Modular certification of Why3

Deducteam seminar

July 2, 2020

Quentin Garchery
The instantiate example in Why3

Program with annotations [ Dailler, Marché, FIDE 2018 ]:

```ocaml
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { result ≥ 0 }
  = let y = 2*x+1 in a[y*y]
```

To prove:

```ocaml
forall a:array int, x:int.
length a ≥ 1000 ∧ 0 ≤ x ≤ 10 →
(forall i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0) →
let y = 2 * x + 1 in (0 ≤ y * y < length a) ∧ a[y * y] ≥ 0
```
The instantiate example in Why3

Program with annotations [ Dailler, Marché, FIDE 2018 ]:

```plaintext
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { result ≥ 0 }
  = let y = 2*x+1 in a[y*y]
```

Transform what there is to prove (with `split_vc`):

- **Req1**: length a ≥ 1000 ∧ 0 ≤ x ≤ 10
- **Req2**: for all i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0
- **y**: int = 2 * x + 1

---

**goal** AccessInBounds : 0 ≤ (y * y) < length a

---

**goal** Postcondition : a[y * y] ≥ 0
The instantiate example in Why3

Program with annotations [ Dailler, Marché, FIDE 2018 ]:

let f (a:array int) (x:int) : int
  requires { a.length \geq 1000 \land 0 \leq x \leq 10 }
  requires { \forall i. 0 \leq 4i+1 < a.length \rightarrow a[4i+1] \geq 0 }
  ensures { result \geq 0 }
  = let y = 2x+1 in a[y*y]

Transform postcondition (with instantiate):

Req1 : length a \geq 1000 \land 0 \leq x \leq 10
Req2 : \forall i:int. 0 \leq 4 \times i + 1 < length a \rightarrow a[4 \times i + 1] \geq 0
y : int = 2 \times x + 1
Hinst : 0 \leq 4 \times (x \times x + x) + 1 < length a \rightarrow a[4 \times (x \times x + x) + 1] \geq 0
goal Postcondition : a[y \times y] \geq 0
Overview of Why3

Why3 step by step:

- Initial task generation
- Logical transformations
- Call to automatic theorem provers
Overview of Why3

Why3 step by step:

- initial task generation
- logical transformations
- call to automatic theorem provers

annotated program $T$ (for Why3)
Overview of Why3

Why3 step by step:

- initial task generation
- logical transformations
- call to automatic theorem provers
Overview of Why3

Why3 step by step:

- initial task generation
- logical transformations
- call to automatic theorem provers

annotated program

Why3

$T \xrightarrow{\phi} T_1 \xrightarrow{} Z3$
$T \xrightarrow{\phi} T_2 \xrightarrow{} Alt-Ergo$
$T \xrightarrow{\phi} T_3 \xrightarrow{} CVC4$
Overview of Why3

Why3 step by step:
- Initial task generation
- Logical transformations
- Call to automatic theorem provers

annotated program

Why3

\( T \) ➔ \( T \phi \) ➔ \( T_1 \) ➔ \( T_1' \) ➔ Z3

\( T_1 \) ➔ Z3 driver

\( T_2 \) ➔ \( T_2' \) ➔ Alt-Ergo

\( T_2 \) ➔ AErgo driver

\( T_3 \) ➔ \( T_3' \) ➔ CVC4

\( T_3 \) ➔ CVC4 driver
Overview

How to improve trust in Why3

Reduce trust base by:

- isolating a kernel
- a posteriori certification

Emphasis on the modular and incremental approach
<table>
<thead>
<tr>
<th>Plan</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Motivation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Certification framework</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Checkers</strong></td>
<td></td>
</tr>
<tr>
<td><strong>High level API</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
<td></td>
</tr>
</tbody>
</table>
Proof tasks with meta-data, type and signature

\[ H_1 : A_1, \ldots, H_m : A_m \vdash G_1 : B_1, \ldots, G_n : B_n \]
Task abstraction: correction

\[ T \xrightarrow{A} T_i \]
\[ cT \xleftarrow{B_3} cT_i \]
\[ B_1 \]
\[ B_2 \]

\[ \Rightarrow \] we need equisatisfiability between a task and its abstraction
Certificates

type certif :=
  | Split of ident * certif * certif
  | Axiom of ident * ident
  | Hole
  | ...

A constructor ⇔ An elementary transformation
Certificates

type certif :=
| Split of ident * certif * certif
| Axiom of ident * ident
| Hole
| ...

A constructor ⇔ An elementary transformation

\[ \Gamma \vdash \Delta, G : A_1 \land A_2 \quad \Rightarrow \quad \text{Split}(G, \_ , \_) \quad \Rightarrow \quad \Gamma \vdash \Delta, G : A_1 \]

\[ \Gamma \vdash \Delta, G : A_1 \quad \Rightarrow \quad \Gamma \vdash \Delta, G : A_2 \]
Certificates

type certif :=
| Split of ident * certif * certif
| Axiom of ident * ident
| Hole
| ...

A constructor ⇔ An elementary transformation
Certificates

type certif :=
| Split of ident * certif * certif
| Axiom of ident * ident
| Hole
| ...

\[ \Gamma, H : A \vdash \Delta, G : A \]

A constructor \(\Leftrightarrow\) An elementary transformation

\text{Axiom}(H, G)
Certificates

type certif :=
| Split of ident * certif * certif
| Axiom of ident * ident
| Hole
| ...

A constructor ⇔ An elementary transformation
Certificates

type certif :=
| Split of ident * certif * certif
| Axiom of ident * ident
| Hole
| ...

A constructor ⇔ An elementary transformation
An example: the blast transformation

Pose:

$$\Gamma := H_1 : A_1, H_3 : A_3$$

$$T := \Gamma \vdash G : A_1 \land (A_2 \land A_3)$$
An example: the blast transformation

Pose:

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]

\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
An example: the blast transformation

Pose:

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]

\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
An example: the blast transformation

Pose:

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]

\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
An example: the blast transformation

Pose:
\[ \Gamma := H_1 : A_1, H_3 : A_3 \]
\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
An example: the blast transformation

Pose:

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]
\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
An example: the blast transformation

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]
\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]

\[ \Gamma \vdash G : A_1 \]
\[ \Gamma \vdash G : A_2 \]
\[ \Gamma \vdash G : A_3 \]

\[ \text{Split}(G, \text{Axiom}(H_1, G)) \]
\[ \text{Split}(G, \text{Hole, Axiom}(H_3, G)) \]

\[ \text{Split}(G, \text{Axiom}(H_1, G), \text{Split}(G, \text{Hole, Axiom}(H_3, G))) \]
Composition of certifying transformations

\[ cT \xrightarrow{\phi_1} cT_1 \xrightarrow{\phi_2} \cdots \xrightarrow{\phi_2} cT_m \rightarrow cT_a' \rightarrow \cdots \rightarrow cT_c' \rightarrow cT_d' \]
Framework for certificates

\[ T \xrightarrow{\phi} T_i \]

\[ abstract \]

\[ cT \xrightarrow{\text{checker}} cT_i \]

\[ certif \]

\[ abstract \]
OCaml checker: overview

\[ \text{ccheck} : \text{certif} \rightarrow \text{task} \rightarrow \text{task list} \]

**Correction of ccheck**

When \( \text{ccheck} \) \( \text{certif} \) \( cT \) computes into \( cT_i' \), the conjunction of all the \( cT_i' \) implies \( cT \).
The blast example:

\[ \Gamma := H_1 : A_1, H_3 : A_3 \]
\[ T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \]
\[ \text{certif} := \text{Split}(G, \text{Axiom}(H_1, G), \text{Split}(G, \text{Hole}, \text{Axiom}(H_3, G))) \]
OCaml checker: computing \textit{ccheck}

The blast example:

\[
\Gamma := H_1 : A_1, H_3 : A_3 \\
T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \\
certif := \text{Split} (G, \text{Axiom}(H_1, G), \text{Split}(G, \text{Hole}, \text{Axiom}(H_3, G)))
\]

Compute:

\[
c\text{check} \ certif \ T \\
\equiv c\text{check} (\text{Axiom}(H_1, G)) (\Gamma \vdash G : A_1) \\
\odot c\text{check} (\text{Split}(G, \text{Hole}, \text{Axiom}(H_3, G))) (\Gamma \vdash G : A_2 \land A_3)
\]
OCaml checker : computing *ccheck*

The blast example:

\[
\Gamma := H_1 : A_1, H_3 : A_3 \\
T := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \\
certif := \text{Split}(G, \text{Axiom}(H_1, G), \text{Split}(G, \text{Hole}, \text{Axiom}(H_3, G)))
\]

Compute:

\[
c\text{check } certif \ T \\
\equiv c\text{check } (\text{Axiom}(H_1, G)) (\Gamma \vdash G : A_1) \\
\circ c\text{check } (\text{Split}(G, \text{Hole}, \text{Axiom}(H_3, G))) (\Gamma \vdash G : A_2 \land A_3) \\
\equiv [] \\
\circ c\text{check } \text{Hole} (\Gamma \vdash G : A_2) \\
\circ c\text{check } (\text{Axiom}(H_3, G)) (\Gamma \vdash G : A_3)
\]
OCaml checker: computing \textit{cccheck}

The blast example:

\[
\begin{align*}
\Gamma & := H_1 : A_1, H_3 : A_3 \\
T & := \Gamma \vdash G : A_1 \land (A_2 \land A_3) \\
certif & := Split (G, Axiom(H_1, G), Split(G, Hole, Axiom(H_3, G)))
\end{align*}
\]

Compute:

\[
\text{cccheck} \, \text{certif} \, T
\]
\[
\equiv \text{cccheck} (Axiom(H_1, G)) (\Gamma \vdash G : A_1)
\]
\[
\oplus \text{cccheck} (Split(G, Hole, Axiom(H_3, G))) (\Gamma \vdash G : A_2 \land A_3)
\]
\[
\equiv [\]
\]
\[
\oplus \text{cccheck} \, \text{Hole} \, (\Gamma \vdash G : A_2)
\]
\[
\oplus \text{cccheck} (Axiom(H_3, G)) (\Gamma \vdash G : A_3)
\]
\[
\equiv [\Gamma \vdash G : A_2]
\]
Dedukti checker : overview

Aim : avoid a formal verification of ccheck’s code

⇒ use Dedukti !!

The checker step by step :

1. generate, from the tasks, a type $ty$ in Dedukti
2. generate, from the certificate, a term $t$ in Dedukti
3. check that $t : ty$ in Dedukti

The second step does not need to be trusted
Dedukti checker : generate type

First order logic encoding + excluded middle

Use the Dedukti arrow $\rightarrow$ to encode sequents:

$$\text{trad}(H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n) =$$

$$A_1 \rightarrow ... \rightarrow A_m \rightarrow \neg B_1 \rightarrow ... \rightarrow \neg B_n \rightarrow \bot$$

With initial task $cT$ and resulting tasks $cT_i$:

$$ty = \text{trad} (cT_1) \rightarrow ... \rightarrow \text{trad} (cT_n) \rightarrow \text{trad} (cT)$$

$\Rightarrow$ shallow embedding
Dedukti checker: generate term

Split:

\[
\frac{\Gamma \vdash \Delta, G : A \quad \Gamma \vdash \Delta, G : B}{\Gamma \vdash \Delta, G : A \land B}
\]

We define

\[
\text{split} : (\neg A \rightarrow \bot) \rightarrow (\neg B \rightarrow \bot) \rightarrow \neg(A \land B) \rightarrow \bot
\]

Need for elaboration:

\[\downarrow\text{find context of application, formulas employed} \ldots\]
Compilation: surface and kernel certificates

Surface certificates → Kernel certificates:

1. Elaborate, go through sequents to find missing information
2. Successively remove certificate constructors
Compilation: remove Construct certificate

$H_a : A, H_b : B \vdash\ Construct(H_a, H_b, H)\ H : A \land B \vdash$
Compilation: remove Construct certificate

\[ H_a : A, H_b : B \vdash \]

\[ \text{Construct}(H_a, H_b, H) \]

\[ H : A \land B \vdash \]

\[ H : A \land B, H_a : A, H_b : B \vdash \]

\[ \text{Weakening}(\ldots) \]

\[ H : A \land B \vdash \]

\[ H_a : A, H_b : B \vdash \]

\[ \text{Cut}(H \land A \land B) \]

\[ H_a : A, H_b : B \vdash H : A \land B \]

\[ \text{Split}(H, \ldots) \]
Allow reuse and reordering of resulting tasks by naming them:

\[[t_2; t_1], \text{Split} \ (G, \text{Hole } t_1, \text{Hole } t_2)\]
Naming tasks: destruct_disjunction

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \land B$

To simplify, assume the task is $\vdash A \land B$
Naming tasks: destruct_disjunction

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \land B$

To simplify, assume the task is $\vdash A \land B$
Naming tasks: destruct_disjunction

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \land B$

To simplify, assume the task is $\vdash A \land B$
Naming tasks: destruct_disjunction

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \land B$

To simplify, assume the task is $\vdash A \land B$
Naming tasks: destruct_disjunction
Naming tasks: destruct_disjunction
Naming tasks: destruct_disjunction

\[
\vdash A \land B \\
\vdash A \quad \text{cA} \quad \vdash A_1, \ldots, A_m \\
\vdash B \\
\vdash B_1, \ldots, B_n \\
\vdash A_1 \land B_1, \ldots, A_m \land B_n
\]
Naming tasks: destruct_disjunction

\[ \vdash A \land B \]

\[ \vdash A \]

\[ c_A \]

\[ \vdash A_1, \ldots, A_m \]

\[ c_{AB} \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]

\[ \vdash B \]

\[ c_B \]

\[ \vdash B_1, \ldots, B_n \]

\[ c_{BA} \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]

\[ t: \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]
Naming tasks: destruct_disjunction

\[ \vdash A \land B \]

Split

- \[ \vdash A \]
- \[ \vdash A_1, \ldots, A_m \]
- \[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]
- \[ \vdash B \]
- \[ \vdash B_1, \ldots, B_n \]
- \[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]
Naming tasks: destruct\_disjunction

\[ \vdash A \land B \]

\[ \vdash A \]

\[ \vdash A_1, \ldots, A_m \quad c_A \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \quad c_{AB} \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]

\[ \vdash A_1, \ldots, A_m \]

\[ \vdash B \]

\[ \vdash B_1, \ldots, B_n \quad c_B \]

\[ \vdash B_1 \land B_2, \ldots, B_n \quad c_{BA} \]

\[ \vdash B \land B_1, \ldots, B_n \]

\[ \vdash A \land B \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]

\[ \vdash A_1, \ldots, A_m \quad c_A \]

\[ \vdash B_1, \ldots, B_n \quad c_B \]

\[ \vdash A_1 \land B_1, \ldots, A_m \land B_n \]

\[ \vdash B_1 \land B_2, \ldots, B_n \quad c_{BA} \]
Contributions

A *generic, fine-grained* method to make transformations certifying *Software*:

- modular verification framework, including compilation of surface certificates
- 2 independant checkers
- transformation composition

*Applications*:

- ~15 simple certifying transformations
- blast
- rewrite and split_vc
- validation of transformations by Dedukti
- pigeonhole principle
- validation of the instantiate example in Why3
Future work and resources

Improve transformation certification:
- extend certification of `split_vc` and rewrite
- improve the certificate format with types, higher order
- compile and optimize surface certificates

Certify generation of verification conditions for imperative programs with aliases

Find more details in JFLA publication
→ https://hal.archives-ouvertes.fr/hal-02384946

Find latest work as a branch of Why3 at