Logical Interpretation

Static Program Analysis Using Theorem Proving

Ashish Tiwari

Tiwari@csl.sri.com

Computer Science Laboratory
SRI International
Menlo Park CA 94025
`http://www.csl.sri.com/~tiwari`

Ideas partly contributed by all my collaborators

The Problem

Complex Systems: How to

- understand ?
- design ?

Examples:

- living cell, drug action
- software systems
- embedded systems
- cyber physical systems

The Only Way We Know

Using formal mathematical models

Explored and analyzed using
Automated Deduction ?

Flashback: Use of deduction technology as Embedded Logical Engines
Resulted in SMT approaches

What We Now Need: Part I

Evidence: Embed the technology in tools

- Embedded System Design Tools: Matlab Simulink/Stateflow
- Software Development Tools
- Drug Design Tools
- Medical Devices
- ⋮

What We Now Need: Part II

Next Generation Automated Deduction Engine: Requirements–

Attributes	Why	Modern SMT Solvers
speed	embedded use	yes
support for theories	symbols have meaning	yes
interface	embedded use	lacking
beyond satisfiability	need more	no
reduced expressiveness		partly
stochastic reasoning		no

Evidence

Some case studies:

Application	Formalism	Core Technology	Example
Embedded Sys.	Hybrid Systems	Th. of Reals	Transmission, Powertrain
Systems Bio.	Discrete Sys.	SAT/MaxSAT	Cell Signalling
Medical Devices	Continuous Sys.	Linear Arith.	Insulin Control
Software Verif.	C programs	----	Benchmarks, Code Fragments

Outline of the Talk

Part I.	Over-approximating $\vee$
Part II.	Over-approximating $\vee$ in a combination of theories
Part III.	Approximating $\vee, \wedge, \exists, \forall$
Part IV.	Theory Anyone?

Example

```
x := 0; y := 0; z := n;
while (*) {
   if (*) {
      x := x+1;
      z := z-1;
   } else {
       y := y+1;
       z := z-1;
   }
}
```

Traditional Approach: Annotate & Check

```
x := 0; y := 0; z := n;
[ z - x - y == n ]
while (*) {
   if (*) {
      x := x+1;
      z := z-1;
      [ z - x - y == n ]
   } else {
       y := y+1;
       z := z-1;
       [ z - x - y == n ]
   }
}
```

Traditional Approach: Annotate & Check

Proof obligation generated:

$$z - x - y = n \ \wedge \ x' = x + 1 \ \wedge \ z' = z - 1 \wedge y' = y$$

$$\overset{\mathbb{T}}{\Rightarrow} \quad z' - x' - y' = n$$

$$z - x - y = n \ \wedge \ y' = y + 1 \ \wedge \ z' = z - 1 \wedge x' = x$$

$$\overset{\mathbb{T}}{\Rightarrow} \quad z' - x' - y' = n$$

The theory $\mathbb{T}$ determined by semantics of the programming language.

Example: Abstract Interpretation

```
[ true ]
x := 0; y := 0; z := n;
[ $x = 0 \wedge y = 0 \wedge z = n$ ]
while (*) {
    if (*) {
        x := x+1;
        z := z-1; [ $(x = 1 \wedge y = 0 \wedge z = n - 1)$ ]
    } else {
         y := y+1;
         z := z-1; [ $(x = 0 \wedge y = 1 \wedge z = n - 1)$ ]
    }
    [ $(x = 1 \wedge y = 0 \wedge z = n - 1) \vee (x = 0 \wedge y = 1 \wedge z = n - 1)$ ]
}
```

Example: Abstract Interpretation

$$(x = 1 \wedge y = 0 \wedge z = n - 1) \vee (x = 0 \wedge y = 1 \wedge z = n - 1)$$

Suppose we do not have $\vee$ in our language

We can only represent conjunctions of atomic facts

We need to overapproximate

We need to find a conjunction of atomic formulas that is implied by both $x = 1 \wedge y = 0 \wedge z = n - 1$ and $x = 0 \wedge y = 1 \wedge z = n - 1$

What is such a fact? $x + y = 1 \wedge z = n - 1$

Example: Abstract Interpretation

```
[ true ]
x := 0; y := 0; z := n;
[ $x = 0 \wedge y = 0 \wedge z = n$ ]
while (*) {
    [ $(x = 0 \wedge y = 0 \wedge z = n) \vee (x + y = 1 \wedge z = n - 1)$ ]
    if (*) {
        x := x+1;
        z := z-1; [ $(x = 1 \wedge y = 0 \wedge z = n - 1)$ ]
    } else {
        y := y+1;
        z := z-1; [ $(x = 0 \wedge y = 1 \wedge z = n - 1)$ ]
    }
    [ $(x + y = 1 \wedge z = n - 1)$ ]
}
```

Hence, we need to over-approximate

$$((x + y = 1 \wedge z = n - 1) \vee x = 0 \wedge y = 0 \wedge z = n)$$

$$\begin{aligned} (x + y = 1 \wedge z = n - 1) &\overset{\mathbb{T}}{\Rightarrow} z + x + y = n \\ (x = 0 \wedge y = 0 \wedge z = n) &\overset{\mathbb{T}}{\Rightarrow} z + x + y = n \end{aligned}$$

This is exactly the invariant we had annotated by hand.

Logical Interpretation

Abstract Interpretation over logical lattices

Lattices defined by

elements	:	some subset of formulas in $\mathbb{T}$ closed under $\wedge$
partial order	:	some subset of $\overset{\mathbb{T}}{\Rightarrow}$

A common class is strictly logical lattices:

elements	:	conjunction ϕ of atomic formulas in Th
partial order	:	$\phi \sqsubseteq \phi'$ if $Th \models \phi \Rightarrow \phi'$

In any logical lattice

meet $\sqcap$ $\mapsto$ (over-approximation of) logical and $\wedge$ $(\lceil\wedge\rceil)$

join $\sqcup$ $\mapsto$ over-approximation of logical or $\lceil\vee\rceil$

partial order $\sqsubseteq$ $\mapsto$ under-approximation of logical implies $\lfloor\Rightarrow\rfloor$

projection $\mapsto$ over-approximation of logical exists $\lceil\exists\rceil$

In strictly logical lattices:

meet $\sqcap$ $\mapsto$ $\wedge$

join $\sqcup$ $\mapsto$ $\phi_1\lceil\vee\rceil\phi_2$ is the strongest $\phi \in \Phi$ s.t. $\phi_i \overset{\mathbb{T}}{\Rightarrow} \phi$ for $i = 1, 2$

partial order $\sqsubseteq$ $\mapsto$ $\overset{\mathbb{T}}{\Rightarrow}$

projection $\mapsto$ $\lceil\exists\rceil U.\phi$ is the strongest $\phi' \in \Phi$ s.t. $(\exists U.\phi) \overset{\mathbb{T}}{\Rightarrow} \phi'$

Challenge: For what domains can we efficiently compute these operations?

Over-Approximation of $\vee$: Examples

- Linear arithmetic with equality (Karr 1976)
  Eg. $\{x = 0, y = 1\}\lceil\vee\rceil\{x = 1, y = 0\} = \{(x + y = 1)\}$

- Linear arithmetic with inequalities (Cousot and Halbwachs 1978)
  Eg. $\{x = 0\}\lceil\vee\rceil\{x = 1\} = \{0 \leq x, x \leq 1\}$

- Nonlinear equations (polynomials) (Rodriguez-Carbonell and Kapur 2004)
  Eg. $\{x = 0\}\lceil\vee\rceil\{x = 1\} = \{x(x - 1) = 0\}$

- Term Algebra (Gulwani, T. and Necula 2004)
  Eg. $\{x = a, y = f(a)\}\lceil\vee\rceil\{x = b, y = f(b)\} = \{y = f(x)\}$

UFS does not define a logical lattice

The join of two finite sets of facts need not be finitely presented. [Gulwani, T. and Necula 2004]

$$\begin{aligned} \phi_1 &\equiv \{a = b\} \\ \phi_2 &\equiv \{fa = a,\ fb = b,\ ga = gb\} \\ \phi_1 \lceil \vee \rceil \phi_2 &\equiv \bigwedge_i gf^i a = gf^i b \end{aligned}$$

The formula $\bigwedge_i gf^i a = gf^i b$ can not be represented by finite set of ground equations.

Proof. It induces infinitely many congruence classes with more than one signature.

Part II. Over-Approximation in Union of Theories

Combining Logical Interpreters: Motivation

```
x :=0; y := 0;
u := 0; v := 0;
while (*) {
    x := u + 1;
    y := 1 + v;
    u := F(x);
    v := F(y);
}
assert( x = y )
```

$\Sigma = \Sigma_{LA} \cup \Sigma_{UFS}$

$Th = Th_{LA} + Th_{UFS}$

```
x := c; y := c;
u := c; v := c;
while (*) {
    x := G(u, 1);
    y := G(1, v);
    u := F(x);
    v := F(y);
}
assert( x = y )
```

$\Sigma = \Sigma_{UFS}$

$Th = Th_{UFS}$

```
x :=0; y := 0;
u := 0; v := 0;
while (*) {
    x := u + 1;
    y := 1 + v;
    u := *;
    v := *;
}
assert( x = y )
```

$\Sigma = \Sigma_{LA}$

$Th = Th_{LA}$

Combining Logical Interpreters

Combining abstract interpreters is not easy [Cousot76]

For combining logical interpreters (over strictly logical lattices), we need to combine:

- $\lceil \vee \rceil$
- $\lceil \exists \rceil$
- $\overset{\mathbb{T}}{\Rightarrow}$

Bad Example:

$$\begin{aligned} & (x = 0 \wedge y = 1) \ \sqcup \ (x = 1 \wedge y = 0) \\ = \quad & x + y = 1 \wedge C[x] + C[y] = C[0] + C[1] \end{aligned}$$

Logical Product

Given two logical lattices, we define the logical product as:

elements	:	conjunction ϕ of atomic formulas in $Th_1 \cup Th_2$
$E \sqsubseteq E'$	:	$E \Rightarrow_{Th_1 \cup Th_2} E'$ and $\mathit{AlienTerms}(E') \subseteq \mathit{Terms}(E)$

$\mathit{AlienTerms}(E)$	=	subterms in E that belong to different theory
$\mathit{Terms}(E)$	=	all subterms in E, plus all terms equivalent to these subterms (in $Th_1 \cup Th_2 \cup E$)

Eg. $\{x = F(a+1), y = a\} \sqcup \{x = F(b+1), y = b\} = \{x = F(y+1)\} \because$

$$\begin{array}{rcl}
x = F(a+1) \wedge y = a & \Rightarrow & x = F(y+1) \\
x = F(b+1) \wedge y = b & \Rightarrow & x = F(y+1) \\
x = F(\underline{a+1}) \wedge y = a & \Rightarrow & y+1 = \underline{a+1} \\
x = F(\underline{b+1}) \wedge y = b & \Rightarrow & y+1 = \underline{b+1}
\end{array}$$

Combining the Preorder Test

Combining satisfiability procedures

Nelson-Oppen combination method

Combining Join Operator

Given procedures:

$$\begin{array}{lcl} \lceil\vee\rceil_{L_1}(E_l, E_r) & : & \text{Computes } E_l \lceil\vee\rceil E_r \text{ in lattice } L_1 \\ \lceil\vee\rceil_{L_2}(E_l, E_r) & : & \text{Computes } E_l \lceil\vee\rceil E_r \text{ in lattice } L_2 \end{array}$$

We wish to compute $E_l \lceil\vee\rceil E_r$ in the logical product $L_1 * L_2$

Example.

$$\{z = a + 1, y = f(a)\} \lceil\vee\rceil \{z = b - 1, y = f(b)\} \quad = \quad \{y = f(1 + z)\}$$

Combining Join Operators

$\boxed{z = a - 1, y = f(a)}$ $\quad$ $\boxed{z = b - 1, y = f(b)}$

Purify+NOSat $\quad$ $z = a - 1 \quad y = f(a) \quad z = b - 1 \quad y = f(b)$

LR-Exchange $\quad$ $a = \langle a, b\rangle \quad a = \langle a, b\rangle \quad b = \langle a, b\rangle \quad b = \langle a, b\rangle$

Base Joins $\quad$ $\boxed{Join_{LA}}$ $\quad$ $\boxed{Join_{UF}}$

$\langle a, b\rangle = 1 + z \qquad y = f(\langle a, b\rangle)$

Quant Elim $\quad$ $\boxed{QE_{UF*LA}}$

Return $\quad$ $\boxed{y = f(1 + z)}$

Existential Quantification Operator

Required to compute transfer function for assignments

$E = \lceil\exists\rceil_L(E', V)$ if E is the least element in lattice L s.t.

- $E' \sqsubseteq_L E$
- $Vars(E) \cap V = \emptyset$

Examples:

- $\lceil\exists\rceil_{LA} a : (x < a \wedge a < y) = (x \leq y)$
- $\lceil\exists\rceil_{UF} a : (x = f(a) \wedge y = f(f(a))) = (y = f(x))$
- $\lceil\exists\rceil_{LA*UF} a, b, c : (a < b < y \wedge z = c + 1 \wedge a = ffb \wedge c = fb) = (f(z-1) \leq y)$

How to construct $\lceil\exists\rceil_{LA*UF}$ using $\lceil\exists\rceil_{LA}$ and $\lceil\exists\rceil_{UF}$?

Combining QE Operators

Problem	$\boxed{a < b < y, z = c + 1, a = ffb, c = fb}$			$\boxed{\{a, b, c\}}$
Purify+NOSat	$\boxed{a < b < y, z = c + 1}$		$\boxed{a = ffb, c = fb}$	
QSat		$\longrightarrow$	$c \mapsto z - 1$	
QSat	$a \mapsto fc$	$\longleftarrow$		
Base QEs	$\boxed{QE_{LA}}$		$\boxed{QE_{UF}}$	
	$\boxed{a \leq y, z = c + 1}$		$\boxed{a = fc}$	
Substitute	$c \mapsto z - 1, a \mapsto fc$			
Return	$\boxed{f(z - 1) \leq y}$			

Part III. Approximating $\vee, \wedge, \exists, \forall$

Quantified Abstract Domain

```
array-init$(A, n)$
  for ($i = 0$; $i < n$; i++) {
        $A[i] = 0$
  }
  [ $\forall k(0 \le k < n \Rightarrow \mathsf{A}[k] = 0)$ ]
```

Array Initialization

```
array-init$(A, n)$
  for ($i = 0$; $i < n$; i++) {
      $(i = 1 \wedge A[0] = 0) \vee (i = 2 \wedge A[0] = 0 \wedge A[1] = 0)$
      $A[i] = 0$
  }
```

Let us write it out as a quantified fact.

Array Initialization

```
array-init($(A, n)$)
  for ($i = 0$; $i < n$; i++) {
        $(i = 1 \wedge \forall k(k = 0 \Rightarrow A[k] = 0)) \vee$
        $(i = 2 \wedge \forall k(k = 0 \Rightarrow A[k] = 0) \wedge \forall k(k = 1 \Rightarrow A[k] = 0))$
        $A[i] = 0$
  }
```

Too many quantified facts...let us merge them into one.

$$i = 2 \wedge \forall k(\_\_\_\_ \Rightarrow A[k] = 0)$$

$\_\_\_\_$ should be $k = 0 \lfloor \vee \rfloor k = 1$:

$$0 \leq k \leq 1 \Rightarrow (k = 0 \vee k = 1)$$

Array Initialization

```
array-init$(A, n)$
  for ($i = 0$; $i < n$; i++) {
        $i = 1 \wedge \forall k (k = 0 \;\Rightarrow\; A[k] = 0) \vee$
        $i = 2 \wedge \forall k (0 \leq k < 2 \;\Rightarrow\; A[k] = 0)$
        $A[i] = 0$
  }
```

Now we need to join two quantified facts.

Array Initialization

$i = 1$ $\qquad\qquad$ $\lceil \vee \rceil$ $\qquad\qquad$ $i = 2$

$\forall k (k = 0 \Rightarrow A[k] = 0)$ $\qquad\qquad$ $\forall k (0 \leq k < 2 \Rightarrow A[k] = 0)$

$$1 \leq i \leq 2$$

$$\forall k (\_\_\_\_ \Rightarrow A[k] = 0)$$

Obviously, $\_\_\_\_$ should be $k = 0 \lfloor \wedge \rfloor 0 \leq k < 2$.

$k = 0$ is no good.

Array Initialization

$i = 1$ $\qquad\qquad$ $\lceil \vee \rceil$ $\qquad\qquad$ $i = 2$

$\forall k(k = 0 \Rightarrow A[k] = 0)$ $\qquad\qquad$ $\forall k(0 \leq k < 2 \Rightarrow A[k] = 0)$

$$1 \leq i \leq 2$$

$$\forall k(\underline{\quad\quad} \Rightarrow A[k] = 0)$$

Hmmm, $\underline{\quad\quad}$ should be

$$i = 1 \Rightarrow k = 0 \lfloor \wedge \rfloor i = 2 \Rightarrow 0 \leq k < 2$$

Let us see if the answer satisfies this.

$$0 \leq k < i \Rightarrow (i = 1 \Rightarrow k = 0 \wedge i = 2 \Rightarrow 0 \leq k < 2)$$

The Quantified Domain

$$E \wedge \bigwedge_i \forall U_i (F_i \Rightarrow e_i)$$

The Interface

Function	Description
$E_1 \lceil\vee\rceil E_2$	join of E_1 and E_2
$E_1 \lceil\wedge\rceil E_2$	meet of E_1 and E_2
$\lceil\exists\rceil x.E$	eliminate x from E
$E_1 \ \lfloor\Rightarrow\rfloor \ E_2$	partial order test comparing E_1 and E_2
$(E_1 \lfloor\vee\rfloor E_2)/E$	under-approximate $E \Rightarrow (E_1 \vee E_2)$
$(E_1 \Rightarrow E_1')\lfloor\wedge\rfloor(E_2 \Rightarrow E_2')$	underapprox. $(E_1 \Rightarrow E_1') \wedge (E_2 \Rightarrow E_2')$
$\lfloor\forall\rfloor x.(E \Rightarrow E')$	underapproximate $\forall x(E \Rightarrow E')$

How are Under-Approximations Computed?

Under-approximation operators == Abduction

Given environment E and observation F, generate an explanation F' such that

$$\begin{array}{rcll} E \wedge F' & \Rightarrow & F & \text{abduction} \\ F' & \Rightarrow & (E \Rightarrow F) & \text{underapproximation} \end{array}$$

We start with over-approximations and then refine them using abduction.

Magic

$i = 1$ $\lceil\vee\rceil$ $i = 2$

$\forall k(k = 0 \Rightarrow A[k] = 0)$ $\forall k(0 \leq k < 2 \Rightarrow A[k] = 0)$

$$1 \leq i \leq 2$$

$$\forall k(\_\_\_\_ \Rightarrow A[k] = 0)$$

Hmmm, $\_\_\_\_$ should be

$$i = 1 \Rightarrow k = 0 \lfloor\wedge\rfloor i = 2 \Rightarrow 0 \leq k < 2$$

Compute

$$i = 1 \wedge k = 0 \lceil\vee\rceil i = 2 \wedge 0 \leq k < 2$$

Join on linear arithmetic returns

$$1 \leq i \leq 2 \wedge 0 \leq k < i$$

Part IV. Theory Anyone?

Part I. Invariant Checking

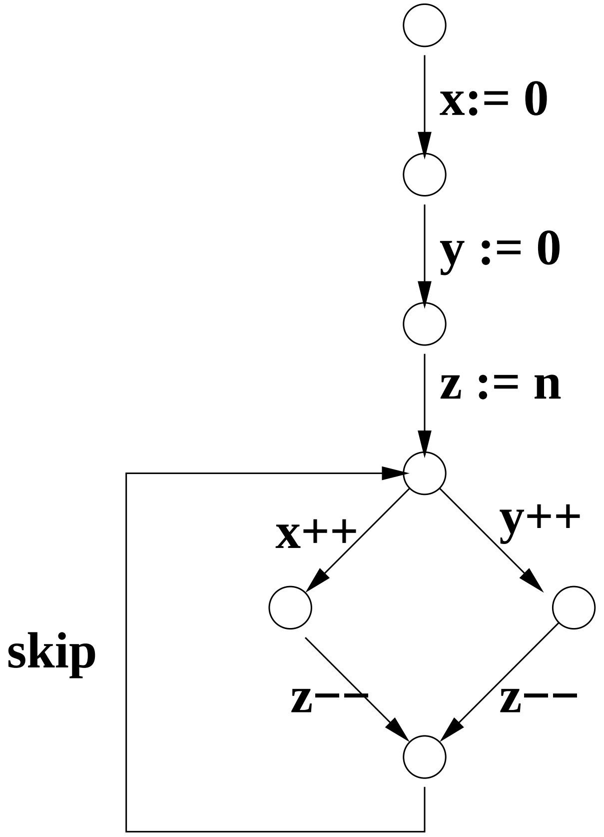
Program: A directed graph whose edges are labelled with:

- $x := e$
- $x :=?$
- `skip`

Example

Given the following program and assertion $z - x - y = n$ at the end, check if assertion is an invariant of the program.

```
x := 0; y := 0; z := n;
while (*) {
   if (*) {
      x := x+1;
      z := z-1;
   } else {
       y := y+1;
       z := z-1;
   }
}
assert(z - x - y = n)
```



Invariant Checking via Backward Propagation

```
[ n - 0 - 0 = n ]
x := 0; y := 0; z := n;
[ z - x - y = n ]
while (*) {
   [ z - x - y = n ]
   if (*) {
      x := x+1;
      z := z-1;
      [ z - x - y = n ]
   } else {
       y := y+1;
       z := z-1;
       [ z - x - y = n ]
   }
   [ z - x - y = n ]
}
```

Simple Programs using Linear Arithmetic

Program P : Simple program using expression language of linear arith.

Assertion : linear arithmetic equality

In this case,

- At each point, we have a conjunction of linear equations
- Such a conjunct can have at most n non-redundant equations
- Therefore fixpoint converges in at most n iterations

Linear arithmetic equality invariant checking on simple programs is in PTIME

Invariant Checking for Unitary Theories

$e_1 = e_2$ is an invariant at point π if every program path to π gives an interpretation σ (for program variables) s.t. $\sigma \models e_1 = e_2$

Let $\sigma_1, \sigma_2, \ldots$ be all the interpretations reachable at π

Let σ be $mgu_{\mathbb{T}}(e_1, e_2)$. For all i,

$$\begin{array}{lll} & e_1\sigma_i =_{\mathbb{T}} e_2\sigma_i & \\ \text{Implies} & \sigma \text{ is more general than } \sigma_i & \\ \text{Implies} & \sigma\sigma_i =_{\mathbb{T}} \sigma_i & \\ \text{Implies} & x\sigma\sigma_i =_{\mathbb{T}} x\sigma_i & \text{for all } x \\ \text{Implies} & x\sigma = x \ \text{ is an invariant} & \end{array}$$

If $e_1 = e_2$ is an invariant, then $mgu_{\mathbb{T}}(e_1, e_2)$ is an invariant in the simple program model

Invariant Checking for Unitary Theories

Program P	:	Expression language of a unitary theory
Assertion	:	$e_1 = e_2$, where e_i are terms in the unitary theory

In this case,

- At each point, we have a conjunction of equations
- Such a conjunct can have at most n non-redundant equations (use unification)
- Therefore fixpoint converges in at most n iterations

Invariant checking of equalities on simple programs over unitary theories is in PTIME

Example: A Simple Program over UFS

```
[ $c = c$ ]
u := c; v := c;
[ $u = v$ ]
while (*) {
    [$F(u) = F(v)$] which is the same as [$u = v$]
    u := F(u);
    v := F(v);
    [$u = v$]
}
[$u = v$]
```

Note that $u = v$ is an invariant since all the following interpretations are models of it:

$$\langle u \mapsto c, v \mapsto c\rangle,\ \langle u \mapsto Fc, v \mapsto Fc\rangle,\ \langle u \mapsto FFc, v \mapsto FFc\rangle,\ \ldots$$

Disequality Invariant Checking is Undecidable

```
SolvePCP($(u_1, v_1), \ldots, (u_k, v_k)$):
$x := u_1(\epsilon)$;  $y := v_1(\epsilon)$;
while (*) {
    if (*) {
        $x := u_2(x)$;  $y := v_2(y)$;
    } elsif (*) {
        $x := u_3(x)$;  $y := v_3(y)$;
    } elsif (*) {
        $\vdots$
    } else {
        $x := u_k(x)$;  $y := v_k(y)$;
    }
}
[ $x \neq y$ ]
```

Disjunctive Equality Invariant Checking is coNP-hard

```
Solve3SAT($\psi$):
 $c_1 := 0$;  $\cdots$;  $c_m := 0$; // All clauses set to 0
 if (*) {
     All clauses containing $b_1$ set to 1
 } else {
     All clauses containing $\neg b_1$ set to 1
 }
 ⋮
 if (*) {
     All clauses containing $b_n$ set to 1
 } else {
     All clauses containing $\neg b_n$ set to 1
 }
 [ $c_1 = 0 \vee c_2 = 0 \vee \cdots \vee c_m = 0$ ];
```

Invariant holds iff at least one clause is not satisfied for each assignment

Equality Invariant Checking over UFS+LA

Recall the unification connection: For a simple program P over UFS+LA

$F(a) + F(b) = F(x) + F(a + b - x)$ is an invariant of P iff
$x = a \vee x = b$ is an invariant of P

Recursively using the same idea, we can write one equation $e_1 = e_2$ s.t.
$e_1 = e_2$ is an invariant of P iff
$0 = c_1 \vee 0 = c_2 \vee \cdots \vee 0 = c_m$ is an invariant of P

But checking this disjunctive assertion is coNP-hard

This proof generalizes to theories that can encode disjunction such as
$x = a \vee x = b$

Simple Programs over UFS+LA

Equality assertion checking is coNP-hard

We can show that it is decidable

The reason is that this theory is finitary

Hence backward propagation + unification can be shown to terminate

The argument generalizes to all convex and finitary theories

The result also generalizes richer program models that include assume disequality nodes

Richer Program Models

Additional edge labels:

- Assume($e_1 \neq e_2$)
- Assume($e_1 = e_2$)
- Call(P)

If we include conditionals, then even for simple programs using simple expression language (either UFS or LA), invariant checking is undecidable

Summary of Results

Unification type of theory of program expressions	Complexity of assertion checking	Examples
Strict Unitary	PTIME	ℓa, uf
Bitary	coNP-hard	ℓa+uf, c
Finitary-Convex	Decidable	ℓa+uf+c+ac

Figure 1: Results for simple programs. Row 4 holds even for disequality guards.

Summary

- Logical lattices are good candidates for thinking about and building abstract interpreters
- Logical lattices can be combined in a new and important way
  Logical Products:
  - Logical product is more powerful than direct or reduced product
  - Operations on logical lattices can be modularly combined to yield operations for logical products
  - Using ideas from the classical Nelson-Oppen combination method

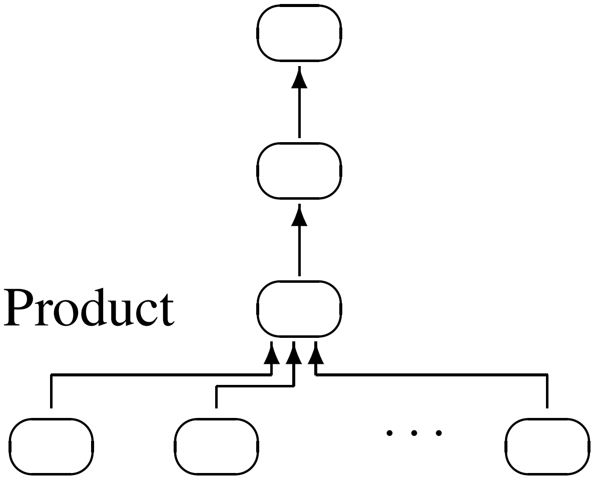
Summary

- The assertion checking problem:
  - Equations in an assertion can be replaced by its complete set of Th-unifiers for purposes of assertion checking
  - Assertion checking over “lattices” defined by combination of two logical lattices can be hard, even when it is in PTime for the lattices defined by individual theories
  - Finitary Th-unification algorithm implies decidability of assertion checking for the logical lattices defined by Th

Summary

- Base Abstract Domain $\mapsto$ Quantified Abstract Domain
- Require a rich interface from the base domain
- Ability to compute over- and under-approximations of various logical operators

Big Picture



Applications: Memory Safety

Quantified Abstract Domain

Combination Domain: Logical Product

Base Domains with rich API

Philosophy

Next Generation Automated Deduction Engine: Requirements–

Attributes	Why	Modern SMT Solvers
speed	embedded use	yes
support for theories	symbols have meaning	yes
interface	embedded use	lacking
beyond satisfiability	need more	lacking
reduced expressiveness		partly