Tools and Techniques for the Verification of Modular Stateful Programs

Mário José Parreira Pereira

LRI, Univ. Paris-Saclay, CNRS, Inria Saclay

PhD Defense
December 10th, 2018
Christmas is coming!
Thousands of South Carolinians won the lottery on Christmas — or so they thought. Now some are suing.
Who said *latte macchiatos* were expensive?

Starbucks stores reopen after computer glitch led to free coffee for some

- Outage was resolved on Friday night after several hours
- It affected registers at 7,000 stores in the US and 1,000 in Canada
We live in a world that runs on software.

Sensitive tasks are delegated to software.

We need to increase trust in software.
How to increase trust on software?

Formal methods

- Test, model checking, abstract interpretation
- Deductive software verification
  - Turn the correctness of your program into a mathematical statement
  - and prove it

  **interactive** proof assistants: Coq, PVS, Isabelle, KeY, ...
  **automatic** theorem provers: Alt-Ergo, CVC4, Z3, Spass, Vampire, ...
Two possible approaches

- verify **existing** code (e.g., Frama-C, SPARK, KeY)
- use a **proof-dedicated** language (e.g., Dafny, Why3)
Two possible approaches

- verify existing code (e.g., Frama-C, SPARK, KeY)
- use a proof-dedicated language (e.g., Dafny, Why3)

Extraction: automatic generation of a program.
The Why3 tool

Verification tool
- VCGen
- support for multiple theorem provers

WhyML, a programming language
- mutable data with static alias control
- exceptions
- ghost code and ghost data

A specification language
- type polymorphism
- algebraic data types, pattern matching
- (co)inductive definitions
The Why3 tool

Verification tool
- VCGen
- support for multiple theorem provers

WhyML, a programming language
- mutable data with static alias control
- exceptions
- ghost code and ghost data

A specification language
- type polymorphism
- algebraic data types, pattern matching
- (co)inductive definitions
Contributions

1. Program extraction in Why3
2. Why3 extraction formalization
3. Application: a framework for verified OCaml programs
4. A specification language for OCaml
5. Proof of OCaml idiomatic features, not directly supported in Why3
Find a **power of two** that is **greater** than a given $n$. 
Find a power of two that is greater than a given \( n \).

```haskell
let power2_above (n: int) : int
    = let ref p = 1 in
      while p <= n do
        p <- p + p;
      done;
p
```
Find a power of two that is greater than a given $n$.

```ml
let power2_above (n: int) : int =
  let ref p = 1 in
  while p <= n do
    variant { n - p }
    p <- p + p;
  done;
  p
```
Find a power of two that is greater than a given \( n \).

```ocaml
def power2_above (n: int): int
    ensures \{ exists k. result = 2^k \&\& result > n \}
= let ref p = 1 in

    while p <= n do
        variant \{ n - p \}
        p <- p + p;
    done;

    p
```
Example of program extraction

Find a power of two that is greater than a given \( n \).

```ocaml
let power2_above (n: int): int
    ensures { exists k. result = 2^k \/\ result > n }
= let ref p = 1 in
  let ghost ref k = 0 in
  while p <= n do
    variant { n - p }
    p <- p + p;
    k <- k + 1
  done;
  p
```
Find a power of two that is greater than a given \( n \).

\[
\text{let power2_above (n: int): int} \\
\quad \text{ensures } \{ \text{exists } k. \text{ result } = 2^k \land \text{ result } > n \} \\
= \text{let ref p = 1 in} \\
\quad \text{let ghost ref k = 0 in} \\
\quad \text{while } p \leq n \text{ do} \\
\quad \quad \text{variant } \{ n - p \} \\
\quad \quad \text{invariant } \{ k \geq 0 \land p = 2^k \} \\
\quad \quad p \leftarrow p + p; \\
\quad \quad k \leftarrow k + 1 \\
\quad \text{done; k + 1} \\
\]

p
Find a power of two that is greater than a given \( n \).

```ocaml
let power2_above (n: int) : (r: int, ghost k: int)
  ensures { r = 2^k \( \text{and} \) r > n }
= let ref p = 1 in
  let ghost ref k = 0 in
  while p <= n do
    variant { n - p }
    invariant { k >= 0 \( \text{and} \) p = 2^k }
    p <- p + p;
    k <- k + 1
  done;
  p, k
```
Data and code added for the purposes of specification/proof.

We should be able to remove it with no observable modification.

Consequently, ghost code

- cannot alter the control flow
- can read regular data, but cannot modify it
> why3 extract -D ocaml64 power2_above.mlw

```ocaml
let power2_above (n: Z.t) : Z.t =
  let p = ref Z.one in
  while Z.leq !p n do
    p := Z.add !p !p
  done;
  !p
```
Re-implementation of the Why3 extraction mechanism

WhyML → intermediate language → OCaml

remove: ghost spec

optim.
Re-implementation of the Why3 extraction mechanism

WhyML → intermediate language → OCaml

remove: ghost spec

C

optim.
driver

module int.Int
    syntax val one "Z.one"
    syntax val (<=) "Z.leq %1 %2"
    syntax val (+) "Z.add %1 %2"
    ...
end
2. Why3 extraction formalization
source code

fun f (y: ghost int, x: int) : (int, int) =
  (x, x) in
fun g () : (ghost int, int) =
  f (89, 42) in
g ()
source code

```haskell
fun f (y: ghost int, x: int) : (int, int) =
  (x, x) in
fun g () : (ghost int, int) =
  f (89, 42) in

extracted code

fun f (y: unit, x: int) : (int, int) =
  (x, x) in
fun g () : (unit, int) =
  f ((), 42) in

```
Extraction challenges

source code

fun f (y: ghost int, x: int) : (int, int) =
  (x, x) in
fun g () : (ghost int, int) =
  f (89, 42) in
g ()

extracted code

fun f (y: unit, x: int) : (int, int) =
  (x, x) in
fun g () : (unit, int) =
  let _, y = f ((), 42) in ((), y) in
g ()
source code

fun f (y: ghost int, x: int) : (int, int) =
   (x, x) in
fun g () : (ghost int, int) =
   f (89, 42) in
g ()

extracted code

fun f (x: int) : (int, int) =
   (x, x) in
fun g () : int =
   let _, y = f (42) in y in
g ()
Simplified WhyML

\[ e ::= \]
\[ \quad \bar{a} \]
\[ \quad \text{tuple of atomic expressions} \]
\[ \quad \text{local variables binding} \]
\[ \quad \text{local function definition} \]
\[ \quad \text{function application} \]
\[ \quad \text{record construction} \]
\[ \quad \text{record field assignment} \]
\[ \quad \text{infinite loop} \]
\[ \quad \text{exception raising} \]
\[ \quad \text{exception catching} \]
\[ \quad \text{ghost expression} \]

\[ a ::= x | c \]
\[ \quad \text{atomic expression} \]

\[ \beta ::= \text{reg} | \text{ghost} \]
\[ \quad \text{ghost status} \]
\[ \delta \cdot \mu \cdot e \Downarrow \mu' \cdot r \quad \delta \cdot \mu \cdot e \Downarrow^{\text{co}} \text{div} \]

\[ \nu ::= c \mid l \quad \text{value} \]

\[ r ::= \overline{\nu} \mid \text{raise } E \overline{\nu} \quad \text{semantic result} \]

\[ \mu \triangleq l \mapsto \{ f = \nu \} \quad \text{store} \]

\[ \delta \triangleq f \mapsto (x : \beta \tau, e) \quad \text{functions environment} \]
Type and effect system

\[ \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\bar{\beta} \tau, \epsilon, \gamma) \]

\[ \tau ::= \alpha | \tau \tau \]

\[ \Delta \equiv f \mapsto (\chi : \bar{\beta} \tau) \rightarrow \bar{\beta}^{'} \tau^{'} \]

\[ \Gamma \equiv \chi \mapsto \beta \tau \]

\[ \Sigma \equiv l \mapsto \tau \tau \]

\[ \epsilon, \gamma \equiv (\text{writes } \bar{x}, \text{div}?, \text{raises } \bar{E}) \]
Given a non-ghost WhyML expression

\[ e : \overline{\beta'}\tau \]

we define another WhyML expression

\[ \mathcal{E}_{\overline{\beta}}(e) \]

for any \(\overline{\beta}\) such that \(\overline{\beta'} \sqsubseteq \overline{\beta}\),

where \(\text{reg} \sqsubseteq \text{ghost}\) is lifted point-wise.
\[ \mathcal{E}_\beta(\overline{a}) \triangleq \overline{a}_{\downarrow \beta} \]

\[ \mathcal{E}_\beta \left( \text{fun } f(x : \beta''\tau') : \overline{\beta'}\tau = e_1 \text{ in } e_2 \right) \triangleq \begin{cases} 
\mathcal{E}_\beta(e_2) & \text{if } \overline{\beta'} = \text{ghost } \land \epsilon(e_1) = \emptyset \\
\text{fun } f(\mathcal{E}(x : \beta''\tau')) : \mathcal{E}(\overline{\beta'}\tau) = \mathcal{E}_{\beta'}(e_1) & \text{otherwise}
\end{cases} \]

\[ \mathcal{E}_\beta(f(\overline{a})) \triangleq \text{let } \overline{\text{reg } x = f(\mathcal{E}_{\beta'}(\overline{a}))} \text{ in } \overline{x}_{\downarrow \beta} \]
Assumption: extraction is applied to a fully verified program

Theorem (Well-typedness preservation by extraction)
If \( \Delta \cdot \Gamma \cdot \Sigma \vdash e : (\overline{\beta'}\tau, \epsilon, \gamma) \), where \( \overline{\beta'} \neq \text{ghost} \lor \epsilon \neq \emptyset \), then for all \( \overline{\beta} \) such that \( \overline{\beta'} \subseteq \overline{\beta} \), we have \( \mathcal{E}(\Delta) \cdot \mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash \mathcal{E}_{\overline{\beta}}(e) : \mathcal{E}_{\overline{\beta}}(\overline{\beta'}\tau, \epsilon, \gamma) \).

Theorem (Semantic preservation by extraction)
\[
\begin{align*}
\delta \cdot \mu \cdot e & \xrightarrow{\mathcal{E}_{\overline{\beta}}} \mathcal{E}_{\overline{\beta}}(\delta \cdot \mu \cdot e) \\
\mu' \cdot r & \xrightarrow{\mathcal{E}_{\overline{\beta}}} \mathcal{E}_{\overline{\beta}}(\mu' \cdot r) \\
\delta \cdot \mu \cdot e & \xrightarrow{\text{co}} \mathcal{E}_{\overline{\beta}}(\delta \cdot \mu \cdot e) \\
\mu' \cdot r & \xrightarrow{\text{div}} \mathcal{E}_{\overline{\beta}}(\mu' \cdot r)
\end{align*}
\]
3. Application: a framework for verified OCaml programs
<table>
<thead>
<tr>
<th>OCaml</th>
<th>Why3</th>
</tr>
</thead>
</table>

Workflow
Workflow

OCaml

spec.

.mli +

Why3

proof of refinement

extraction

implements
Workflow

OCaml

.mli + spec.

translation

Why3

.mlw spec.
**Workflow**

OCaml -> .mli + spec. -> translation -> .mlw spec. -> Why3

.mlw implem.
Workflow

OCaml

.mli + spec.

Why3

.mlw spec.

.mlw implem.

proof

translation
OCaml

.ml + spec.

Why3

.mlw spec.

proof of refinement

.mlw implem.

proof of refinement
Workflow

OCaml

.ml + spec.

Why3

.mlw spec.

proof of refinement

.ml

.extraction

.mlw implem.

proof
Workflow

OCaml

.ml + spec.

translates to

.spec.

Why3

.mlw spec.

proof of refinement

.ml implements

.extracted to

.mlw implem.

proof
DEMO
# Experimental evaluation

<table>
<thead>
<tr>
<th>module</th>
<th>.mli</th>
<th>.mlw</th>
<th>#VCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnionFind</td>
<td>67</td>
<td>84</td>
<td>63</td>
</tr>
<tr>
<td>PairingHeap</td>
<td>34</td>
<td>85</td>
<td>45</td>
</tr>
<tr>
<td>ZipperList</td>
<td>50</td>
<td>89</td>
<td>58</td>
</tr>
<tr>
<td>Arrays</td>
<td>31</td>
<td>73</td>
<td>95</td>
</tr>
<tr>
<td>Queue</td>
<td>38</td>
<td>104</td>
<td>124</td>
</tr>
<tr>
<td>Vector</td>
<td>122</td>
<td>146</td>
<td>98</td>
</tr>
<tr>
<td>MergeSort</td>
<td>11</td>
<td>187</td>
<td>703</td>
</tr>
<tr>
<td>VectorHeap</td>
<td>40</td>
<td>131</td>
<td>223</td>
</tr>
<tr>
<td>Total</td>
<td>393</td>
<td>899</td>
<td>1314</td>
</tr>
</tbody>
</table>
ANR project, 1/10/2015 - 30/9/2020

VALS @UPSaclay, Gallium @Inria, PACSS @Verimag, TrustInSoft, OCamlPro

A library of efficient general-purpose data structures and algorithms.

A combination of three different tools

- Why3
- CFML
- Coq
Conclusion
Contributions

Implementation and formalization of Why3’s extraction mechanism.

Use of Why3 inside the VOCaL project.

Proof of non-trivial OCaml programs

- recursive mutable data structures
- functorial code
- a unified treatment of iteration

UnionFind, MergeSort, PairingHeap, VectorHeap

Vector.iter, Vector.fold
The road ahead

Short and mid-term:
- more verified modules in VOCaL
- formalization of the Why3 module system
- support for higher-order stateful programs

Long-term:
- smooth collaboration with other verification tools
- application to industrial-size projects: can the extraction approach scale up? e.g., verification of Mirage OS?
M. Clochard, L. Gondelman, and M. Pereira.

J-C. Filliâtre and M. Pereira.
A modular way to reason about iteration. *NFM*, 2016.

J-C. Filliâtre and M. Pereira.
Producing all ideals of a forest, formally. *VSTTE*, 2016.

A. Charguéraud, J-C. Filliâtre, M. Pereira, and F. Pottier.

J-C. Filliâtre and M. Pereira.

M. Pereira.

J-C. Filliâtre, M. Pereira, and S. M. de Sousa.

VOCaL – The Verified OCaml Library
[github.com/vocal-project/vocal](https://github.com/vocal-project/vocal)
1. mutable recursive data structures
mutable recursive data structures

Why3 type system constraint:

all aliases are statically known

⇔

depth bounded mutability

pitfall: recursive mutable data types cannot be directly encoded

solution: build a specific model of the heap

• a type of pointers
• a table to map each pointer to its current value
• read/write operations

examples: Frama-C, Dafny, VeriFast, CFML
```
type content =
  | Link of content ref
  | Root
```
memory model for union find

type loc

type content = Link loc | Root

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

type content = Link loc | Root

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

val alloc_ref (ghost h: heap) (v: content) : loc
  ...

val get_ref (ghost h: heap) (l: loc) : content
  ...

val set_ref (ghost h: heap) (l: loc) (v: content) : unit
  ...

val (==) (x y: loc) : bool
  ...
memory model for union find

type loc

type content = Link loc | Root

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

val alloc_ref (ghost h: heap) (v: content) : loc
  writes { h.refs }
  ensures { (old h).refs result = None }
  ensures { h.refs = (old h.refs)[result <- Some v] }

val get_ref (ghost h: heap) (l: loc) : content
  ...

val set_ref (ghost h: heap) (l: loc) (v: content)
  ...

val (==) (x y: loc) : bool
  ...
small custom driver file uf.drv

module uf.UnionFind
    syntax type loc "content ref"
syntax val alloc_ref "ref %1"
syntax val get_ref "!%1"
syntax val set_ref "%1 := %2"
syntax val (==) "%1 == %2"
end

> why3 extract -D ocaml64 -D uf.drv -L .
    uf.UnionFind -o uf.ml
2. functorial code
functorial code
(** Pairing Heaps *)

module Make (X: sig
  type t
  val compare: t -> t -> int
end) = struct

  type heap = E | T of X.t * heap list

  let merge h1 h2 = match h1, h2 with
    | T (x1, l1), T (x2, l2) ->
      if X.compare x1 x2 <= 0 then T (x1, h2 :: l1)
      else T (x2, h1 :: l2)
    ...

end
Why3 functors-ish

```ocaml
scope Make
  scope X
    type t
    val compare t t : int63
  end

type heap = E | T X.t (list heap)

let merge (h1 h2: heap) : heap = match h1, h2 with
  | T x1 l1, T x2 l2 ->
    if X.compare x1 x2 <= 0 then T x1 (Cons h2 l1)
    else T x2 (Cons h1 l2)
  ...
end
```

> why3 extract -D ocaml64 -L . ph.PairingHeap --modular -o .
3. reasoning about iteration
fold : (‘b -> ’a -> ’b) -> ’b -> ’a collection -> ’b
fold : ('b -> 'a -> 'b) -> 'b -> 'a collection -> 'b

iteration formalized with

1. a ghost state \( \nu \)

\[
\nu_0 \quad \nu_1 \quad \ldots \quad \nu_{n-1}
\]

\( \text{already enumerated} \)

\( \text{to come} \)

2. two predicates
   - \textit{permitted} characterizes the elements \textit{already produced}
   - \textit{complete} states when we are done with the enumeration
proving iteration providers and clients

different forms of iteration

• cursors
• higher-order iterators

abstraction barrier between client code and provider

proof via program transformations

• pure higher-order iterators turned into first-order implementations
• stateful higher-order iterators turned into cursors