Deductive Program Verification with Why3

Jean-Christophe Filliâtre  
CNRS

MSR-Inria Joint Centre “Spring Day”  
May 21, 2013
deductive program verification

Tony Hoare.
An Axiomatic Basis for Computer Programming.
which programs? which specs?

program + specification \rightarrow verification conditions \rightarrow proof

programs
- pseudo code / mainstream languages / DSL
- small / large

specs
- safety, i.e. the program does not crash
- absence of arithmetic overflow
- complex behavioral property, e.g. “sorts an array”
which logic?

- too rich: we won’t be able to automate proofs
- too poor: we can’t model programming languages and we can’t specify programs

typically, a compromise
- e.g. first-order logic + equality + arithmetic
what about proofs?

A gift: theorem provers

- proof assistants: Coq, PVS, Isabelle, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- SMT solvers: CVC3, Z3, Yices, Alt-Ergo, etc.
- dedicated provers
checking a large routine (Turing, 1949)

\[
\begin{align*}
  r' &= 1 \\
  u' &= 1 \\
  v' &= u \\
  s' &= 1 \\
  u' &= u + v \\
  s' &= s + 1 \\
  r' &= r + 1 \\
  \text{TEST } s - r \\
  \text{TEST } r - n \\
  \text{STOP}
\end{align*}
\]
$u \leftarrow 1$

for $r = 0$ to $n - 1$ do

$v \leftarrow u$

for $s = 1$ to $r$ do

$u \leftarrow u + v$

checking a large routine (Turing, 1949)
requires $\{n \geq 0\}$

$u \leftarrow 1$

for $r = 0$ to $n - 1$ do

$v \leftarrow u$

for $s = 1$ to $r$ do

$u \leftarrow u + v$

ensures $\{u = \text{fact}(n)\}$
checking a large routine (Turing, 1949)

```
\begin{align*}
    r' &= 1 \\
    u' &= 1 \\
    v' &= u \\
    s' &= 1 \\
    u' &= u + v \\
    s' &= s + 1 \\
\end{align*}
```

\begin{itemize}
\item requires \( \{ n \geq 0 \} \)
\item \( u \leftarrow 1 \)
\item \text{for } r = 0 \text{ to } n - 1 \text{ do}
\item \hspace{1em} \text{invariant } \{ u = \text{fact}(r) \} \\
\item \hspace{1em} v \leftarrow u \\
\item \hspace{1em} \text{for } s = 1 \text{ to } r \text{ do}
\item \hspace{2em} \text{invariant } \{ u = s \times \text{fact}(r) \} \\
\item \hspace{2em} u \leftarrow u + v \\
\item \text{ensures } \{ u = \text{fact}(n) \}
\end{itemize}
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
            (u1 = (r + 1) * fact(r) → u1 = fact(r + 1)))))) ∧
    (u = fact((n - 1) + 1) → u = fact(n))))
function fact(int) : int
taxiom fact0: fact(0) = 1

goal vc: ∀ n:int. n ≥ 0 →
   (0 > n - 1 → 1 = fact(n)) ∧
function fact(int) : int

axiom factn: \( \forall n: \text{int}. \ n \geq 1 \rightarrow \text{fact}(n) = n \times \text{fact}(n-1) \)
goal vc: \( \forall n: \text{int}. \ n \geq 0 \rightarrow \)

\[ (0 \leq n - 1 \rightarrow \]

\[ (\forall u: \text{int}. \]
\[ (\forall r: \text{int}. \ 0 \leq r \land r \leq n - 1 \rightarrow u = \text{fact}(r) \rightarrow \]

\[ (1 \leq r \rightarrow \]

\[ (\forall u1: \text{int}. \]

\[ (u1 = (r + 1) \times \text{fact}(r) \rightarrow u1 = \text{fact}(r + 1)))))) \land \]
SMT means Satisfiability Modulo Theories

an SMT solver combines

$$\forall + \text{SAT} + \text{Equality} + \text{Arith} + \ldots$$

e.g.

$$\begin{align*}
\frac{n \geq 0}{n = 0} (\text{Arith}) & \quad \frac{0 > n - 1}{\text{fact}(0) = 1} (\text{Ax}) \\
\frac{0 > n - 1}{1 = \text{fact}(n)} (\text{Equality})
\end{align*}$$
extracting verification conditions

a well-known technique
(weakest preconditions, Dijkstra 1971, Barnett/Leino 2005)

yet doing it for a realistic programming language is a lot of work
extracting verification conditions

a well-known technique
(weakest preconditions, Dijkstra 1971, Barnett/Leino 2005)

yet doing it for a realistic programming language is a lot of work

as in a compiler, we rather design an intermediate language from which we extract VCs

two examples:
• Boogie (Microsoft Research)
• Why3 (Univ. Paris Sud / Inria)
Why3 in a nutshell

- a **programming language**, with
  - polymorphism
  - pattern-matching
  - exceptions
  - mutable data structures, with controlled aliasing

- a **polymorphic first-order logic**, with
  - algebraic data types
  - recursive definitions
  - inductive and coinductive predicates

http://why3.lri.fr/
1. a taste of program verification
Bresenham’s line drawing algorithm
Bresenham’s line drawing algorithm

assuming we are in the first octant
i.e. $0 \leq y_2 \leq x_2$

\[ e \leftarrow 2y_2 - x_2 \]
\[ y \leftarrow 0 \]
\[ \text{for } x = 0 \text{ to } x_2 \]
\[ \quad \text{plot } (x, y) \]
\[ \quad \text{if } e < 0 \]
\[ \quad \quad e \leftarrow e + 2y_2 \]
\[ \quad \text{else} \]
\[ \quad \quad y \leftarrow y + 1 \]
\[ \quad \quad e \leftarrow 2(y_2 - x_2) \]
Bresenham’s algorithm explained

\[ \frac{e}{2x} = (x + 1) \frac{y}{x} - y - \frac{1}{2} \]
Bresenham’s algorithm explained

\[ e = 2(x + 1)y_2 - (2y + 1)x_2 \]
demo
Bresenham’s algorithm explained

\[ \frac{y_2}{x_2} - 1 \leq \frac{e}{2x_2} \leq \frac{y_2}{x_2} \]
Bresenham’s algorithm explained

\[ 2(y_2 - x_2) \leq e \leq 2y_2 \]
2. one logic to use them all
there are many theorem provers

- SMT solvers: Alt-Ergo, Z3, CVC3, Yices, etc.
- TPTP provers: Vampire, Eprover, SPASS, etc.
- proof assistants: Coq, PVS, Isabelle, etc.
- dedicated provers, e.g. Gappa

we want to use all of them if possible

we make a compromise
logic of Why3 = **polymorphic first-order logic**, with

- (mutually) recursive algebraic data types
- (mutually) recursive function/predicate symbols
- (mutually) inductive predicates
- let-in, match-with, if-then-else

more details:

*Expressing Polymorphic Types in a Many-Sorted Language* *(FroCos 2011)*

*One Logic To Use Them All* *(CADE 2013)*
• types
  • abstract: type t
  • alias: type t = list int
  • algebraic: type list α = Nil | Cons α (list α)

• function / predicate
  • uninterpreted: function f int : int
  • defined: predicate non_empty (l: list α) = l ≠ Nil

• inductive predicate
  • inductive trans t t = ...

• axiom / lemma / goal
  • goal G: ∀ x: int. x ≥ 0 → x*x ≥ 0
logic declarations organized in theories

- theories can be reused
  - e.g. the theory of integers

- generic theories can be instantiated
  - e.g. notion of path
a technology to talk to provers

central concept: task
  • a context (a list of declarations)
  • a goal (a formula)
theory
end

theory
end

theory
end

Alt-Ergo

Z3

Vampire
workflow

theory
end

theory
end

goal

Alt-Ergo

Z3

Vampire
Alt-Ergo
Z3
Vampire
The workflow diagram shows a sequence of theories and goals. The theories are labeled 'theory' and 'end', and the goals are represented by 'goal'. The transitions are labeled as $T_1$ and $T_2$. The tools mentioned on the right are Alt-Ergo, Z3, and Vampire.
Workflow theory end theory end theory end theory end goal goal goal goal Alt-Ergo Vampire Z3
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
- 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
proof sessions

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

- why3ide
- OCaml API
- why3session
- why3replayer

More details:
Preserving User Proofs Across Specification Changes (VSTTE 2013)
3. to do what?
Why3 as a front-end to many theorem provers

- saves you a lot of work
- currently supports
  - Coq 8.4, PVS 6.0
  - Alt-Ergo 0.95.1, CVC3 2.4.1, CVC4 1.0, Simplify 1.5.4, Yices 1.0.38, Yices2 2.0.4, Z3 4.3.1
  - E 1.6, iProver 0.8.1, SPASS 3.7, Vampire 0.6, Zenon 0.7.1
  - Gappa 0.15.1, Mathematica 8.0
- through files or via the OCaml API
Why3 comes with a Coq tactic to call external ATPs as oracles

Coq

let why3 p gl =
...

Why3

application: proof of lemma closest in Bresenham’s demo uses 18 calls to Alt-Ergo
Why3 as a programming language to prove algorithms

- more than 80 examples in our gallery, e.g.
  - Knuth-Morris-Pratt algorithm
  - Bellman-Ford algorithm
  - Red-black trees
  - etc.

- automatically translated to OCaml code
Why3 as an intermediate language, to verify programs written in other, more complex programming languages

- Java programs: Krakatoa (Marché Paulin Urbain)
- C programs: Frama-C (Marché Moy)
- Ada programs: GNATprove (Adacore)
- probabilistic programs: EasyCrypt (Barthe et al.)
some systems using Why3

- Ada → GNATprove
- Java → Krakatoa
- C → Frama-C
  - Jessie
  - WP
- prob. → Easycrypt

Why3

- WhyML
- logic

- proof assistants
- SMT solvers
- TPTP provers
- other provers
WhyML imposes a static **control of aliasing**

*why?* to get simpler verification conditions

*how?* using regions (internally)

more details:

*Why3 — Where Program Meet Provers* (ESOP 2013)
to use Why3 to verify programs with aliasing, you have to come up with a memory model

```
  type loc
  type value = ...
  type state = map loc value
  ...
```

this is what is done for C, Java, Ada, etc.
conclusion
to simultaneously

• increase proof automation
  • e.g. automatic induction

• enrich the specification logic
  • e.g. higher-order logic

• support more programming constructs
  • e.g. continuations, coroutines, higher-order functions
• LGPL software
• related publications
• more than 80 examples
• documentation, lecture notes