A Formally-Verified C static analyzer

David Pichardie

joint work with J.-H. Jourdan, V. Laporte, S.Blazy, X. Leroy, presented at POPL’15
How do you trust your software?
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques.
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques

• Manual verifications
  – do not scale
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
How do you trust your software?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms
  - ex: the Astrée static analyzer


~1M loc of a critical control-command software analyzed

0 false alarms

manual verification | bug finders | sound verifiers
---|---|---
yesterday | today | tomorrow
How do you trust the tool that verifies your software?

The increasing complexity of safety critical systems requires efficient validation techniques

- Manual verifications
  - do not scale
- Automatic bug finders
  - may miss some bugs
- Automatic, sound verifiers
  - find all bugs, may raise false alarms
  - ex: the Astrée static analyzer
- Formally-verified verifiers
  - the verifier comes with a soundness proof
  - that is machine checked

~1M loc of a critical control-command software analyzed
0 false alarms

manual verification    bug finders    sound verifiers    verified verifiers
yesterday              today           tomorrow           after tomorrow
How do you we verify a verifier?
How do you we verify a verifier?

A simple idea:
How do you we verify a verifier?

A simple idea:

Program and prove your verifier in the same language!
How do you verify a verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language?
How do you we verify a verifier?

A simple idea:

Program and prove your verifier in the same language!

Which language?

Coq
Coq: an animal with two faces
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
- a functional programming language with a very rich type system
Coq: an animal with two faces

First face:
• a proof assistant that allows to interactively build proof in constructive logic

Second face:
• a functional programming language with a very rich type system
example:

\[
\text{sort: } \forall l: \text{list int}, \{ l': \text{list int} \mid \text{Sorted } l' \land \text{PermutationOf } l \ l' \}
\]
Coq: an animal with two faces

First face:
- a proof assistant that allows to interactively build proof in constructive logic

Second face:
- a functional programming language with a very rich type system
  example:

  `sort : ∀ l : list int, { l' : list int | Sorted l' ∧ PermutationOf l l' }`

- with an extraction mechanism to Ocaml

  `sort : int list → int list`
Our methodology
Our methodology

We program the static analyzer inside Coq

\textbf{Definition} analyzer \((p:\text{program}) := \ldots\)
Our methodology

We program the static analyzer inside Coq

**Definition** analyzer (p:program) := ...

We state its correctness wrt a formal specification of the language semantics

**Theorem** analyser_is_sound :
\[ \forall p, \text{analyser } p = \text{Yes} \rightarrow \text{Sound}(p) \]
Our methodology

We program the static analyzer inside Coq

\textbf{Definition} \ analyser \ (p:\text{program}) := \ldots

We state its correctness wrt. a formal specification of the language semantics

\textbf{Theorem} \ analyser\_is\_sound :
\forall \ p, \ analyser \ p = \text{Yes} \rightarrow \text{Sound}(p)

We interactively and mechanically prove this theorem

\textbf{Proof}. \ldots \ (\ast \ \text{few days later} \ \ast) \ \ldots \ \text{Qed}. 
Our methodology

We program the static analyzer inside Coq

Definition analyzer (p:program) := ...

We state its correctness wrt. a formal specification of the language semantics

Theorem analyser_is_sound :
\forall p, analyzer p = Yes \rightarrow Sound(p)

We interactively and mechanically prove this theorem

Proof. ... (* few days later *) ... Qed.

We extract an OCaml implementation of the analyzer

Extraction analyzer.
Verified Static Analysis meets the verified CompCert C compiler
Background: verifying a compiler

CompCert, a moderately optimizing C compiler usable for critical embedded software

= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

For all source programs S and compiler-generated code C, if the compiler generates machine code C from source S, without reporting a compilation error, then «C behaves like S».

• Compiler written from scratch, along with its proof; not trying to prove an existing compiler
Compcert meets the industrial world

Fly-by-wire software, for recent Airbus planes
  • control-command code generated from block diagrams (3600 files, 3.96 MB of assembly code)
  • minimalistic OS

Results
  • Estimated WCET for each file
  • Average improvement per file: 14%
  • Compiled with CompCert 2.3, May 2014

Conformance to the certification process (DO-178)
  • Trade-off between traceability guarantees and efficiency of the generated code
The Verasco project
INRIA Celtique, Gallium, Abstraction, Toccata + VERIMAG + Airbus

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation

- Language analyzed: the CompCert subset of C
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée's
- Decent alarm reporting

Slogan:

- if « CompCert \(\approx\) 1/10\(^{th}\) of GCC but formally verified »,
- likewise « Verasco \(\approx\)1/10\(^{th}\) of Astrée but formally verified »

http://verasco.imag.fr
Modularity

Astrée is highly modular and programmed in ML

Verasco is highly modular and programmed in Coq
Building a static analyzer in ML

Modular design

```ocaml
module IntervalAbVal : ABVAL = ...
module NonRelAbEnv (AV:ABVAL) : ABENV = ...
module SimpleAbMem (AE:ABENV) : ABMEMORY = ...
module Iterator (AM:ABMEMORY) : ANALYZER = ...
module myAnalyzer = Iterator(SimpleAbMem(NonRelAbEnv(IntervalAbVal)))
```

Example of interface

```ocaml
module type ABDOM = sig
  type ab
  val le : ab -> ab -> bool
  val top : ab
  val join : ab -> ab -> ab
  val widen : ab -> ab -> ab
end
```
Building a static analyzer

in ML

module type ABDOM = sig
  type ab
  val le : ab → ab → bool
  val top : ab
  val join : ab → ab → ab
  val widen : ab → ab → ab
end

in Coq

Class adom (ab:Type) (c:Type) := {
  le : ab → ab → bool;
  top : ab;
  join : ab → ab → ab;
  widen : ab → ab → ab;

  gamma : ab → ℘(c);

  gamma_monotone : ∀ a1 a2,
    le a1 a2 = true ⇒
    gamma a1 ⊆ gamma a2;

  gamma_top : ∀ x,
    x ∈ gamma top;

  join_sound : ∀ x y,
    gamma x U gamma y
    ⊆ gamma (join x y)
}
Lazy proofs

Proof by necessity

- We don’t prove properties that are not strictly necessary to establish a soundness theorem.

What we don’t prove

- \((ab, le, join)\) enjoy a lattice structure
- \(\gamma\) is a meet morphism between complete lattices (Galois connection)
- \(\text{widen}\) is a sound widening operator

```plaintext
Class adom (ab:Type) (c:Type) := {
  le : ab \to ab \to bool;
  top : ab;
  join : ab \to ab \to ab;
  widen : ab \to ab \to ab;

  gamma : ab \to \wp(c);

  gamma_monotone : \forall a1 a2, 
                  le a1 a2 = true \Rightarrow 
                  gamma a1 \subseteq gamma a2;
  gamma_top : \forall x, 
              x \in gamma top;
  join_sound : \forall x y, 
              gamma x \cup gamma y 
              \subseteq gamma (join x y)
}
```
Each layer is parameterized by the underlying one.
CompCert: 1 compiler, 11 languages

Where should we perform the analysis?
Compcert behavior preservation theorems

Theorem [Behavior Preservation]
\[ \forall P_i \forall P_{i+1}, C(P_i) = P_{i+1} \implies B(P_{i+1}) \subseteq B(P_i) \]

Corollaries

- If target program goes wrong, source program goes wrong too.
- A program verifier on \( P_i \), gives useful information on \( P_{i+1} \), but not necessarily on \( P_{i-1} \).
Which CompCert representation?

**RTL**
- the place where most CompCert optimizations take place
- but platform specific, flat expressions

**C source**
- a language for programmer, not for tools
- ultimately we want to gives alarms at this level

**Clight**
- C syntax without side-effect in expressions

**C#minor**
- almost like Clight but with tool-friendly syntax (e.g. store/load instruction)
C#minor Abstract interpreter (1/2)

C#minor

• structured statements
• exit \(n\) (encoding break/continue) : jumps to the end of the \((n+1)\)-th enclosing block
• goto with labels
• variables
  • global (their address can be taken, statically allocated)
  • local (their can be taken, dynamically allocated/freed at each function call/return)
• temporary (not resident in memory)
C#minor Abstract interpreter (2/2)

Structural approach instead of CFG approach

- obviates the need to define program points
- uses less memory than the CFG-based interpreter
- transfer functions are more involved (control can leave a stmt in many ways)
- local fixpoint solving at each loop
- function call trigger a recursive call of the abstract interpreter
- goto requires a global fixpoint computation

Parameterized by a relational abstract domain for execution states (environment + memory state + call stack)
The state abstract domain (1/2)

Abstract memory cell: 1 unit of storage

\[ c ::= \text{temp}(f,t) \mid \text{local}(f,x,\text{offset},\text{size}) \mid \text{global}(x,\text{offset},\text{size}) \]

Abstract value: (cell types, points-to graph, numerical abstraction)

The domain is parameterized by a relational numerical domain where cells act as variables.
The state abstract domain (2/2)

Example:

\[
t := &T; \; \ldots; \; s := s + \text{load}(\text{int32}, t + 8\times i + 4)
\]

1. Points-to information says « t contains a pointer to global T »

2. Numerical domain gives a range \{4, 12\} for the numerical value \(8\times i + 4\)

3. We approximate the load expression with two numerical cell assignments
   \[
   \text{temp}(f,s) := \text{temp}(f,s) + \text{global}(T,4,\text{int32})
   \]
   \[
   \text{temp}(f,s) := \text{temp}(f,s) + \text{global}(T,12,\text{int32})
   \]

4. And finally take the join of the two new numerical abstract states thus obtained.
Abstract numerical domains

Verimag work

Z → int

Convex polyhedra

Symbolic equalities

Nonrel → Rel

Nonrel → Rel

Integer congruences

Integer & F.P. intervals

CompCert C → Clight → C#minor → ... CompCert compiler

Alarms

Abstract interpreter

control flow

Memory & value domain

states

VERIMAG work
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms ← Abstract interpreter

control flow

Memory & value domain

Memory & value domain transforms any rel. domain over \( \mathbb{Z} \) into a rel. domain over machine integers with modulo arithmetic

states

numbers

\( \mathbb{Z} \rightarrow \text{int} \)

Convex polyhedra
Symbolic equalities
Nonrel \( \rightarrow \) Rel
Nonrel \( \rightarrow \) Rel
Integer congruences
Integer & F.P. intervals

VERIMAG work
Abstract numerical domains

- **CompCert C** → **Clight** → **C#minor** → ... → **CompCert compiler**

- **Alarms** ← **Abstract interpreter** with control flow

- **Memory & value domain** with states

- **Z → int**

- **Convex polyhedra**

- **Symbolic equalities**

- **Nonrel → Rel**

- **Integer congruences**

- **Nonrel → Rel**

- **Integer & F.P. intervals**

Conjunctions of linear inequalities $\sum a_i x_i \leq c$ [SAS’13]

VERIMAG work
Abstract numerical domains

- CompCert C → Clight → C#minor → ... → CompCert compiler
- Abstract interpreter
  - Alarms
  - Memory & value domain
    - Z → int
      - Symbolic conditional expressions (improve precision of assume commands)
    - Convex polyhedra
      - Symbolic equalities
    - Nonrel → Rel
      - Integer congruences
    - Nonrel → Rel
      - Integer & F.P. intervals

VERIMAG work
Abstract numerical domains

- CompCert C → Clight → C#minor → ... → CompCert compiler
- Abstract interpreter
  - Alarms
  - Memory & value domain
    - Nonrel → Rel
    - Nonrel → Rel
  - $Z \rightarrow \text{int}$
  - Convex polyhedra
  - Symbolic equalities
  - Integer congruences
  - Integer & F.P. intervals

VERIMAG work

transforms any non-rel. domain into a (reduced) rel. domain
Abstract numerical domains

- **CompCert C** → **Clight** → **C#minor** → ... → **CompCert compiler**

- **Alarms** → **Abstract interpreter**
  - **control flow**

- **Memory & value domain**
  - **states**

- **Z → int**

- **crucial to analyze the safety of memory accesses (memory alignment)**

- **Convex polyhedra**: VERIMAG work

- **Symbolic equalities**

- **Nonrel → Rel**
  - **Integer congruences**
  - **Integer & F.P. intervals**

- **Nonrel → Rel**
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms ← Abstract interpreter

control flow

Memory & value domain

states

Z → int

Convex polyhedra
Symbolic equalities
Nonrel → Rel
Nonrel → Rel

 Integer congruences

 Integer & F.P. intervals

VERIMAG work

requires reasoning on double-precision floating-point numbers (IEEE754)
Abstract numerical domains

VERIMAG work
Combining abstract domains

Implementations of reduced products tend to be specific to the 2 domains being combined.

System of inter-domains communication channels inspired by that of Astrée [ASIAN’O6]

- Channels are used by domains when they need information from another domain.
- The information already present in a channel is enriched with information of a query.
Implementation

34,000 lines of Coq, excluding blanks and comments

- half proof, half code & specs
- plus parts reused from CompCert

Bulk of the development: abstract domains for states and for numbers (involve large case analyses and difficult proofs over integer and floating points arithmetic)

Except for the operations over polyhedra, the algorithms are implemented directly in Coq’s specification language.
Experimental results

Preliminary experiments on small C programs (up to a few hundred lines)

- CompCert benchmarks
- Cryptographic routines (NaCl library)

Exercise many delicate aspects of the C language: arrays, pointer arithmetic, function pointers, floating-point arithmetic.

The analyzer can takes < 1 minute to analyze a few hundred lines of C.

We obtain automatically foundational « can’t go wrong » proofs for non-trivial C programs
Conclusion

Static analyzer based on abstract interpretation which establishes the absence of run-time errors in C programs (excluding recursion and dynamic allocation)

\begin{theorem}
\text{vanalysis\_is\_correct:}
\begin{align*}
\forall \text{prog, tr}, \\
\text{vanalysis prog = OK} \Rightarrow \\
\text{program\_behaves (semantics prog) (Goes\_wrong tr) \Rightarrow False.}
\end{align*}
\end{theorem}

Modular architecture supporting the extensible combination of multiple abstract domains (relational and non-relational)

Integrates with CompCert, so that the soundness of the analysis is guaranteed on the compiled code as well
A holistic effect with compiler verification

Compiler

**Theorem** transf_c_program_is_refinement:
forall p tp, transf_c_program p = OK tp →
(forall behv, exec_C_program p behv → not_wrong behv) →
(forall behv, exec_Asm_program tp behv → exec_C_program p behv).
A holistic effect with compiler verification

Compiler

**Theorem** transf_c_program_is_refinement:
for all \( p, tp \), \( \text{transf\_c\_program} \ p = \text{OK} \ tp \rightarrow \\
(\forall \text{behv}, \text{exec\_C\_program} \ p \ \text{behv} \rightarrow \text{not\_wrong} \ \text{behv}) \rightarrow \\
(\forall \text{behv}, \text{exec\_Asm\_program} \ tp \ \text{behv} \rightarrow \text{exec\_C\_program} \ p \ \text{behv}).

Static analyzer

**Theorem** analyzer_is_correct:
for all \( p \), \( \text{static\_analyzer\_result} \ p = \text{Success} \rightarrow \\
(\forall \text{behv}, \text{exec\_C\_program} \ p \ \text{behv} \rightarrow \text{not\_wrong} \ \text{behv}).
A holistic effect with compiler verification

Compiler

\textbf{Theorem} \texttt{transf\_c\_program\_is\_refinement}:
\begin{align*}
& \forall p \text{ tp}, \texttt{transf\_c\_program } p = \texttt{OK} \text{ tp } \rightarrow \\
& (\forall \text{ behv}, \texttt{exec\_C\_program } p \text{ behv } \rightarrow \texttt{not\_wrong } \text{ behv}) \rightarrow \\
& (\forall \text{ behv}, \texttt{exec\_Asm\_program } \text{ tp } \text{ behv } \rightarrow \texttt{exec\_C\_program } p \text{ behv}).
\end{align*}

\textbf{Static analyzer}

\textbf{Theorem} \texttt{analyzer\_is\_correct}:
\begin{align*}
& \forall p, \texttt{static\_analyzer\_result } p = \texttt{Success } \rightarrow \\
& (\forall \text{ behv}, \texttt{exec\_C\_program } p \text{ behv } \rightarrow \texttt{not\_wrong } \text{ behv}).
\end{align*}

\textbf{Stronger correctness result}

\textbf{Theorem} \texttt{transf\_c\_program\_is\_refinement}:
\begin{align*}
& \forall p \text{ tp}, \texttt{transf\_c\_program } p = \texttt{OK} \text{ tp } \rightarrow \\
& \texttt{static\_analyzer\_result } p = \texttt{Success } \rightarrow \\
& (\forall \text{ behv}, \texttt{exec\_Asm\_program } \text{ tp } \text{ behv } \rightarrow \texttt{exec\_C\_program } p \text{ behv}).
\end{align*}
Future directions

Engineering
• from C#minor to Clight
• better support for precision debugging

Improving the algorithmic efficiency of the static analyzer
• from Coq’s integer and FP arithmetic (list of bits) to more efficient libraries
• improve physical sharing of purely functional data structures used for maps and sets

Extend the memory abstract domain
• dynamic memory allocation
• recursion (also requires modifying the iterator)

Improving the precision of the analysis
• on-the-fly unrolling of certain loops (based on unverified heuristics)
• new abstract domains, e.g. octagons
Questions ?
Related work


Communications between domains

Communication channels between abstract operators.

\[ A_{in} \rightarrow C_{in} \rightarrow C'_{out} \rightarrow A_{out} \]

\[ A_{in} \rightarrow (C_{in} \rightarrow C'_{out} \rightarrow A_{out}) \]

\[ (ab, chan) \rightarrow \]