Floating Point Verification

Unifying Abstract Interpretation and Decision Procedures

Leopold Haller
(joint work with Vijay D’Silva, Michael Tautschnig, Daniel Kroening)

2nd November 2011
Presentation Outline

Part I

Existing approaches to FP - Verification

- Manual, Semi-automated
- Decision Procedures
- Abstract Interpretation

Part II

Decision Procedures

- Precise
- Scalable
- Abstract Interpretation

Our research

Abstract Satisfiability
Part I
IEEE754 Floating Point Numbers

Special values: \(-0, +0, -\infty, \infty, NaN\)
The Pitfalls of FP

I

\[
\begin{align*}
\text{if}(x < y) \\
\ldots \\
\text{else if}(x > y) \\
\ldots \\
\text{else assert}(x == y);
\end{align*}
\]

II

\[
\begin{align*}
\text{if}(x > 0) \\
\{ \\
\text{for}(\text{float sum} = 0; \text{sum} \leq N; \text{sum} += x) \\
\ldots \\
\text{//does the loop terminate?}
\}
\end{align*}
\]

III

\[
\begin{align*}
\text{float } r1 &= a+b; \\
\text{float } r2 &= b+c; \\
\text{r1} &= c; \text{r2} += a; \\
\text{assert}(r1 == r2);
\end{align*}
\]

IV

\[
\begin{align*}
\text{float } r1 &= a+b; \\
\text{float } r2 &= a+b; \\
\text{assert}(r1 == r2);
\end{align*}
\]

V

\[
\begin{align*}
\text{bool } b &= \text{false}; \\
\text{if}(f < 1) \\
\quad b &= \text{true}; \\
\text{if}(!b) \\
\quad \text{assert}(f \geq 1);
\end{align*}
\]
Is this program correct?
What does correctness mean?

Three possible meanings:

• Result is sufficiently close to the real number result

• Result is sufficiently close to the sine function

• The assertion cannot be violated
How can we check correctness?

- Manual
- Abstract Interpretation
- Decision Procedures
Manual, semi-automated

- Use an interactive theorem prover
- Experts write proof scripts with machine assistance
- Potentially powerful, but expensive
- Proof scripts require expert understanding, may be much harder to write than programs
Manual, semi-automated

User enters proof

Computer keeps track of what is left to prove
Manual, semi-automated

This proof file has been written by Sylvie Boldo(1), following a proof presented by Pr William Kahan (2), and adapted to Coq proof checker with the help of Guillaume Melquiond(1) and Marc Daumas(1). This work has been partially supported by the CNRS grant PICS 2533.

(1) LIP Computer science laboratory UMR 5668 CNRS - ENS de Lyon - INRIA Lyon, France
(2) University of California at Berkeley Berkeley, California

Section Discriminant1.
Variable bo : Fbound.
Variable precision : nat.
Let radix := 2%Z.
Let FtoRadix := FtoR radix.
Coercion FtoRadix : float -> R.

Theorem TwoMoreThanOne : (1 < radix)%Z.

Let radixMoreThanZERO := 2lt_1_0 _ (2lt_le_weak _ TwoMoreThanOne).
Hypothesis precisionGreaterThanOne : 1 < precision.
Hypothesis pGivesBound : Πpos (vNum bo) = Zpower_nat radix precision.

Variables a b b' c p q d : float.

Let delta := (Rabs (d-(b*b'-a*c)))%R.

Hypothesis Fa : (Fbounded bo a).
Hypothesis Fb : (Fbounded bo b).
Hypothesis Fb' : (Fbounded bo b').
Hypothesis Fc : (Fbounded bo c).
Hypothesis Fp : (Fbounded bo p).
Hypothesis Fq : (Fbounded bo q).
Hypothesis Fd : (Fbounded bo d).

There is no underflow

Hypothesis U1: (-dExp bo <= Fexp d - 1)%Z.
Hypothesis Nd: (Fnormal radix bo d).
Hypothesis Nq: (Fnormal radix bo q).
Hypothesis Np: (Fnormal radix bo p).
Hypothesis Square: (0 <= b*b')%R.

Hypothesis Roundp : (EvenClosest bo radix precision (b*b')%R p).
Hypothesis Roundq : (EvenClosest bo radix precision (a*c)%R q).
Hypothesis Firstcase : (p+q <= 3*(Rabs (p-q)))%R.
Hypothesis Roundd : (EvenClosest bo radix precision (p-q)%R d).

Theorem delta_inf : (delta <= ((2)*(Fulp bo radix precision d)+((2)*(Fulp bo radix precision p))%(2)*(Fulp bo radix precision q)))%R.

Theorem P_positive : (Rle 0 p)%R.

Theorem Fulp_le_twice_1 : forall x y:float, (0 <= x)%R ->
(Fnormal radix bo x) -> (Fbounded bo y) -> (2*x<y)%R ->
((2*(Fulp bo radix precision x)) <= (Fulp bo radix precision y))%R.

Theorem Fulp_le_twice_r : forall x y:float, (0 <= x)%R ->
(Fnormal radix bo y) -> (Fbounded bo x) -> (x<=2*y)%R ->
((Fulp bo radix precision x) <= 2*(Fulp bo radix precision y))%R.

Theorem Half_Closest_Round: forall (x:float) (r:R),
(- dExp bo <= Zpred (Fexp x))%Z -> (Closest bo radix r x)
-> (Closest bo radix (r/2)%R (Float (Fnum x) (Zpred (Fexp x))))%R.

Theorem Twice_EvenClosest_Round: forall (x:float) (r:R),
(-dExp bo <= (Fexp x)-1)%Z -> (Fnormal radix bo x)
-> (EvenClosest bo radix precision r x)
-> (EvenClosest bo radix precision (2*r)%R (Float (Fnum x) (Zsucc (Fexp x))))%R.

Theorem EvenClosestMonotone2: forall (p q : R) (p' q' : float),
(p <= q)%R -> (EvenClosest bo radix precision p p') ->
(EvenClosest bo radix precision q q') -> (p' <= q)%R.

Theorem Fulp_le_twice_r_round: forall (x y:float) (r:R), (0 <= x)%R ->
(Fbounded bo x) -> (Fnormal radix bo y) -> (- dExp bo <= Fexp y - 1)%Z
-> (x<=2*r)%R ->
(EvenClosest bo radix precision r y) ->
((Fulp bo radix precision x) <= 2*(Fulp bo radix precision y))%R.

Theorem descril: (delta <= 2*(Fulp bo radix precision d))%R.
End Discriminant1.
Selection of notable work:

- John Harrison (Intel) - Verification of FP hardware and firmware using HOL

- Various formalizations of IEEE754 FP arithmetic for different theorem provers

- Boldot, Filliâtre, Melquiond et. al. - Theorem prover combined with incomplete FP prover.
Conclusion:

• Manual or semi-automated techniques can be very powerful, but require experts and large time investments.

• Results have limited reusability.

• Typically feasible for small system components of critical importance (e.g., Intel’s verification of processor components).
# References

## Axiomatisations of FP


J. Harrison. A machine-checked theory of floating-point arithmetic. TPHOLs 1999

## Specification of FP properties


S. Boldo and J.C. Filliâtre. Formal verification of floating-point programs. ARITH 2007
References

Applications

J. Harrison. Floating point verification in HOL light: The exponential function. FMSD, 16(3), 2000

J. Harrison. Floating-point verification. FM 2005

J. Harrison. Formal verification of square root algorithms. FMSD, 22(2), 2003


R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. TPHOLs 2000


Requires experts, expensive, powerful

Manual

Abstract Interpretation  Decision Procedures
Abstract Interpretation

- Instead of exploring all executions, explore a single abstract execution
- Abstract execution contains all concrete executions!
- Highly efficient and scalable, but imprecise

Program traces → Abstract representation → Program is safe

Error states do not overlap abstract representation, hence program is safe
An abstract interpreter modularly uses operations provided by an abstract domain. Changing the domain changes the analysis.

Example

<table>
<thead>
<tr>
<th>Signs domain</th>
<th>Constants domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>{+, -} ∪ {?}</td>
<td>{c</td>
</tr>
</tbody>
</table>

```c
float y = 5;
if(x > 0) {
  float z = x*y;
  assert(z > 0);
}
```

<table>
<thead>
<tr>
<th>Result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe!</td>
<td>Possibly unsafe</td>
</tr>
</tbody>
</table>
An abstract interpreter modularly uses operations provided by an abstract domain. Changing the domain changes the analysis.

Example

```
int x, y;
if (y < 0) {
    x = y;
} else {
    y++;  
    x = 5;
}
assert(x < 6);
```

**Interval Domain**

\[
\{ [l, u] \mid l, u \in \text{Int} \}
\]

- \(x, y \in [\min(\text{Int}), \max(\text{Int})]\)
- \(x, y \in [\min(\text{Int}), -1]\)
- \(x \in [5, 5], y \in [\min(\text{Int}), \max(\text{Int})]\)
- \(x \in [\min(\text{Int}), 5], y \in [\min(\text{Int}), \max(\text{Int})]\)
Floating Point Intervals

\[ \left\{ [l, u] \mid l, u \in FP \right\} \cup \{?\} \]
Astrée Abstract Interpreter

- Mature abstract interpreter by Cousot et. al
- Large number of domains
- Sold and supported by Absint GmbH
- Successful in proving correct large avionics control software: 100k lines of code in 1h -> highly scalable
- Various domains for floating point analysis:

Original traces  Ellipses  Octagons  Intervals
Abstract Domains for Floating Point

• Abstract domains are typically formulated over the real or rational numbers.

• Numeric domains rely on mathematical properties such as associativity which do not hold over floating point numbers:

\[(a + b) + c = a + (b + c)\]

• Solution (Minet 2004): Interpret operations over floating point numbers as real number operations + error terms.

```c
double d;
float f1, f2;

f1 = (float) d;
f2 = f1 * f2;
```

```c
real d;
real f1, f2;

f1 = d + round_error(FLOAT_CAST, d);
f2 = f1 * f2 + round_error(FLOAT_MULT, f1, f2);
```
Fluctuat: Errors as First Class Citizens

- Static analyser built for FP precision analysis
- Idea: Keep track separately of three distinct values for each variable

\[(f^x, r^x, e^x)\]

FP value  Real value  FP error

- Abstract these values separately

```c
float x;
if(*) x = 1f;
else x = 0f;
```

\([1, 1], [1, 1], [0, 0]\)
\([0, 0], [0, 0], [0, 0]\)
\([0, 1], [0, 1], [0, 0]\)

FP and real value are imprecise, but there is no rounding error
Fluctuat: Tracking errors with Zonotopes

- Fluctuat uses zonotope abstractions which combines intervals with noise symbols

```c
void foo(float x, bool b)
{
    if(!(x <= 1 && x >= 0))
        return;
    float y = 1.0f + 2*x;
}
```

Noise variables take values in $[0, 1]$

$x = \varepsilon_1$  \hspace{1cm} (\equiv x \in [0, 1])$

$y = 2 \times \varepsilon_1 + c \times \varepsilon_2 + 1.0f$

Relation to $x$ is preserved  \hspace{1cm} new error symbol models rounding error

- The source of imprecisions can be precisely traced
Imprecision in Abstract Interpretation

- The efficiency of abstract interpreters comes at the cost of precision. Imprecision is accumulated from three sources:

  - **Statements**

    \[
    x \in [-5, 5] \quad y = x * x; \quad y \in [-25, 25]
    \]

    \[
    x \in [0, 1] \quad y = x; \quad x, y \in [0, 1]
    \]

  - **Control-flow**

    \[
    \text{if}(y < 0) \quad x = 1; \quad x \in [-1, 1]
    \]

    \[
    \text{else} \quad x = -1;
    \]

  - **Loops**

    \[
    x, y \in [1, 1] \quad \text{while}(x < 100000) \quad x \in [100001, \text{max}(Int)]
    \]

    \[
    \{ \quad x++; \quad y++; \quad \}
    \]

    \[
    y \in [\text{min}(Int), \text{max}(Int)]
    \]
Imprecision in Abstract Interpretation

- For efficiency reasons, most numeric abstract domains are convex
Imprecision in Abstract Interpretation

What if convex abstractions are too weak?

Very common scenario

```plaintext
if(*)
    x = 1;
else
    x = -1;
assert(x != 0);
```
Handling Imprecision

What happens if the analysis is imprecise?

Customer

sends code

sends manually created configuration

AbsInt GmbH

manually creates new abstract domain

Researcher

Error

Error

Error
Conclusion:

- Very scalable
- Imprecise
- Precise results require experts and research effort
- Expert created domains are moderately reusable
- Feasible for programs with homogenous structure and behaviour (success in avionics)
References

Floating point abstract domains

A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. SAS 2010

L. Chen, A. Miné and P. Cousot. A sound floating-point polyhedra abstract domain. APLAS 2008

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. ESOP 2004

L. Chen, A. Miné, J. Wang and P. Cousot. An abstract domain to discover interval Linear Equalities. VMCAI 2010


K. Ghorbal, E. Goubault and S. Putot. The zonotope abstract domain Taylor I. CAV 2009

B. Jeannet, and A. Miné. Apron: A library of numerical abstract domains for static analysis. CAV 2009

D. Monniaux. Compositional analysis of floating-point linear numerical filters. CAV 2005

J. Feret. Static analysis of digital filters. ESOP 2004


E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. NSAD 2005

E. Goubault. Static analyses of the precision of floating-point operations. SAS 2001
References

Industrial Case Studies

E. Goubault, S. Putot, P. Baufreton, J. Gassino. Static analysis of the accuracy in control systems: principles and experiments. FMICS 2007


J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. SAFECOMP 2007

J. Souyris. Industrial experience of abstract interpretation-based static analyzers. IFIP 2004

P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. EMSOFT 2007

FP Static Analysers


P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and Xavier Rival. The ASTREÉ analyzer. ESOP 2005

E. Goubault, M. Martel and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. ESOP 2002
Requires experts, expensive, powerful

---

**Manual**

---

**Abstract Interpretation**

- Scalable and efficient.
- Precise analysis requires experts

**Decision Procedures**
• Precisely explore a large set of program traces

• For efficiency, represent problem *symbolically* as satisfiability of a logical formula

Program traces

Program is safe exactly if \( isTrace(t) \land error(t) \) is satisfied by some \( t \)
Propositional formula: \( \varphi = (a \lor \neg b) \land (\neg a \lor b) \land \neg b \)

Is there an assignment to a,b that makes the formula true?

Decrease in SAT solving time for SAT algorithms 2000-2007
Why are SAT solvers so efficient

- SAT solvers learn from failure
- SAT solvers spot relevance
Example

```c
int foo(int a, int b, bool c)
{
    int result;
    if(c)
        result = a/b;
    else
        result = a*b;
    if(a>0 && b>0)
        assert(result >= 0);
}
```

\[
c \rightarrow (r = a/32b) \\
\wedge \neg c \rightarrow (r = a *_{32} b) \\
\wedge a > 0 \wedge b > 0 \wedge r < 0
\]

Can be translated to *propositional logic* using divider and multiplier circuits.

The formula evaluates to true under the following assignment:

\[
a, b \mapsto 123456789 \\
r \mapsto -1757895751 \\
c \mapsto \text{false}
\]

Counter-example!
Loops require unrolling before translation

```c
int foo(int *a)
{
    int sum;
    int i = 0;
    if (i < N)
    {
        sum += a[i];
        if (++i < N)
        {
            sum += a[i];
            if (++i < N)
            {
                ...
            }
        }
    }
    assert(sum > 0);
    return sum;
}
```

If the loop does not have a known fixed bound, the result is unrolled up to a chosen depth.
Bounded Model Checking

**CBMC**

- C/C++ Source
- parse
- parse tree
- CFG
- unwind
- formula
- flattening
- CNF
- SMT AUFBV

**Decision Procedure**

- Unsatisfiable
- Satisfiable

Program has bug, *counter-example* is returned?
FP support in CBMC (2008)

- CBMC implements **bit-precise reasoning** over floating-point numbers using a propositional encoding
- Uses IEEE-754 semantics with support for various rounding-modes
- Allows proofs of **complex, bit-level** properties

```c
int main()
{
    union {
        int i;
        float f;
    } u;

    u.f += u.i + 1;
    assert(u.i != 0);
}
```
Scalability of Propositional Encoding

- Floating-point arithmetic is flattened to propositional logic
- Requires instantiation of large floating point arithmetic circuits

```
for(int i = 0; i < N; i++)
{
    f *= f;
}
```

<table>
<thead>
<tr>
<th>N</th>
<th>Nr. Variables</th>
<th>Memory use</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>~130000</td>
<td>~90MB</td>
</tr>
<tr>
<td>10</td>
<td>~260000</td>
<td>~180MB</td>
</tr>
</tbody>
</table>

- Resulting formulas are hard for SAT solvers and take up large amounts of memory
• Use propositional abstraction to increase efficiency and ease memory requirements

• Novel *mixed abstraction framework*
  
  • Over-approximations allow more behaviours: Reduce the initial number of variables. Eases memory requirements and improves efficiency.
  
  • Under-approximations restrict behaviours: Allows us to quickly identify solutions.

• Integrated with CBMC and the Boolector SMT solver
Mixed Abstractions for Floating Point Arithmetic (2009)

![Diagram](image)

**Fig. 4. The Framework of Mixed Abstraction**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Satisfiable?</th>
<th>No Abstr. time (s)</th>
<th>Mixed time (s)</th>
<th>#iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt.c, claim 1</td>
<td>109</td>
<td>no</td>
<td>25</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>sqrt.c, claim 2</td>
<td>109</td>
<td>no</td>
<td>25</td>
<td>0.6</td>
<td>7</td>
</tr>
<tr>
<td>sqrt.c, claim 3</td>
<td>109</td>
<td>no</td>
<td>25</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>sqrt.c, claim 4</td>
<td>109</td>
<td>no</td>
<td>OM</td>
<td>478</td>
<td>103</td>
</tr>
<tr>
<td>sqrt.c, claim 5</td>
<td>109</td>
<td>no</td>
<td>25</td>
<td>1.2</td>
<td>15</td>
</tr>
<tr>
<td>sqrt.c, claim 6</td>
<td>109</td>
<td>no</td>
<td>25</td>
<td>0.6</td>
<td>7</td>
</tr>
<tr>
<td>sqrt.c, claim 7</td>
<td>109</td>
<td>no</td>
<td>6716</td>
<td>84</td>
<td>86</td>
</tr>
<tr>
<td>sqrt.c, claim 1</td>
<td>51</td>
<td>no</td>
<td>24</td>
<td>13589</td>
<td>44</td>
</tr>
<tr>
<td>sqrt.c, claim 2</td>
<td>51</td>
<td>yes</td>
<td>9</td>
<td>TO</td>
<td>107</td>
</tr>
<tr>
<td>minver.c, claim 1</td>
<td>156</td>
<td>no</td>
<td>1</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>minver.c, claim 2</td>
<td>156</td>
<td>yes</td>
<td>2</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>sin.c, claim 1</td>
<td>46</td>
<td>no</td>
<td>13864</td>
<td>781</td>
<td>47</td>
</tr>
<tr>
<td>sin.c, claim 2</td>
<td>46</td>
<td>no</td>
<td>13831</td>
<td>281</td>
<td>47</td>
</tr>
<tr>
<td>sin.c, claim 3</td>
<td>46</td>
<td>no</td>
<td>TO</td>
<td>1774</td>
<td>63</td>
</tr>
<tr>
<td>gaussian.c, claim 1</td>
<td>108</td>
<td>no</td>
<td>TO</td>
<td>14437</td>
<td>137</td>
</tr>
</tbody>
</table>
Related work

Constraint satisfaction

C. Michel, M. Rueher and Y. Lebbah: Solving constraints over floating-point numbers. CP2001

B. Botella, A. Gotlieb and C. Michel: Symbolic execution of floating-point computations. STVR2006

SMT

P. Ruemmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. SMT 2010

A. Brillout, D. Kroening and T. Wahl. Mixed abstractions for floating point arithmetic. FMCAD 2009

R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. TACAS 2009

Incomplete Solvers

Abstract Interpretation
Scalable.
Precision requires experts

Decision Procedures
Precise.
Scalability requires experts

Requires experts, scalable, precise

Manual
Conclusion Part I

Automatic

Scalable

Theorem proving

Precise

Abstract interpretation

Decision procedures

Abstract Interpreter

Safe

Decision Procedures

Bug
Questions so far?
Part II
We are interested in techniques that are
• scalable
• sufficiently precise to prove safety
• fully automatic

Central insight: Modern decision procedures are abstract interpreters!
Manually adjusting analysis precision by abstract partitioning

void foo(int x)
{
    int y;
    if(x < 0)
        y = 1;
    else
        y = -1;  \[ y \in [-1, 1] \]
    assert(y != 0);
}

Potentially unsafe!

\[ y \in [-1, 1] \]

\[ y \in [-1, 1] \]

void foo_precise(int x)
{
    if(x < 0)
        foo(x);
    else
        foo(x);
}

Safe!
How do we find the partition automatically?
SAT solving by example

SAT solvers accept formulas in conjunctive normal form

\[ \varphi = (p \lor \neg q) \land \ldots \land (\neg r \lor w \lor q) \]

Their main data structure is a partial variable assignment which represents a solution candidate

\[ V \rightarrow \{ t, f \} \]
SAT solving: Deduction

\[ \varphi = p \land (\neg p \lor \neg q) \land (q \lor r \lor \neg w) \land (q \lor r \lor w) \]

SAT deduces new facts from clauses:

\[ p \leftrightarrow t \quad q \leftrightarrow f \]

At this point, clauses yield no further information
SAT is Abstract Analysis: Deduction

\[ \varphi = p \land (\neg p \lor \neg q) \land (q \lor r \lor \neg w) \land (q \lor r \lor w) \]

\[ \rightarrow \quad p \mapsto t \quad \rightarrow \quad p \mapsto t \]

\[ q \mapsto f \]

The result of deduction is identical to applying interval analysis to the program:

```c
void foo(void)
{
    bool p, q, r, w;
    
    if(p)
    {
        if(!p || !q)
        {
            if(q || r || !w)
            {
                if(q || r || w)
                    assert(0);
            }
        }
    }
}
```

Deduction in a SAT solver is abstract analysis
SAT solving: Decisions

$$\varphi = p \land (\neg p \lor \neg q) \land (q \lor r \lor \neg w) \land (q \lor r \lor w)$$

SAT solver makes a “guess”

Pick an unassigned variable and assign a truth value

$$p \mapsto t$$  \hspace{1cm}  $$p \mapsto t$$

$$q \mapsto f$$  \hspace{1cm}  $$q \mapsto f$$

$$r \mapsto f$$

Now new deductions are possible
SAT solving: Learning

\[ \varphi = p \land (\neg p \lor \neg q) \land (q \lor r \lor \neg w) \land (q \lor r \lor w) \]

\[ p \mapsto t \]
\[ q \mapsto f \]
\[ r \mapsto f \]

The variable \( w \) would have to be both true and false.

The contradiction is the result of \( r \) being assigned to false as part of a decision. The SAT solver therefore learns that \( r \) must be true:

\[ \varphi \leftarrow \varphi \land r \]
SAT solving: Learning

\[ \varphi = p \land (\neg p \lor \neg q) \land (q \lor r \lor \neg w) \land (q \lor r \lor w) \]

\[ p \mapsto t \quad q \mapsto f \quad r \mapsto f \quad w \mapsto f \]

The variable \( w \) would have to be both true and false.

The contradiction is the result of \( r \) being assigned to false as part of a decision. The SAT solver therefore learns that \( r \) must be true:

\[ \varphi \leftarrow \varphi \land r \]
SAT is Abstract Analysis: Decisions & Learning

\[ \varphi \rightarrow \varphi \land r \]

```
void foo(void)
{
    bool p, q, r, w;
    if(p)
        if(!p || !q)
            if(q || r || !w)
                if(q || r || w)
                    assert(0);
}
```

```
void foo_precise()
{
    if(r)
        foo();
}
```

Decisions and learning in a SAT solver are abstract partitioning
SAT is Abstract Analysis

- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.

Domain A

Expanding the scope of SAT
SAT is Abstract Analysis

- Deduction in SAT is abstract interpretation
- Decisions and learning are abstract partitioning
- The SAT algorithm is really an automatic partition refinement algorithm.

Expanding the scope of SAT

Domain A

SAT(A)

Data

Rich logic, e.g. FP

Prop. Logic

Programs

Boolean programs

Control

Monday, 23 July 12
SAT for programs

**Abstract Implication Graph**

**DL0**
- $c_2 : a \leq -1$
- $c_3 : a \leq 0$
- $c_3 : a \geq 0$
- $c_2 : a \geq -1$
- $c_1 : a \leq -2$

**DL1**
- $n_1 : a \leq -2$
- $c_1 : \top$
- $c_2 : \bot$
- $c_3 : \bot$
- $c_4 : \bot$

**DPLL is Abstract Interpretation**

**SAFE**

- $a \leq -2$
- $[a = -1]$
- $[a = 0]$
- $[a \geq 1]$
- $b := -1$
- $b := 2$
- $b := -2$
- $b \leq 0$
- $b = 0$

**Generalise!**

- find cut

$\neg(n_2 : b \geq 1)$
Prototype:
Abstract Conflict Driven Learning (ACDL)

- Implementation over floating-point intervals
- Automatically refines an analysis in a way that is
  - Property dependent
  - Program dependent
- Uses learning to intelligently explore partitions
- **Significantly more precise** than mature abstract interpreters
- **Significantly more efficient** than floating-point decision procedures on short non-linear programs
Demo
Fig. 2. Execution times of Astrée, CBMC, and CDFL; wrong results set to 3600s.

Fig. 3. Effects of learning and decision heuristics.

Several observations: on average, our analysis is 264 times faster than CBMC, if CBMC terminates properly at all. The largest speed-up is a factor of 1595. Although Astrée is often faster than our prototype, its precision is insufficient in many cases – we obtained 16 false alerts for the 33 safe benchmarks.

Decision Heuristics and Learning

Figure 3 visualises the effects of learning and decision heuristics. Learning has a significant influence on runtime, as does the choice of a decision heuristic. We compare a random heuristic, which picks a restriction over a random variable, with a range-based one, which always aims to restrict the least restricted variable. Random decision making outperforms range-based. Activity-based heuristics common in sat may work as well in our case.

Dynamic Precision Adjustment

One of the main advantages of our procedure is that refinement is property-dependent. The precision of the analysis dynamically adapts to match the precision required by the property. This is illustrated in Figure 4 where we check bounds on the result of computing a sine approximation under the input range $\pi^2$, $\pi^2$. The input value is shown on the x-axis, the result of the computation on the y-axis. The bound we check against is depicted as two red horizontal lines, boundaries of explored partitions are shown as black vertical lines. The actual maximum of the function lies at about $1\nabla 0.0921$. As the checked bound (Figure 4 shows bounds $1\nabla 2$ and $1\nabla 0.1$) approaches

Average speedup over CBMC $\sim 270x$
Implementation
Number of partitions vs. tightness of bound

result ≤ 2.0

result ≥ -2.0
Number of partitions vs. tightness of bound

result $\leq 1.5$

$\pi \over 2$

$\pi \over 2$

result $\geq -1.5$
Number of partitions vs. tightness of bound

result ≤ 1.2

result ≥ -1.2
Number of partitions vs. tightness of bound

result $\leq 1.1$

result $\geq -1.1$
Number of partitions vs. tightness of bound

\[ \text{result} \leq 1.01 \]

\[ \text{result} \geq -1.01 \]
Number of partitions vs. tightness of bound

\begin{center}
\begin{tikzpicture}
    \draw[->] (-3,0) -- (3,0);
    \draw[->] (0,-3) -- (0,3);
    \draw (-2.99,0) -- (-2.99,0.01);
    \draw (-2.99,0.01) -- (-2.99,3);
    \draw (-2.99,3) -- (-2.99,3.01);
    \draw (-3,0.01) -- (-2.99,0.01);
    \draw (-3,0.01) -- (-3,-2.99);
    \draw (-3,-2.99) -- (-2.99,-2.99);
    \node at (-3,0.5) {$\pi / 2$};
    \node at (-3,-0.5) {$-\pi / 2$};
    \node at (0.5,0) {$\pi / 2$};
    \node at (0.5,-0.5) {-1.001};
    \node at (0.5,-1.5) {1.001};
    \node at (1.5,-1.5) {-1.001};
    \node at (1.5,-2.5) {1.001};
    \node at (2.5,-2.5) {-1.001};
    \node at (2.5,-3.5) {1.001};
    \node at (3.5,-3.5) {-1.001};
    \node at (3.5,-4.5) {1.001};
    \node at (-3.5,3.5) {-1.001};
    \node at (-3.5,2.5) {1.001};
    \node at (-3.5,1.5) {-1.001};
    \node at (-3.5,0.5) {1.001};
    \node at (-3.5,-0.5) {-1.001};
    \node at (-3.5,-1.5) {1.001};
    \node at (-3.5,-2.5) {-1.001};
    \node at (-3.5,-3.5) {1.001};
    \node at (-3.5,-4.5) {-1.001};
    \node at (3.5,3.5) {-1.001};
    \node at (3.5,2.5) {1.001};
    \node at (3.5,1.5) {-1.001};
    \node at (3.5,0.5) {1.001};
    \node at (3.5,-0.5) {-1.001};
    \node at (3.5,-1.5) {1.001};
    \node at (3.5,-2.5) {-1.001};
    \node at (3.5,-3.5) {1.001};
    \node at (3.5,-4.5) {-1.001};
    \draw[blue,thick] (-3,-2.5) .. controls (-2,-1) .. (-1,0) .. controls (0,1) .. (1,2) .. controls (2,3) .. (3,2.5);
    \draw[red,thick] (-3,-2.5) -- (-3,-2.5);
    \draw[red,thick] (-2,-2.5) -- (-2,-2.5);
    \draw[red,thick] (-1,-2.5) -- (-1,-2.5);
    \draw[red,thick] (0,-2.5) -- (0,-2.5);
    \draw[red,thick] (1,-2.5) -- (1,-2.5);
    \draw[red,thick] (2,-2.5) -- (2,-2.5);
    \draw[red,thick] (3,-2.5) -- (3,-2.5);
    \draw[red,thick] (-3,2.5) -- (-3,2.5);
    \draw[red,thick] (-2,2.5) -- (-2,2.5);
    \draw[red,thick] (-1,2.5) -- (-1,2.5);
    \draw[red,thick] (0,2.5) -- (0,2.5);
    \draw[red,thick] (1,2.5) -- (1,2.5);
    \draw[red,thick] (2,2.5) -- (2,2.5);
    \draw[red,thick] (3,2.5) -- (3,2.5);
    \node at (-3,-2.5) {result \leq 1.001};
    \node at (-3,2.5) {result \geq -1.001};
\end{tikzpicture}
\end{center}
Current and Future Work

- Develop an SMT solver for floating point logic
- Model on the success of propositional SAT:
  - Simple abstract domain
  - Highly efficient data structures

![Graph showing time vs. performance from 2000 to 2007]
Current and Future Work

- Develop an SMT solver for floating point logic
- Model on the success of propositional SAT:
  - Simple abstract domain
  - Highly efficient data structures

![Rich logic, e.g. FP](#)

![Prop. Logic](#)

<table>
<thead>
<tr>
<th>Programs</th>
<th>Boolean programs</th>
</tr>
</thead>
</table>

![Graph](#)
Current and Future Work

- Reengineer prototype into a tool for floating point verification
- Significantly improved efficiency
- Generic interface for integrating abstract domains
- Development and generalisation of heuristics and learning strategies
Current and Future Work

- Reengineer prototype into a tool for floating point verification
- Significantly improved efficiency
- Generic interface for integrating abstract domains
- Development and generalisation of heuristics and learning strategies

```
Rich logic, e.g. FP
Prop. Logic
```

```
Programs
Boolean programs
```

Monday, 23 July 12
Refining loops with ACDL

• Currently, loops cause imprecision in our analysis
• Analysis may fail to prove safety

```c
void foo()
{
    int i = -1;
    while(*)
        i *= -1;
    assert(i != 0);
}
```

Successful analysis requires distinguishing between even and odd numbers of loop iterations
Refining loops with ACDL

- Solution: Apply the SAT algorithm to control flow itself
  - Make decisions over control-flow (e.g., assume odd number of loop iterations)
  - Learning permanently alters control flow
- Resulting analysis can dynamically vary precision from full abstraction to precise case exploration
Conclusion - Part II

Automatic

Scalable
Abstract interpretation

Decision procedures

Theorem proving

Precise

Fully automatic

Scalability
ACDL

Precision
Thank you for your attention
Additional slides
Two approaches to lift SAT to a richer logic $\mathcal{L}$

**Eager approach**

- $\Phi \in \mathcal{L}$
- $\varphi \in \text{Prop}$
- **Fully translate to propositional logic**
- **Propositional SAT solver**
- SAT $\rightarrow$ UNSAT

**Lazy approach**

- $\Phi \in \mathcal{L}$
- $\varphi \in \text{Prop}$
- **Partially translate to propositional logic**
- **Propositional SAT solver**
- UNSAT
- SAT $\rightarrow$ infeasible
- $\mathcal{L}$ Theory solver
- SAT

---

Monday, 23 July 12
Lazy SMT works if the logic can be decomposed into an efficiently solvable theory component and a propositional component.

The approach breaks down if significant communication is necessary between the two.

Due to the non-numeric behaviour in floating-point arithmetic such as rounding, special values, etc., there is no clear decomposition. Therefore, analysis is often performed over the real numbers instead, which may lead to unsound results.
Collected references
Axiomatisations of FP


J. Harrison. A machine-checked theory of floating-point arithmetic. TPHOLs 1999

Specification of FP properties


S. Boldo and J.C. Filliâtre. Formal verification of floating-point programs. ARITH 2007
References

Applications

J. Harrison. Floating point verification in HOL light: The exponential function. FMSD, 16(3), 2000

J. Harrison. Floating-point verification. FM 2005

J. Harrison. Formal verification of square root algorithms. FMSD, 22(2), 2003


R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. TPHOLs 2000


Floating point abstract domains

A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. SAS 2010

L. Chen, A. Miné and P. Cousot. A sound floating-point polyhedra abstract domain. APLAS 2008

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. ESOP 2004

L. Chen, A. Miné, J. Wang and P. Cousot. An abstract domain to discover interval Linear Equalities. VMCAI 2010


K. Ghorbal, E. Goubault and S. Putot. The zonotope abstract domain Taylor I. CAV 2009

B. Jeannet, and A. Miné. Apron: A library of numerical abstract domains for static analysis. CAV 2009

D. Monniaux. Compositional analysis of floating-point linear numerical filters. CAV 2005

J. Feret. Static analysis of digital filters. ESOP 2004


E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. NSAD 2005

E. Goubault. Static analyses of the precision of floating-point operations. SAS 2001
References

Industrial Case Studies

E. Goubault, S. Putot, P. Baufreton, J. Gassino. Static analysis of the accuracy in control systems: principles and experiments. FMICS 2007


J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. SAFECOMP 2007

J. Souyris. Industrial experience of abstract interpretation-based static analyzers. IFIP 2004

P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. EMSOFT 2007

FP Static Analysers


P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and Xavier Rival. The ASTRÉÉ analyzer. ESOP 2005

E. Goubault, M. Martel and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. ESOP 2002
Related work

Constraint satisfaction

C. Michel, M. Rueher and Y. Lebbah: Solving constraints over floating-point numbers. CP2001

B. Botella, A. Gotlieb and C. Michel: Symbolic execution of floating-point computations. STVR2006

SMT

P. Ruemmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. SMT 2010

A. Brillout, D. Kroening and T. Wahl. Mixed abstractions for floating point arithmetic. FMCAD 2009

R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. TACAS 2009

Incomplete Solvers