and the maximum ending at the scanning position



gallery of verified programs

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

more than 130 examples

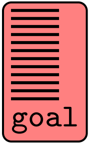
- data structures: AVL, red-black trees, skew heaps, Braun trees, ropes, resizable arrays, etc.
- sorting, graph algorithms, etc.
- solutions to most competition problems (VSComp, VerifyThis)

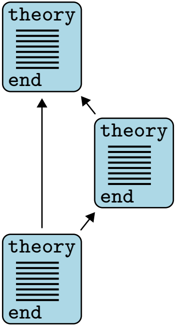
under the hood

a technology to talk to provers

central concept: task

- a context (a list of declarations)
- a goal (a formula)

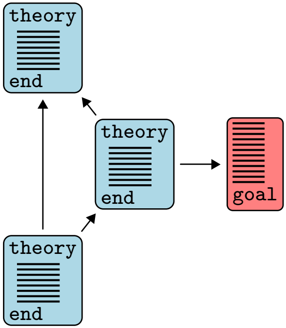




Alt-Ergo

Z3

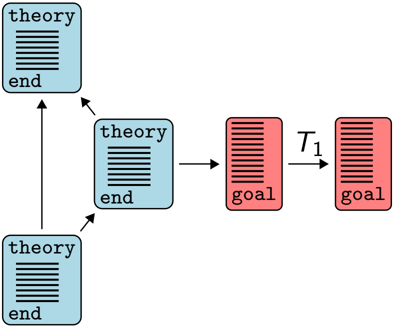
Vampire



Alt-Ergo

Z3

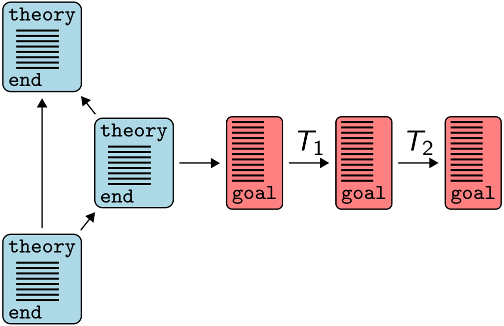
Vampire



Alt-Ergo

Z3

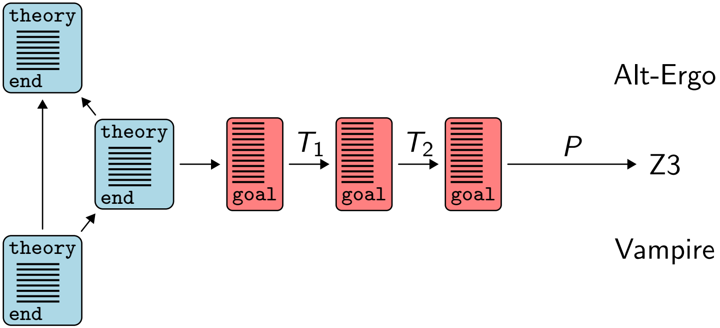
Vampire



Alt-Ergo

Z3

Vampire


theory
end
theory
end
theory
end
goal
T1
goal
T2
goal
P
Alt-Ergo
Z3
Vampire

logical transformations

- eliminate algebraic data types and match-with
- eliminate inductive predicates
- eliminate if-then-else, let-in
- encode polymorphism, encode types
- etc.

efficient: results of transformations are memoized a task journey is driven by a file

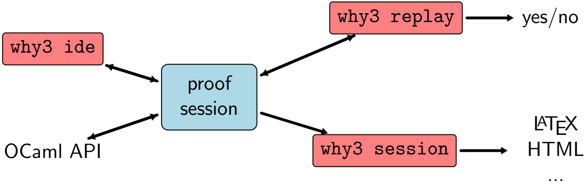
- transformations to apply
- prover's input format
  - syntax
  - predefined symbols
  - axioms to be removed
- prover's diagnostic messages

more details: *Why3: Shepherd your herd of provers* [Boogie 2011]

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"
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
...
```

proofs are stored into an XML file and read/written/updated by various tools



more details:
Preserving User Proofs Across Specification Changes [VSTTE 2013]

and many other things

- running Why3+Alt-Ergo in your browser
- Python frontend for teaching purposes
- Why3's OCaml API [BOOGIE 2011]
- proof by reflection [VSTTE 2016]
- logical connectives `by` and `so`
- checking the consistency of our library using Coq
- extraction to C