Verification Techniques for Low-Level Programs

Samuel D. Pollard

University of Oregon
Computer and Information Science

February 21, 2019
Outline

1. Introduction
2. Formal Methods in Practice
3. Intermediate Representations
4. Floating Point Arithmetic
5. SIMD Parallelism
6. Conclusion
Why These Three Topics?

Mathematics
- category theory
- real analysis
- numerical analysis

Parallelism
- distributed systems
- shared memory parallelism
- vectorization

Programming
- high-level language
- intermediate representation
- assembly

more abstraction

§ 3

§ 4

§ 5
1 Introduction

2 Formal Methods in Practice

3 Intermediate Representations

4 Floating Point Arithmetic

5 SIMD Parallelism

6 Conclusion
What is Formal Methods? (In 5 Seconds)

Using math to make sure your computer does what it’s supposed to do
What is Formal Methods?

- The application of theoretical computer science, math, type theory, logic, and even philosophy to computer programs

- Formal methods is like a coin with two sides
  1. Designing a machine-readable and machine-checkable formal specification
     - English is always informal
     - e.g. logical formula, program in Coq, or even the type of a function
  2. Ensuring software obeys this specification.
     - {Type, model, satisfiability, proof} checker

- In math, the parallel is theorem and proof.
  - This parallel is a very deep result of computer science
Why Do We Need Formal Methods?

- Therac-25 - race condition gave 100× radiation dosage
- Patriot Missile bug - accumulating timing error killed 28
Liar’s Paradox: “This statement is false.”
The issue lies with self-reference
Analogue in λ-calculus is Curry’s Paradox
Simply Typed Lambda Calculus (STLC)

- By Alonzo Church in 1940
- Invented to prevent paradoxes of untyped $\lambda$-calculus
- STLC is total so is not Turing Complete

STLC Grammar

\[
t ::= x \mid t \cdot t \mid \lambda x : \tau. \ t
\]
\[
v ::= \lambda x : \tau. \ t
\]
\[
\tau ::= \text{bool} \mid \text{int} \mid \tau \rightarrow \tau
\]
\[
\Gamma ::= \emptyset \mid \Gamma, x : \tau
\]

E.g. type differently `fst` and `True`

\[
(\lambda x : \tau. \ \lambda y : \tau. \ x) : \text{bool}
\]
Propositional Logic: $\iff, \land, \lor, \implies, \neg$
- $\mathcal{NP}$-Complete: given a formula, we can determine satisfiability in at most exponential time
- Rules of chess in 100,000 pages

First-Order Logic: $\forall, \exists$
- Undecidable: Given a formula, we cannot in general determine satisfiability
- Rules of Chess in 1 page
Canonical Verification Techniques

- Hoare Logic
- Satisfiability Modulo Theories (SMT)
- Abstract interpretation
Hoare Logic

- Has the form $\{P\} Q \{R\}$
- Specify predicate transformer semantics
- Compute weakest precondition/strongest postcondition

\[ \{x > 5\} \ x := x \times 2 \ \{x > 10\} \]
Satisfiability Modulo Theories (SMT)

- Propositional logic + theory of ________________
- Matrix Multiplication of formal methods

- Satisfiable (Integers):
  \[ x + 2y = 20 \land x - y = 2 \]

- Unsatisfiable (Integers):
  \[(x > 0) \land (y > 0) \land (x + y < 0)\]

- Unsatisfiable (Float) but long runtime
  \[(-2 \leq x \leq 2) \land (-2 \leq y \leq 2) \land (-1 \leq z \leq 1) \land (x \leq y) \land (y + z < x + z)\]
Sound static analysis
- Static analysis - discovering properties of a program without executing it
- Sound - no false negatives

3. Basic GC semantics  Basic GCs are primitive abstractions of properties. Classical examples are the identity abstraction $S[\bot[\langle C, \sqsubseteq \rangle]] \triangleq \langle C, \sqsubseteq \rangle \iff \lambda \bullet Q : Q \rightarrow \langle C, \sqsubseteq \rangle$, the top abstraction $S[T[\langle C, \sqsubseteq \rangle, T]] \triangleq \langle C, \sqsubseteq \rangle \iff \lambda \bullet Q : T \rightarrow \langle C, \sqsubseteq \rangle$, the join abstraction $S[\sqcup[C]] \triangleq \langle \varphi(C), \subseteq \rangle \iff \lambda \alpha \beta : P \sqcap T \rightarrow \langle \varphi(C), \subseteq \rangle$ with $\alpha P \sqcup P$, $\varphi(Q) \triangleq \lambda \beta : T \rightarrow \langle \varphi(C), \subseteq \rangle$ the complement abstraction $S[-[C]] \triangleq \langle \varphi(C), \subseteq \rangle \iff \lambda \beta : T \rightarrow \langle \varphi(C), \subseteq \rangle$, the finite/infinite sequence abstraction $S[\infty[C]] \triangleq \langle \varphi(C), \subseteq \rangle \iff \lambda \beta : T \rightarrow \langle \varphi(C), \subseteq \rangle$ with $\alpha\infty(P) \triangleq \{ \sigma_i | \sigma \in P \land i \in \text{dom}(\sigma) \}$ and $\gamma\infty(Q) \triangleq \{ \sigma \in C \land \forall i \in \text{dom}(\sigma) : \sigma_i \in Q \}$, the transformer abstraction $S[\mapsto[C_1, C_2]] \triangleq \langle \varphi(C_1 \times C_2), \subseteq \rangle \iff \lambda \beta : T \rightarrow \langle \varphi(C_1) \sqcup \varphi(C_2), \subseteq \rangle$ mapping relations to join-preserving transformers with $\alpha\mapsto(R) \triangleq \lambda X \cdot \{ y | \exists x \in X : \langle x, y \rangle \in R \}$, $\gamma\mapsto(g) \triangleq \{ \langle x, y \rangle | y \in g(\{x\}) \}$, the function abstraction $S[\mapsto[C_1, C_2]] \triangleq \langle \varphi(C_1 \mapsto C_2), \subseteq \rangle \iff \lambda \beta : T \rightarrow \langle \varphi(C_1) \mapsto \varphi(C_2), \subseteq \rangle$ with $\alpha\mapsto(P) \triangleq \lambda X \cdot \{ f(x) | f \in P \land x \in X \}$, $\gamma\mapsto(g) \triangleq \{ f \in C_1 \mapsto C_2 | \forall X \in \varphi(C_1) : \forall x \in X : f(x) \in g(X) \}$.
Abstract Interpretation

```c
float unsafe(float x) {
    if (x==0.0)
        return 0.0;
    else
        return 1.0 / x;
}
```

- Create an abstract domain
- Define semantics on abstract domain
- “Interpret” your program to see what values pop out
  - For example, we wish to ensure we never divide by zero.
Abstract Interpretation

```c
float unsafe(float x) {
    if (x==0.0)
        return 0.0;
    else
        return 1.0 / x;
}
```

<table>
<thead>
<tr>
<th>Abstract Domain:</th>
<th>Handled?</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnz / 0</td>
<td>±∞</td>
</tr>
<tr>
<td>nnz / flt</td>
<td></td>
</tr>
<tr>
<td>NaN / NaN</td>
<td></td>
</tr>
<tr>
<td>±∞ / 0</td>
<td></td>
</tr>
<tr>
<td>flt / flt</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Interpretation

```c
#include <math.h>
float mostlysafe(float x) {
    if (x==0. || isnan(x) || isinf(x))
        return 0.;
    else
        return 1. / x;
}
```
Abstract Interpretation

```c
#include <math.h>
float mostlysafe(float x) {
    if (x==0. || isnan(x) || isinf(x))
        return 0.;
    else
        return 1. / x;
}
```

<table>
<thead>
<tr>
<th>Abstract Domain:</th>
<th>=</th>
<th>Handled?</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnz /</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>±∞</td>
<td>✔️</td>
</tr>
<tr>
<td>nnz</td>
<td>flt</td>
<td>X</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>✔️</td>
</tr>
<tr>
<td>±∞</td>
<td>0</td>
<td>✔️</td>
</tr>
<tr>
<td>flt</td>
<td>flt</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Interpretation

```c
#include <math.h>
float reallysafe(float x) {
    // Cast to int without changing bits
    unsigned long c = *(unsigned long*) &x;
    if (isnan(x) || isinf(x) ||
        (0x80000000 <= c && c <= 0x80200000) ||
        (0x00000000 <= c && c <= 0x00200000))
        return 0.;
    else
        return 1. / x;
}
```

<table>
<thead>
<tr>
<th>Abstract Domain:</th>
<th>nnz /</th>
<th>=</th>
<th>Handled?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>±∞</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>nnz</td>
<td>flt</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>±∞</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>flt</td>
<td>flt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 Introduction

2 Formal Methods in Practice

3 Intermediate Representations

4 Floating Point Arithmetic

5 SIMD Parallelism

6 Conclusion
Design Decisions

1. To what degree of confidence must the software be guaranteed?
2. What tools can be used to accomplish 1?
3. How much time can a human spend on 2? How much time can a computer spend on 2?
Degree of Confidence

- Formal methods is not a silver bullet
  - More like a flu vaccine
- Security vulnerability found in the WPA2 WiFi standard which had been proven secure
- Vulnerability related to temporal property which WPA2 did not specify

“Beware of bugs in the above code; I have only proved it correct, not tried it” — Donald Knuth
Properties You Might Care About

- **Basic safety guarantees**
  - Will \texttt{ipow} work for all integers?
  - Can be handled (mostly) automatically by SMT solvers

```c
int ipow(int x, int n) {
    if (n==0) return 1;
    return x * ipow(x,n-1);
}
int main() {
    int a,b,c,n;
    n = 3; c = 0;
    while (1) {
        c++;
        for (a = 1; a < c; a++) {
            for (b = 1; b < c; b++) {
                if (ipow(a,n)+ipow(b,n)==ipow(c,n))
                    return 0;
            }
        }
    }
}
```
Properties You Might Care About

▶ Does this program test all combinations of a, b, and c?
  - Requires loop invariants and annotating code
  - Proves partial correctness; if the code terminates, then we get the right answer

```c
int ipow(int x, int n) {
    if (n==0) return 1;
    return x * ipow(x,n-1);
}

int main() {
    int a,b,c,n;
    n = 3; c = 0;
    while (1) {
        c++;
        for (a = 1; a < c; a++) {
            for (b = 1; b < c; b++) {
                if (ipow(a,n)+ipow(b,n)==ipow(c,n))
                    return 0;
            }
        }
    }
}
```
Properties You Might Care About

Does this program terminate?

- Until 1995, no one knew

```c
int ipow(int x, int n) {
    if (n==0) return 1;
    return x * ipow(x,n-1);
}

int main() {
    int a,b,c,n;
    n = 3; c = 0;
    while (1) {
        c++;
        for (a = 1; a < c; a++) {
            for (b = 1; b < c; b++) {
                if (ipow(a,n)+ipow(b,n)==ipow(c,n))
                    return 0;
            }
        }
    }
}
```
Tools for Formal Methods

Nadia Polikarpova
@polikarn

software engineer: linear is fast, quadratic is slow
complexity theorist: P is fast, NP-hard is slow
verification researcher: decidable is fast, undecidable is slow

8:47 AM - 13 Dec 2018

378 Retweets 1,441 Likes
Tools for Formal Methods

- Tradeoff between automation and strength of what can be verified
- What prevents us from reaching the Holy Grail is the fundamental limit of computation
- Rice's Theorem
## Notable Proof Assistants

<table>
<thead>
<tr>
<th>System</th>
<th>de Bruijn criterion</th>
<th>Tactics</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq</td>
<td>✓</td>
<td>✓</td>
<td>Coq - Proved the four color theorem</td>
</tr>
<tr>
<td>HOL</td>
<td>✓</td>
<td>✗</td>
<td>HOL/Isabelle - Intel and Cambridge</td>
</tr>
<tr>
<td>ACL2</td>
<td>✗</td>
<td>✓</td>
<td>ACL2 - UTexas’, also (probably) AMD</td>
</tr>
<tr>
<td>PVS</td>
<td>✓</td>
<td>✓</td>
<td>PVS - Used at NASA</td>
</tr>
<tr>
<td>Twelf</td>
<td>✗</td>
<td>✗</td>
<td>F*, Lean - Microsoft</td>
</tr>
<tr>
<td>F*</td>
<td>✓</td>
<td>✓</td>
<td>NuPRL - Cornell; oldest listed here; 1984</td>
</tr>
<tr>
<td>NuPRL</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Agda</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Lean</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
The Martian

```
(lemma both_sides_times_pos_le1
   ("pz" "px\:1" "x" "1" "y" "expt(1+px\:1,jl1)")
   (("2" (rewrite "expt_gt1_bound1" -1)
     (("2" (assert nil nil)) nil))
    nil))
   nil))
   nil))
   nil))
   nil)
   nil)
---
```
1 Introduction
2 Formal Methods in Practice
3 Intermediate Representations
4 Floating Point Arithmetic
5 SIMD Parallelism
6 Conclusion
Verifying IRs

- WebAssembly - Machine-checkable semantics
- Vellvm - LLVM in Coq
- BAP - Binary analysis platform
1. Introduction
2. Formal Methods in Practice
3. Intermediate Representations
4. Floating Point Arithmetic
5. SIMD Parallelism
6. Conclusion
Some Quirks of Floating Point Arithmetic

- Tension between “floats as bits” and “floats as reals”
- Having the same bit pattern is neither necessary nor sufficient for two IEEE 754 floats to be considered equal
  - 0...0 and 10...0 represent −0 and +0 which are equal
  - NaNs are all not equal to each other
Formalizing Real Numbers

- Coq library Flocq
- Doesn’t concern itself with bit-level representations
- No maximum exponents!
1 Introduction
2 Formal Methods in Practice
3 Intermediate Representations
4 Floating Point Arithmetic
5 SIMD Parallelism
6 Conclusion
Why is This Challenging?

- Increasing vector widths means more complex CFG, difficult to automatically (and hand) vectorize
- Code transformations may assume associativity (too aggressive)
- Code transformations may match scalar code (too conservative)
1 Introduction
2 Formal Methods in Practice
3 Intermediate Representations
4 Floating Point Arithmetic
5 SIMD Parallelism
6 Conclusion
Formal Methods are two things:

1. **Formal Specification** - creating unambiguous, computer-checkable description of the program
2. **Model Checking** - proving (or refuting) the program obeys this spec.

Theoretically many of FM algorithms are undecidable or at least \( \mathcal{NP} \)-Hard

- In practice these scale surprisingly well
- i.e. programs terminate or scale beyond \( n = 35 \)
A complaint of formal methods is it’s too expensive

FM isn’t all-or-nothing

- Consider a type checker
- Static analyzers can be part of software engineering workflow
- e.g. CI, red squiggly lines in Eclipse

Ariane 5 - $500 million software bug: incorrectly converted 64-bit float to 16-bit integer
ASKING AIRCRAFT DESIGNERS ABOUT AIRPLANE SAFETY:
NOTHING IS EVER FOOLPROOF, BUT MODERN AIRLINERS ARE INCREDIBLY RESILIENT. FLYING IS THE SAFEST WAY TO TRAVEL.

ASKING BUILDING ENGINEERS ABOUT ELEVATOR SAFETY:
ELEVATORS ARE PROTECTED BY MULTIPLE TRIED-AND-TESTED FAISAFE MECHANISMS. THEY'RE NEARLY INCAPABLE OF FALLING.

ASKING SOFTWARE ENGINEERS ABOUT COMPUTERIZED VOTING:
THAT'S TERRIFYING.

WAIT, REALLY?
DON'T TRUST VOTING SOFTWARE AND DON'T LISTEN TO ANYONE WHO TELLS YOU IT'S SAFE.

WHY?
I DON'T QUITE KNOW HOW TO PUT THIS, BUT OUR ENTIRE FIELD IS BAD AT WHAT WE DO, AND IF YOU RELY ON US, EVERYONE WILL DIE.

THEY SAY THEY'VE FIXED IT WITH SOMETHING CALLED "BLOCKCHAIN."
AAAA!!!
WHATEVER THEY SOLD YOU, DON'T TOUCH IT.
BURY IT IN THE DESERT.
WEAR GLOVES.
Future Work

- Formalizing IEEE-754, MIL-STD-1750A, and Posits in a unified way
  - Look at floats both as bits and real numbers
  - Existing packages do only one (Flocq: \( \mathbb{R} \), SMT Solvers: bits)
- Existing scientific codes only flip one switch - to use float or double; can we do better?
- SIMD transformations are rigid, GCC -O3 is not rigid enough
- Quameleon - multi-ISA binary analysis at Sandia