Verasco: a Formally Verified C Static Analyzer

Jacques-Henri Jourdan

Joint work with:
Vincent Laporte, Sandrine Blazy,
Xavier Leroy, David Pichardie, . . .

June 13, 2017, Montpellier
GdR GPL thesis prize
Static analyzers

They **automatically** prove the absence of certain kinds of bugs

- Examples: “No invalid memory access”, “No division by 0”, “Small rounding errors”, ...
- Undecidable problem:
  - Success $\Rightarrow$ no bug of some class
  - In case of doubt, emit an alarm

Exemples : Astrée, Frama-C EVA, Fluctuat, Sparrow, ...
Abstract interpretation

**Abstract interpretation**: theory for building static analyzer

- Run the program using an *abstract semantics*
- Always **terminating** computation
- **Soundly** approximating the concrete semantics

**Abstract domains** used to approximate program states

- Ex: variation intervals for integer variables...
Abstract interpretation: example

x=0

loop {
  if x > 11
    break
  x+=2
}

4 of 28
Abstract interpretation: example

\{\top\}
\begin{align*}
x &= 0 \\
\{x = 0\} \\
\text{loop} \{ \\
\{x \in [0, 13] \wedge x \text{ mod } 2 = 0\} \\
\quad \text{if } x > 11 \\
\{x = 12\} \\
\quad \text{break} \\
\{x \in [0, 11] \wedge x \text{ mod } 2 = 0\} \\
\quad x += 2 \\
\{x \in [2, 13] \wedge x \text{ mod } 2 = 0\} \\
\} \\
\{x = 12\}
Uses of static analyzers

Two main use cases:
- Bug finders
  - Very precise, no direct need for correctness
- Program verification
  - Used to prove properties on critical code
  - Need for trust

Static analyzers are complex
- Advanced algorithms (linear optimization, symbolic manipulations, ...)
- Technical problems (floating-point, C semantics, ...)
⇒ probably buggy
A static analyzer is a program, we can prove it!

Verasco:

- Programmed and **formally verified** using the Coq proof assistant
- Based on **abstract interpretation**
- Handles **most of C99**
  - industrial, widely used language
  - no dynamic memory allocation, no recursion
- Proves the absence of undefined behaviors
  - dynamic type errors
  - memory errors
  - arithmetic exceptions
Introduction

Overview of Verasco

Technical zoom: numerical abstract domains

Conclusions
Textbook formalized static analyzers

- IMP
- Abstract interpreter
- Abstract domain

- IMP toy language
- No handling of pointers
- Unbounded integers abstract domain
Modular architecture

Reuses the **CompCert** front-end
- Until C#minor
- Language simpler than C99
- Uses the same formal semantics as CompCert
  - **Guarantees** provided by the analyzer provably extend to the assembly code.
- A priori unsound if used with another C99 semantics
Modular architecture

Abstract interpreter

- Fixpoints using widening for loops and gotos
- Proved correct using a Hoare logic for C#minor
- Parameterized by a state abstract domain
Modular architecture

State abstract domain
- Solves pointer references
  - Points-to domain
- Checks type and memory safety
  - Types domain
  - Permissions domain
- Design, implemented and proved correct by V. Laporte
- Parameterized by a numerical abstract domain
  - concretizes to numerical environments
Modular architecture

Several numerical domains
- Intervals over $\mathbb{Z}$ and floats
- Arithmetical congruences
- Symbolic equalities
- Octagons
- Convex polyhedra
  - Contributed by Fouilhe, Boulmé and Périn (Verimag)
- Modular communication system using channels
Bounded **machine integers** analysis

- Wraparound when overflow
- Checks numerical errors
  - Division by 0
  - Undefined overflows
- Parameterized by an abstract domain for $\mathbb{Z}$
Example 1

```
const double sqrt_tab[128] = { ... };

int main() {
    double x = verasco_any_double();
    verasco_assume(0.3 < x && x < 0.7);
    double y = sqrt_tab[ (int)(x*128.) ];
    verasco_assert(0.54 < y && y < 0.84);
    return 0;
}
```
Example 1

```c
const double sqrt_tab[128] = { ... };

int main() {
    double x = verasco_any_double();
    verasco_assume(0.3 < x && x < 0.7);
    double y = sqrt_tab[(int)(x*128.)];
    verasco_assert(0.54 < y && y < 0.84);
    return 0;
}
```
Example 1

```c
const double sqrt_tab[128] = { ... };

int main() {
    double x = verasco_any_double();
    verasco_assume(0.3 < x && x < 0.7);
    double y = sqrt_tab[(int)(x*128.)],
    verasco_assert(0.54 < y && y < 0.84);
    return 0;
}
```
Example 1

```c
const double sqrt_tab[128] = { ... };

int main() {
  double x = verasco_any_double();
  verasco_assume(0.3 < x && x < 0.7);
  double y = sqrt_tab[(int)(x*128.)],
  verasco_assert(0.54 < y && y < 0.84);
  return 0;
}
```
Example 2

```c
int main() {
    int v0 = 0, v1 = 1;
    int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};

    for(int i = -5; i < 6; i++) {
        int in_range = (0 <= i && i <= 2);
        int some_other_computation = i*i+32;
        if(in_range)
            verasco_assert(*tab[2*i+1] == 1);
    }
    return 0;
}
```
```c
int main() {
    int v0 = 0, v1 = 1;
    int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};

    for(int i = -5; i < 6; i++) {
        int in_range = (0 <= i && i <= 2);
        int some_other_computation = i*i+32;
        if(in_range)
            verasco_assert(*((tab[2*i+1])) == 1);
    }
    return 0;
}
```
Example 2

```c
int main() {
    int v0 = 0, v1 = 1;
    int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};

    for(int i = -5; i < 6; i++) {
        int in_range = (0 <= i && i <= 2);
        int some_other_computation = i*i+32;
        if(in_range) {
            verasco_assert(*(tab[2*i+1]) == 1);
        }
    }
    return 0;
}
```

Symbolic propagation of conditions

Use of parity (congruence domain)
Example 2

```c
int main() {
    int v0 = 0, v1 = 1;
    int* tab[6] = {&v0, &v1, &v0, &v1, &v0, &v1};

    for(int i = -5; i < 6; i++) {
        int in_range = (0 <= i && i <= 2);
        int some_other_computation = i*i+32;
        if(in_range)
            verasco_assert(*(tab[2*i+1]) == 1);
    }
    return 0;
}
```

- **Symbolic propagation of conditions**
- **Complex memory access**
- **Use of parity (congruence domain)**
Example 3

```c
int src[10] = { ... };  
int dst[10];

int main() {
    int i_src = 0, i_dst = 9;
    while(i_dst >= 0) {
        dst[i_dst] = src[i_src];
        i_dst--; i_src++;
    }
    verasco_assert(i_src == 10);
    return 0;
}
```
Example 3

```c
int src[10] = { ... };  
int dst[10];

int main() {
    int i_src = 0, i_dst = 9;
    while(i_dst >= 0) {
        dst[i_dst] = src[i_src];
        i_dst--; i_src++;
    }
    verasco_assert(i_src == 10);
    return 0;
}
```

Octagons prove the relational invariant
`i_src + i_dst = 9` ...
Example 3

```c
int src[10] = { ... };  
int dst[10];

int main() {
    int i_src = 0, i_dst = 9;
    while(i_dst >= 0) {
        dst[i_dst] = src[i_src];
        i_dst--; i_src++;
    }
    verasco_assert(i_src == 10);
    return 0;
}
```

Octagons prove the relational invariant \( i_{\text{src}} + i_{\text{dst}} = 9 \) ...

... and deduce precise bounds for \( i_{\text{src}} \)
Modular interfaces between components

Operations:
- $T \cup \subseteq \triangledown$
- Transfer functions

With Coq specifications
- Concretization function
  \[ \gamma : \text{Abs} \rightarrow \text{Concr} \rightarrow \text{Prop} \]
- Soundness theorems
- Abstraction function $\alpha$ is not needed
  - No optimality result
Example of interface
State abstract domain

- **Concrete states:**

  \[
  \text{list}( \text{ident} \quad * \quad \text{env} \quad ) \quad * \quad \text{mem}
  \]

- **Operations**

  - **Standard abstract interpretation operators:**
    - \( \sqsubseteq : \text{abstate} \to \text{abstate} \to \text{bool} \)
    - \( \sqcup, \sqcap : \text{abstate} \to \text{abstate} \to \text{abstate} \)

  - **Abstract transfer functions:**
    - assign, store, assume, push_frame, pop_frame...
Example of specification
State abstract domain

\[ \gamma : \text{abstate} \to \text{concrete_state} \to \text{Prop} \]

- Specifications:
  - \( \forall a b, \gamma(a) \cup \gamma(b) \subseteq \gamma(a \sqcup b) \)
  - \( \forall x e a, \{\rho[x := v] \mid \rho \in \gamma(a) \land \rho \vdash e \Downarrow v\} \subseteq \gamma(\text{assign } x e a) \)

- Other domains have similar specifications
Methodology

- Programmed and proved correct in Coq.
- Extracted into OCaml, then compiled into an executable.
- No use of a posteriori validation.
  - Except for the third-party polyhedra domain (Farkas certificates).
- About 45000 lines of Coq.
  - Half proofs, half code & specs
Introduction

Overview of Verasco

Technical zoom: numerical abstract domains

Conclusions
Numerical abstract domains in Verasco

**Intervals**

\[ y \]

\[ x \in [l, h] \]

**Symbolic equalities**

\[ z \doteq y \ast y \]

\[ c \doteq (x < 1 \text{ && } 2 \leq y) \]

\[ (y \leq 2.5) \doteq \text{true} \]

**Congruences**

\[ x \mod M = R \]

**Octagons**

\[ y \]

\[ x \]

\[ \pm x \pm y \leq \text{Cst} \]

**Polyhedron**

\[ y \]

\[ x \]

\[ \sum_{i=0}^{n} \alpha_i x_i \leq \text{Cst} \]
Modularity of numerical abstract domains

Modular **communication** system between domain

- **Weakly reduced** product
  - Precise (vs. direct products) and **practical** (vs. reduced products)
  - All abstract domains share a **common interface**
  - A domain **pulls** information using **query channels**
  - A domain **pushes** information using **message channels**
  - Inspired from Astrée

- Easy to soundly add or deactivate an abstract domain
  - Flexible precision-performance tradeoff
  - Some abstract domains are essential (e.g., intervals) for precision
Octagons
Overview

- Very popular weakly relational domain
  - Originally by Miné from Astrée
- Inequalities of the form: $\pm x \pm y \leq \text{Cst}$
- Interval constraints expressible:
  
  $$x + x \leq 2h \quad -x - x \leq -2l$$

- Data structure: difference bound matrix $A_{xy}$ for $x, y$ signed variables:
  
  $$\forall x, y, x + y \leq A_{xy}$$
Octagons
Sparse algorithms

Usual algorithms for Octagons:
- Maintain a saturated set of constraints $\pm x \pm y \leq Cst$
- Problem:
  \[
  \begin{cases}
    x + x \leq A_{x\bar{x}} \\
    y + y \leq A_{y\bar{y}}
  \end{cases} \implies x + y \leq A_{x\bar{y}} \leq \frac{A_{x\bar{x}} + A_{y\bar{y}}}{2}
  \]
  $\implies$ Dense constraints even for interval bounds

Our algorithms for Octagons:
- **Weak** form of saturation
- **Maintain the sparsity** of the constraints
Octagons

Example

\[
\begin{pmatrix}
-x & +x & -y & +y & -z & +z \\
   0 &   &   &   &   &   \\
   &   &   &   &   &   \\
   &   &   &   &   &   \\
   &   &   &   &   &   \\
   &   &   &   &   &   \\
   &   &   &   &   &   \\
   &   &   &   &   &   \\
\end{pmatrix}
\]
Octagons

Example

Interval constraints:

\[ x \in [0, 1] \]
\[ y \in [1, 3] \]
\[ z \in [0, 2] \]
Octagons

Example

Strong saturation

$\begin{bmatrix}
-x & +x & -y & +y & -z & +z \\
+x & 0 & 2 & 0 & 4 & 1 & 3 \\
-x & 0 & 0 & -1 & 3 & 0 & 2 \\
+y & 3 & 4 & 0 & 6 & 3 & 5 \\
-y & -1 & 0 & -2 & 0 & -1 & 1 \\
+z & 2 & 3 & 1 & 5 & 0 & 4 \\
-z & 0 & 1 & -1 & 3 & 0 & 0 \\
\end{bmatrix}$

$\implies$ Dense matrix
Octagons

Example

\[
\begin{pmatrix}
-x & +x & -y & +y & -z & +z \\
+0 & 2 & 0 & 0 & 0 & 0 \\
-0 & 0 & 0 & 0 & 0 & 0 \\
-0 & 0 & 0 & 0 & 0 & 0 \\
+0 & 4 & 0 & 0 & 0 & 0 \\
-0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

Weak saturation
(nothing to do here)
Octagons

Example

Adding the constraint:

\[
\begin{pmatrix}
-x & +x & -y & +y & -z & +z \\
+2 & -1 & +2 & -2 & 0 & 0 \\
+2 & -y & +2 & -y & 0 & 0 \\
+4 & -z & +4 & -z & 0 & 0 \\
\end{pmatrix}
\]

\[x + y \leq 2\]
Octagons

Example

Weak saturation:

\[
\begin{align*}
&y + y \leq 4 \\
&y - x \leq 2 \\
&-y - x \leq -1 \\
&x - y \leq 0
\end{align*}
\]

\[
\begin{pmatrix}
-x & +x & -y & +y & -z & +z \\
+0 & 2 & 0 & 2 & 0 & 1 \\
-x & 0 & 0 & -1 & 2 & 0 \\
+y & 2 & 2 & 0 & 4 & 0 \\
-y & -1 & 0 & -2 & 0 & 0 \\
+z & 0 & 4 & 0 & 0 & 0 \\
-z & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

Still sparse
Octagons

Example

Weak saturation:

\[
\begin{align*}
&\begin{aligned}
\text{y + y} & \leq 4 \\
\text{y - x} & \leq 2 \\
\text{-y - x} & \leq -1 \\
\text{x - y} & \leq 0
\end{aligned}
\end{align*}
\]

Still sparse
Abstract domain of symbolic equalities

Motivating example

CompCert front-end transformation:

```plaintext
if(0 <= a && a < 10) {
  ...
}

⇒
if(0 <= a)
  tmp = a < 10;
else
  tmp = 0;
if(tmp) {
  ...
}
```

Need to reason on the relation between `a` and `tmp`. 
Abstract domain of symbolic equalities

- Two kinds of equalities:
  - $\text{var} \doteq \text{expr}$
  - $\text{expr} \doteq \text{bool}$

- Example:

```c
if(0 <= a) {
  tmp = a < 10;
  (0 <= a) \doteq \text{true}
  (0 <= a) \doteq \text{true}
  tmp \doteq a < 10
}
else {
  tmp = 0;
  (0 <= a) \doteq \text{false}
  (0 <= a) \doteq \text{false}
  tmp \doteq 0
}
if(tmp) {
  Unfolding tmp
  a \in [0, 9]
  tmp \doteq (0 <= a) ? a < 10 : 0
}
```
Introduction

Overview of Verasco

Technical zoom: numerical abstract domains

Conclusions
Final theorem

Definition \( \text{vanalysis \ prog} = \ldots \) iter \ldots

Theorem \( \text{vanalysis\_correct}: \)
\[
\forall \ prog \ res \ tr, \quad \text{vanalysis \ prog} = (res, \text{nil}) \rightarrow
\]
\[
\text{program\_behaves} (\text{semantics \ prog}) (\text{Goes\_wrong \ tr}) \rightarrow \text{False}.
\]
Conclusion

Proving correct a realistic static analyzer based on abstract interpretation is feasible

- Realistic, feature-rich language (C99)
- Advanced combination of abstract domains

By expliciting the proofs, we clarified:

- The specification of abstract domains
- The architecture of a static analyzer
- Their implementation
Future work

- New analysis techniques:
  - Better handling of control flow (trace partitioning, recursion, ...)
  - Other numerical abstract domains (linear filters, arithmetic-geometric progression, pentagons, ...)
  - Support for dynamic memory allocation

- Better performance
  - Faster abstract domains (arrays summarization, variable packing, ...)
  - Better tools for formally verified software

- Using the results of the analysis (Optimizations, other analyzers)

- Experimenting with industrial code
http://compcert.inria.fr/verasco/

Questions ?
Appendices
My contributions in Verasco

- In CompCert:
  - Parser (verified in Coq)
  - Floats (verified in Coq)

- Abstract interpreter
  - Dedicated program logic

- Most of the numerical domains
  - Handling of machine arithmetic
  - Symbolic equalities
  - Intervals
  - Linearization
  - Octagons
  - Communication channels
In my thesis...

- Design and proof of abstract iterator
  - Handles all C# minor control constructs
  - Proved using a dedicated program logic

- Sharing, hash-consing and memoization in Coq
  - Examplified on a BDD library
  - Applications in Verasco

- Contributions to CompCert
  - Parsing, support for floating-point numbers
Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>integr.c</td>
<td>42 lines</td>
<td>&lt; 0.1s</td>
</tr>
<tr>
<td>smult.c</td>
<td>330 lines</td>
<td>19.3s</td>
</tr>
<tr>
<td>nbody.c</td>
<td>179 lines</td>
<td>10.7s</td>
</tr>
<tr>
<td>almabench.c</td>
<td>352 lines</td>
<td>5.7s</td>
</tr>
</tbody>
</table>

- No alarms on those examples.
  - Pointers, arrays, floats
- Much room for improvement in performance.
Non-relational numerical abstract domains

Congruences & Intervals

**Intervals**
- On $\mathbb{Z}$ and floating-point numbers.
- Handles all arithmetic and bit-level operations of C99

**Arithmetical congruences**
- Needed to check alignment of memory accesses

**Specific interface for non-relationnal domains**
- Common relational adaptation layer
Complex control flow

x=0

loop {
    if x > 11
        break
    x+=2
}

6 of 8
Complex control flow

\{ T \}\n\begin{align*}
x &= 0 \\
\{x = 0\} \\
\text{loop } &\{ \\
\{x \in [0, 13] \land x \text{ mod } 2 = 0\} \\
&\quad \text{if } x > 11 \\
\{x = 12\} \\
&\quad \text{break} \\
\{x \in [0, 11] \land x \text{ mod } 2 = 0\} \\
&\quad x += 2 \\
\{x \in [2, 13] \land x \text{ mod } 2 = 0\} \\
\}\{x = 12\} \\
\end{align*}

- Based on control points
- We want definitions following the AST structure
Complex control flow

x=0

loop {

if x > 11
{x = 12}
break
{⊥, x = 12}
}

Dedicated program logic:

\{P\} s \{Q, Q_b\}

- Several postconditions
- Defined structurally
x=0

loop {
  \{ x \in [0, 13] \land x \text{ mod } 2 = 0 \}
  if x > 11
    break
  x+=2
  \{ x \in [2, 13] \land x \text{ mod } 2 = 0, \ x = 12 \}
}

Dedicated program logic:

\{ P \} s \{ Q, \ Q_b \}

- Several postconditions
- Defined structurally
Complex control flow

x=0
{x = 0}
loop {
  if x > 11
    break
  x+=2
}
{x = 12, ⊥}

Dedicated program logic:
{P} s { Q, Q_b }

- Several postconditions
- Defined structurally
Abstract interpreter
Simplified implementation & proof

\[
\text{Fixpoint } \text{iter (ab:abstate) (s:stmt) \{struct s\}} \\
: \text{abstate} \ast \text{abstate} := ... \\
\]

- Proof step 1: interpreter soundness

  \textbf{Lemma iter\_ok: } \forall \text{ab\_pre stmt ab\_post ab\_break}, \\
  \text{iter ab\_pre stmt} = (\text{ab\_post}, \text{ab\_break}) \rightarrow \\
  \{ \gamma(\text{ab\_pre}) \} \text{ stmt} \{ \gamma(\text{ab\_post}), \gamma(\text{ab\_break}) \}.

- Proof step 2: program logic soundness

  \{\cdot\} \cdot \{\cdot,\cdot\} \implies \text{No undefined behavior}
Abstract interpreter

Big picture

- Handles all C#minor control constructs
  - Infinite loop, break, if/else, switch, goto, call, return
  - 2 pre-conditions, 4 post-conditions
- Parameterized by a state abstract domain
- Works in a monad for alarms
- Structural recursion on syntax tree
  - Unfolding functions at call sites
- Fixpoint iteration using widening and narrowing
  - One fixpoint iteration per loop
  - Gotos: one fixpoint iteration per function