

L26: Constraint-Based Analysis

Software Analysis

IIT Guwahati · October 14, 2026

Motivation (1/3)

Designing an efficient program analysis is challenging.

Program Analysis = **Specification** + **Implementation**

“What”

No null pointer is dereferenced
along any path in the program

Motivation (2/3)

Designing an efficient program analysis is challenging.

Program Analysis = **Specification** + **Implementation**

“What”

No null pointer is dereferenced
along any path in the program

“How”

Many design choices:
forward vs. backward traversal
symbolic vs. explicit representation, ...

Motivation (3/3)

Designing an efficient program analysis is challenging.

Program Analysis = **Specification** + **Implementation**

“What”

No null pointer is dereferenced
along any path in the program

“How”

Many design choices:
forward vs. backward traversal
symbolic vs. explicit representation, ...

Nontrivial! Consider null pointer dereference analysis.
No null pointer assignments (`v = null`): forward is best.
No pointer dereferences (`v->next`): backward is best.

What Is Constraint-Based Analysis? (1/2)

Designing an efficient program analysis is challenging.

Program Analysis = **Specification** + **Implementation**

“What”

Defined by the user in the
constraint **language**

What Is Constraint-Based Analysis? (2/2)

Designing an efficient program analysis is challenging.

Program Analysis = **Specification** + **Implementation**

“What”

Defined by the user in the
constraint **language**

“How”

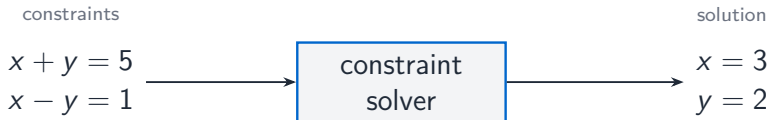
Automated by the
constraint **solver**

What Is a Constraint Solver? (1/3)



Input: a set of constraints over some variables.

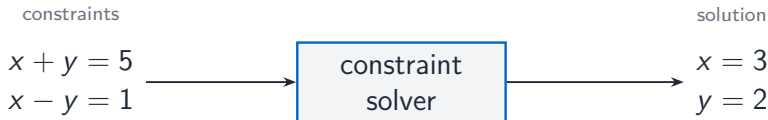
What Is a Constraint Solver? (2/3)



Input: a set of constraints over some variables.

Output: values that satisfy all of them, or “no solution”.

What Is a Constraint Solver? (3/3)



Input: a set of constraints over some variables.

Output: values that satisfy all of them, or “no solution”.

General-purpose: Soufflé (Datalog), MiniSAT (SAT¹), Z3 (SMT²).

¹ SAT, Boolean satisfiability: is there an assignment of true and false to the variables that makes a Boolean formula true? ² SMT, satisfiability modulo theories: the same question for formulas that also contain integers, reals, arrays and other data.

Benefits of Constraint-Based Analysis (1/3)

- Separates analysis specification from implementation

Analysis writer can focus on “what” rather than “how”

Benefits of Constraint-Based Analysis (2/3)

- Separates analysis specification from implementation

Analysis writer can focus on “what” rather than “how”

- Yields natural program specifications

Constraints are usually local, whose conjunctions capture global properties

Benefits of Constraint-Based Analysis (3/3)

- Separates analysis specification from implementation

Analysis writer can focus on “what” rather than “how”

- Yields natural program specifications

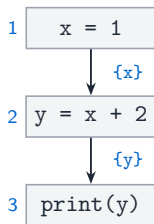
Constraints are usually local, whose conjunctions capture global properties

- Enables sophisticated analysis implementations

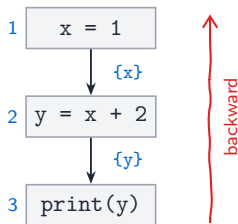
Leverage powerful, off-the-shelf solvers

Recall: Live Variables Analysis (1/4)

Objective: which variables are live at each point, that is, may be read later?



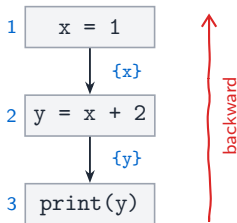
Recall: Live Variables Analysis (2/4)



Objective: which variables are live at each point, that is, may be read later?

How: backward, union over successors, chaotic iteration.

Recall: Live Variables Analysis (3/4)



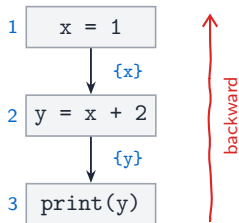
Objective: which variables are live at each point, that is, may be read later?

How: backward, union over successors, chaotic iteration.

$$IN[n] = (OUT[n] - KILL[n]) \cup GEN[n]$$

$$OUT[n] = \bigcup_{n' \in successors(n)} IN[n']$$

Recall: Live Variables Analysis (4/4)



The sets for this program

n	statement	GEN	KILL	OUT	IN
1	$x = 1$	$\emptyset$	$\{x\}$	$\{x\}$	$\emptyset$
2	$y = x + 2$	$\{x\}$	$\{y\}$	$\{y\}$	$\{x\}$
3	$\text{print}(y)$	$\{y\}$	$\emptyset$	$\emptyset$	$\{y\}$

GEN $[n]$: the variables that n reads.

KILL $[n]$: the variables that n assigns.

QUIZ: Specification & Implementation (1/2)

Consider a dataflow analysis such as **live variables analysis**. If one expresses it as a constraint-based analysis, one must still decide:

- ☐ The order in which statements should be processed.
- ☐ What the gen and kill sets for each kind of statement are.
- ☐ In what language to implement the chaotic iteration algorithm.
- ☐ Whether to take intersection or union at merge points.

QUIZ: Specification & Implementation (2/2)

Consider a dataflow analysis such as **live variables analysis**. If one expresses it as a constraint-based analysis, one must still decide:

- ☐ The order in which statements should be processed.
- ☒ What the gen and kill sets for each kind of statement are.
- ☐ In what language to implement the chaotic iteration algorithm.
- ☒ Whether to take intersection or union at merge points.

A Constraint Language: Datalog (1/5)

- A declarative logic programming language

A Constraint Language: Datalog (2/5)

- A declarative logic programming language
- **Not Turing-complete**¹: subset of Prolog, or SQL with recursion
⇒ Efficient algorithms to evaluate Datalog programs

¹ Turing-complete: able to express every computation that a general-purpose computer can perform; such a program need not terminate. A Datalog program always terminates on finite input.

A Constraint Language: Datalog (3/5)

- A declarative logic programming language
- **Not Turing-complete**¹: subset of Prolog, or SQL with recursion
⇒ Efficient algorithms to evaluate Datalog programs
- Originated as query language for deductive databases

¹ Turing-complete: able to express every computation that a general-purpose computer can perform; such a program need not terminate. A Datalog program always terminates on finite input.

A Constraint Language: Datalog (4/5)

- A declarative logic programming language
- **Not Turing-complete**¹: subset of Prolog, or SQL with recursion
⇒ Efficient algorithms to evaluate Datalog programs
- Originated as query language for deductive databases
- Later applied in many other domains: **software analysis**, data mining, networking, security, knowledge representation, cloud computing, ...

¹ Turing-complete: able to express every computation that a general-purpose computer can perform; such a program need not terminate. A Datalog program always terminates on finite input.

A Constraint Language: Datalog (5/5)

- A declarative logic programming language
- **Not Turing-complete**¹: subset of Prolog, or SQL with recursion
⇒ Efficient algorithms to evaluate Datalog programs
- Originated as query language for deductive databases
- Later applied in many other domains: **software analysis**, data mining, networking, security, knowledge representation, cloud computing, ...
- Many implementations: Soufflé, bddbdb, IRIS, ...

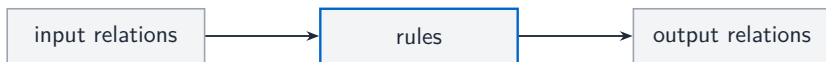
¹ Turing-complete: able to express every computation that a general-purpose computer can perform; such a program need not terminate. A Datalog program always terminates on finite input.

Relations in a Datalog Program (1/3)



Input relation: facts supplied to the program before evaluation.

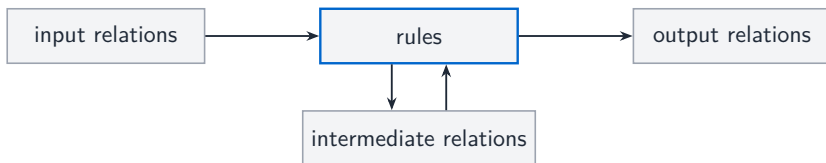
Relations in a Datalog Program (2/3)



Input relation: facts supplied to the program before evaluation.

Output relation: facts the program is expected to derive using rules.

Relations in a Datalog Program (3/3)



Input relation: facts supplied to the program before evaluation.

Output relation: facts the program is expected to derive using rules.

Intermediate relation: derived facts used to compute other relations.

Syntax of Datalog: Example (1/5)

Problem: **graph reachability**. Which pairs of nodes are connected by a path?

Input Relations:

`edge(n:N, m:N)`

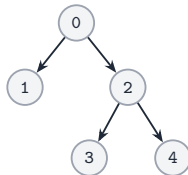
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`



0 reaches 4. 3 does not reach 0.

Syntax of Datalog: Example (2/5)

Problem: **graph reachability**. Which pairs of nodes are connected by a path?

Input Relations:

`edge(n:N, m:N)`

Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`

A relation is similar to a table in a database. A tuple in a relation is similar to a row in a table.

Syntax of Datalog: Example (3/5)

Problem: **graph reachability**. Which pairs of nodes are connected by a path?

Input Relations:

`edge(n:N, m:N)`

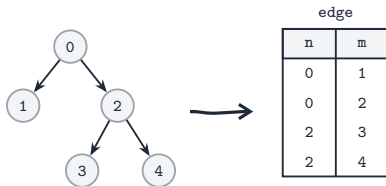
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`



Syntax of Datalog: Example (4/5)

Problem: **graph reachability**. Which pairs of nodes are connected by a path?

Input Relations:

`edge(n:N, m:N)`

Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`

Deductive rules that hold
universally

(i.e., variables like x, y, z can be
replaced by any constant).

Specify “if ... then ...” logic.

Syntax of Datalog: Example (5/5)

Problem: **graph reachability**. Which pairs of nodes are connected by a path?

Input Relations:

`edge(n:N, m:N)`

Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`

(If TRUE,) there is a path from each node to itself.

If there is a path from node x to y , and there is an edge from y to z , then there is a path from x to z .

Reading a Datalog Rule (1/6)

path(x, z) :- path(x, y) , edge(y, z) .
conclusion

Reading a Datalog Rule (2/6)

read as "if"

$\downarrow$

path(x, z) :- path(x, y) , edge(y, z) .

conclusion hypotheses

Reading a Datalog Rule (3/6)

$$\text{path}(x, z) \text{ :- } \text{path}(x, y), \text{ edge}(y, z) .$$

read as "if" $\downarrow$ AND $\downarrow$
 conclusion hypotheses

Reading a Datalog Rule (4/6)

read as "if" AND

path(x, z) :- path(x, y), edge(y, z) .

conclusion hypotheses

x, y, z are variables: the rule holds for all nodes. A repeated variable is the same node.

Reading a Datalog Rule (5/6)

$\text{path}(x, z) \text{ :- path}(x, y), \text{edge}(y, z).$

read as "if" AND ends the rule

conclusion hypotheses

x, y, z are variables: the rule holds for all nodes. A repeated variable is the same node.

Reading a Datalog Rule (6/6)

$\text{path}(x, z) \text{ :- } \text{path}(x, y), \text{edge}(y, z) \text{ .}$

read as "if" AND ends the rule

conclusion hypotheses

x, y, z are variables: the rule holds for all nodes. A repeated variable is the same node.

No hypotheses, as in $\text{path}(x, x) \therefore$ an **axiom**.

Semantics of Datalog: Example

Input Relations:

`edge(n:N, m:N)`

Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y), edge(y, z).`

```
path := { (x, x) | x ∈ N }
```

```
do
```

```
  path := path ∪ { (x, z) | ∃ y ∈ N:  
                      (x, y) ∈ path and (y, z) ∈ edge }
```

```
until path relation stops changing
```

Semantics of Datalog: Example Run (1/4)

Input Relations:

`edge(n:N, m:N)`

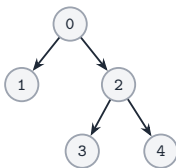
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y),
edge(y, z).`



Input Tuples:

`edge(0, 1), edge(0, 2), edge(2, 3),
edge(2, 4)`

Output Tuples:

`path(0, 0), path(1, 1), path(2, 2),
path(3, 3), path(4, 4), path(0, 1),
path(0, 2), path(2, 3), path(2, 4),
path(0, 3), path(0, 4)`

Semantics of Datalog: Example Run (2/4)

Input Relations:

`edge(n:N, m:N)`

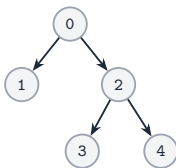
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y),
edge(y, z).`



Input Tuples:

`edge(0, 1), edge(0, 2), edge(2, 3),
edge(2, 4)`

Output Tuples:

`path(0, 0), path(1, 1), path(2, 2),
path(3, 3), path(4, 4), path(0, 1),
path(0, 2), path(2, 3), path(2, 4),
path(0, 3), path(0, 4)`

Semantics of Datalog: Example Run (3/4)

Input Relations:

`edge(n:N, m:N)`

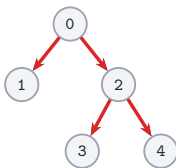
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y),
edge(y, z).`



Input Tuples:

`edge(0, 1), edge(0, 2), edge(2, 3),
edge(2, 4)`

Output Tuples:

`path(0, 0), path(1, 1), path(2, 2),
path(3, 3), path(4, 4), path(0, 1),
path(0, 2), path(2, 3), path(2, 4),
path(0, 3), path(0, 4)`

Semantics of Datalog: Example Run (4/4)

Input Relations:

`edge(n:N, m:N)`

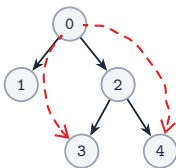
Output Relations:

`path(n:N, m:N)`

Rules:

`path(x, x).`

`path(x, z) :- path(x, y),
edge(y, z).`



Input Tuples:

`edge(0, 1), edge(0, 2), edge(2, 3),
edge(2, 4)`

Output Tuples:

`path(0, 0), path(1, 1), path(2, 2),
path(3, 3), path(4, 4), path(0, 1),
path(0, 2), path(2, 3), path(2, 4),
path(0, 3), path(0, 4)`

For Fun: What Is a Soufflé? (1/3)



A **soufflé** (French pronunciation: [sufle]) is a baked egg dish originating in France in the early 18th century.

Combined with various other ingredients, it can be a savoury main dish or a sweet dessert.

Text: Wikipedia, "Soufflé". Photo: Mark Miller, CC BY-SA 4.0, Wikimedia Commons

For Fun: What Is a Soufflé? (2/3)



A **soufflé** (French pronunciation: [sufle]) is a baked egg dish originating in France in the early 18th century.

Combined with various other ingredients, it can be a savoury main dish or a sweet dessert.

The word comes from *souffler*: to blow, to puff up.

Text: Wikipedia, "Soufflé". Photo: Mark Miller, CC BY-SA 4.0, Wikimedia Commons

For Fun: What Is a Soufflé? (3/3)



A **soufflé** (French pronunciation: [sufle]) is a baked egg dish originating in France in the early 18th century.

Combined with various other ingredients, it can be a savoury main dish or a sweet dessert.

The word comes from *souffler*: to blow, to puff up.

The Datalog engine: the facts are the uncooked soufflé, and the rules make it rise.

Text: Wikipedia, "Soufflé". Photo: Mark Miller, CC BY-SA 4.0, Wikimedia Commons

The Soufflé Datalog Engine (1/4)

- A logic language inspired by Datalog, built for **large-scale static analysis**



Systematic, Ontological, Undiscovered

Fact Finding Logic Engine

The Soufflé Datalog Engine (2/4)



Systematic, Ontological, Undiscovered

Fact Finding Logic Engine

- A logic language inspired by Datalog, built for **large-scale static analysis**
- **Creator:** project led by Prof. Bernhard Scholz; it commenced at Oracle Labs, Brisbane

The Soufflé Datalog Engine (3/4)



Soufflé

Systematic, Ontological, Undiscovered
Fact Finding Logic Engine

- A logic language inspired by Datalog, built for **large-scale static analysis**
- **Creator**: project led by Prof. Bernhard Scholz; it commenced at Oracle Labs, Brisbane
- Rules are compiled to **parallel C++**

The Soufflé Datalog Engine (4/4)



Soufflé

Systematic, Ontological, Undiscovered
Fact Finding Logic Engine

- A logic language inspired by Datalog, built for **large-scale static analysis**
- **Creator**: project led by Prof. Bernhard Scholz; it commenced at Oracle Labs, Brisbane
- Rules are compiled to **parallel C++**
- **Current state**: open source since March 2016; latest release 2.5 (March 2025); actively developed

souffle-lang.github.io

Reachability in Soufflé (1/2)

reach.dl

```
.decl node(n:number)
.decl edge(n:number, m:number)
.decl path(n:number, m:number)
.output path(IO=stdout)

node(0). node(1). node(2). node(3). node(4).
edge(0, 1). edge(0, 2). edge(2, 3). edge(2, 4).

path(x, x) :- node(x).
path(x, z) :- path(x, y), edge(y, z).
```

```
$ souffle reach.dl
```

```
path
```

```
0 0
```

```
0 1
```

```
0 2
```

```
0 3
```

```
0 4
```

```
1 1
```

```
2 2
```

```
2 3
```

```
2 4
```

```
3 3
```

```
4 4
```

Reachability in Soufflé (2/2)

reach.dl

```
.decl node(n:number)
.decl edge(n:number, m:number)
.decl path(n:number, m:number)
.output path(IO=stdout)

node(0). node(1). node(2). node(3). node(4).
edge(0, 1). edge(0, 2). edge(2, 3). edge(2, 4).

path(x, x) :- node(x).
path(x, z) :- path(x, y), edge(y, z).
```

```
$ souffle reach.dl
```

```
path
```

```
=====
0  0
0  1
0  2
0  3
0  4
1  1
2  2
2  3
2  4
3  3
4  4
=====
```

.decl declares a relation. Soufflé needs x bound by a hypothesis, hence `node(x)`.

Dataflow Analysis in Datalog

Recall the specification of **reaching definitions analysis**:

Objective: which definitions (assignments) may reach each label?

How: forward, union over predecessors.

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Reaching Definitions Analysis in Datalog (1/7)

Input Relations:

`kill(n:N, d:D)`

Output Relations:

Rules:

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Definition of statement

Reaching Definitions Analysis in Datalog (2/7)

Input Relations:

`kill(n:N, d:D)`

`gen (n:N, d:D)`

Output Relations:

Rules:

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)}$$

Definition d
state

Reaching Definitions Analysis in Datalog (3/7)

Input Relations:

kill($n:N$, $d:D$)

gen ($n:N$, $d:D$)

next($n:N$, $m:N$)

Output Relations:

Rules:

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$
$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Statement m
successor c

Reaching Definitions Analysis in Datalog (4/7)

Input Relations:

kill($n:N, d:D$)

gen ($n:N, d:D$)

next($n:N, m:N$)

Output Relations:

in ($n:N, d:D$)

Rules:

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Definition
point

Reaching Definitions Analysis in Datalog (5/7)

Input Relations:

kill($n:N$, $d:D$)

gen ($n:N$, $d:D$)

next($n:N$, $m:N$)

Output Relations:

in ($n:N$, $d:D$)

out($n:N$, $d:D$)

Rules:

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Definition
point

Reaching Definitions Analysis in Datalog (6/7)

Input Relations:

```
kill(n:N, d:D)  
gen (n:N, d:D)  
next(n:N, m:N)
```

Output Relations:

```
in (n:N, d:D)  
out(n:N, d:D)
```

Rules:

```
out(n, d) :- gen(n, d).  
out(n, d) :- in(n, d), !kill(n, d).
```

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

Reaching Definitions Analysis in Datalog (7/7)

Input Relations:

```
kill(n:N, d:D)  
gen (n:N, d:D)  
next(n:N, m:N)
```

Output Relations:

```
in (n:N, d:D)  
out(n:N, d:D)
```

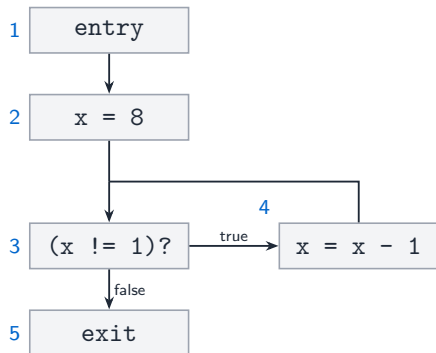
Rules:

```
out(n, d) :- gen(n, d).  
out(n, d) :- in(n, d), !kill(n, d).  
in (m, d) :- out(n, d), next(n, m).
```

$$\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]$$

$$\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']$$

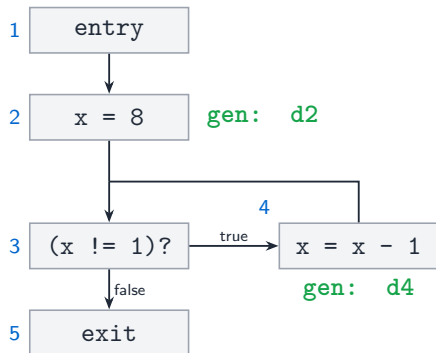
Reaching Definitions Analysis: Example (1/5)



Definitions are named by their label:

d2: $x = 8$ d4: $x = x - 1$

Reaching Definitions Analysis: Example (2/5)

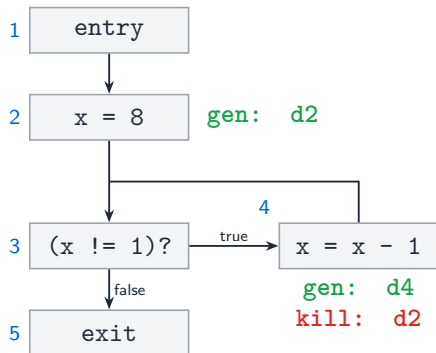


gen: gen(2, 2), gen(4, 4)

A label that assigns a variable generates that definition.

Labels 1, 3 and 5 assign nothing.

Reaching Definitions Analysis: Example (3/5)

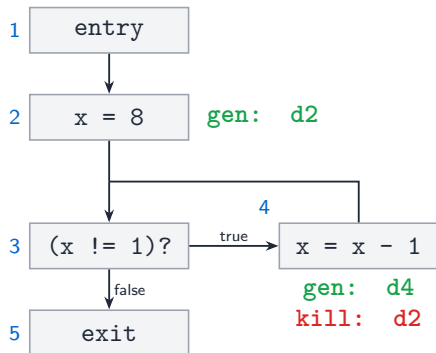


kill: kill(4, 2)

Label 4 assigns `x` again, and `d2`, another definition of `x`, can reach label 4.

Not kill(2, 4): no path leads from label 4 to label 2.

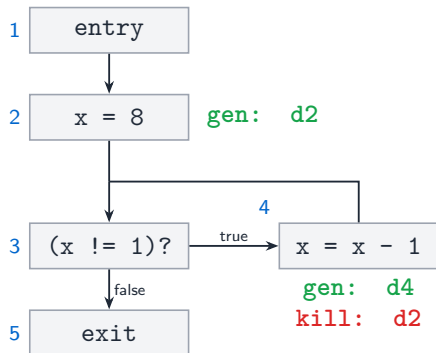
Reaching Definitions Analysis: Example (4/5)



next: one tuple per edge

`next(1, 2), next(2, 3), next(3, 4),`
`next(3, 5), next(4, 3)`

Reaching Definitions Analysis: Example (5/5)



Output tuples

in(3, 2), in(3, 4), in(4, 2),
in(4, 4), in(5, 2), in(5, 4),

out(2, 2), out(3, 2), out(3, 4),
out(4, 4), out(5, 2), out(5, 4)

in(4, 2) but not out(4, 2): killed at label 4.

Reaching Definitions in Soufflé (1/2)

`reachdef.dl` (declarations omitted)

```
.output in(IO=stdout)
.output out(IO=stdout)
next(1, 2).  next(2, 3).  next(3, 4).
next(3, 5).  next(4, 3).
gen(2, 2).   gen(4, 4).
kill(4, 2).
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).
in(m, d)   :- out(n, d), next(n, m).
```

```
$ souffle reachdef.dl
```

```
-----
in
=====
```

```
3  2
3  4
4  2
4  4
5  2
5  4
=====
```

```
-----
out
=====
```

```
2  2
3  2
3  4
4  4
5  2
5  4
=====
```

Reaching Definitions in Soufflé (2/2)

`reachdef.dl` (declarations omitted)

```
.output in(IO=stdout)
.output out(IO=stdout)
next(1, 2). next(2, 3). next(3, 4).
next(3, 5). next(4, 3).
gen(2, 2). gen(4, 4).
kill(4, 2).
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).
in(m, d) :- out(n, d), next(n, m).
```

```
$ souffle reachdef.dl
```

```
-----
in
=====
```

```
3 2
3 4
4 2
4 4
5 2
5 4
=====
```

```
-----
out
=====
```

```
2 2
3 2
3 4
4 4
5 2
5 4
=====
```

Exercise. Delete `kill(4, 2)`. Which tuple is new, and what does it claim?

QUIZ: Live Variables Analysis (1/2)

Complete the Datalog program below by filling in the rules for **live variables analysis**.

Input Relations:

kill(n:N, v:V)
gen (n:N, v:V)
next(n:N, m:N)

Output Relations:

in (n:N, v:V)
out(n:N, v:V)

Rules:

:- .

:- , ! .

:- , .

QUIZ: Live Variables Analysis (2/2)

Complete the Datalog program below by filling in the rules for **live variables analysis**.

Input Relations:

kill(n:N, v:V)
gen (n:N, v:V)
next(n:N, m:N)

Output Relations:

in (n:N, v:V)
out(n:N, v:V)

Rules:

`in(n, v) :- gen(n, v) .`

`in(n, v) :- out(n, v) , ! kill(n, v) .`

`out(n, v) :- in(m, v) , next(n, m) .`

Pointer Analysis in Datalog

Consider a flow-insensitive **may-alias analysis** for a simple language:

(function body)	$f(v) \{ s1, \dots, sn \}$
(statement)	$s ::= v = \text{new } h \mid v = u$ $\mid \text{return } u \mid v = f(u)$
(pointer variable)	u, v
(allocation site)	h
(function name)	f

Pointer Analysis in Datalog: Intra-Procedural (1/6)

(function body) $f(v) \{ s1, \dots, sn \}$

```
(statement)  s ::= v = new h  |  v = u
              |  return u    |  v = f(u)
```

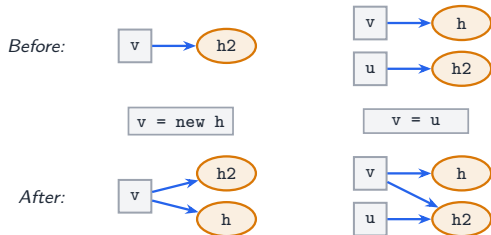
(pointer variable) u, v

(allocation site) h

(function name) f

Pointer Analysis in Datalog: Intra-Procedural (2/6)

Recall the specification:



Pointer Analysis in Datalog: Intra-Procedural (3/6)

Input Relations:

new (v:V, h:H)

assign(v:V, u:V)

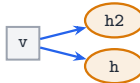
Before:



v = new h

v = u

After:



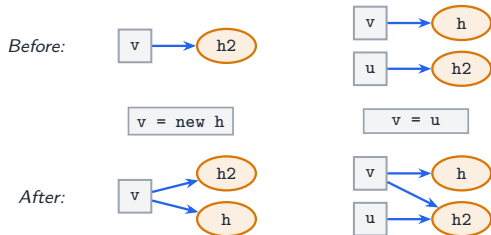
Pointer Analysis in Datalog: Intra-Procedural (4/6)

Input Relations:

`new (v:V, h:H)`
`assign(v:V, u:V)`

Output Relations:

`points(v:V, h:H)`



Pointer Analysis in Datalog: Intra-Procedural (5/6)

Input Relations:

```
new    (v:V, h:H)  
assign(v:V, u:V)
```

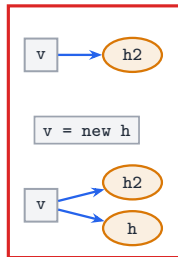
Output Relations:

```
points(v:V, h:H)
```

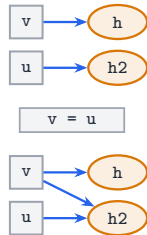
Rules:

```
points(v, h) :- new(v, h).
```

Before:



After:



Pointer Analysis in Datalog: Intra-Procedural (6/6)

Input Relations:

```
new    (v:V, h:H)
assign(v:V, u:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

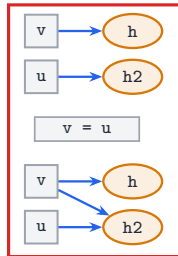
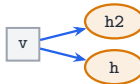
```
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
```

Before:



```
v = new h
```

After:



Points-to Analysis in Soufflé (1/2)

```
x = new h1;  
y = x;  
z = new h2;  
y = z;  
w = y;
```

```
new("x", "h1").    new("z", "h2").  
assign("y", "x").  assign("y", "z").  
assign("w", "y").  
  
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).
```

```
$ souffle pointsto.dl  
-----  
points  
=====
```

y	h1
y	h2
x	h1
z	h2
w	h1
w	h2

```
=====
```

Points-to Analysis in Soufflé (2/2)

```
x = new h1;  
y = x;  
z = new h2;  
y = z;  
w = y;
```

```
new("x", "h1").    new("z", "h2").  
assign("y", "x").  assign("y", "z").  
assign("w", "y").  
  
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).
```

```
$ souffle pointsto.dl  
-----  
points  
=====
```

y	h1
y	h2
x	h1
z	h2
w	h1
w	h2

```
=====
```

One fact per statement. *y* gets both sites, and *w* copies *y*.

Pointer Analysis in Datalog: Inter-Procedural (1/7)

So far: only $v = \text{new } h$ and $v = u$, in one function.

Now: **calls** and **returns**, across functions.

```
(function body)  f(v) { s1, ..., sn }
```

```
(statement)  s ::= v = new h   |   v = u
              |   return u     |   v = f(u)
```

(pointer variable) u, v

(allocation site) h

(function name) f

Pointer Analysis in Datalog: Inter-Procedural (2/7)

Input Relations:

`new(v:V, h:H)`
`assign(v:V, u:V)`

Output Relations:

`points(v:V, h:H)`

Rules:

`points(v, h) :- new(v, h).`
`points(v, h) :- assign(v, u), points(u, h).`

```
x = new h1;  
y = f(x);  
f(v) {  
    u = v;  
    return u;  
}
```

Pointer Analysis in Datalog: Inter-Procedural (3/7)

Input Relations:

```
new(v:V, h:H)  
assign(v:V, u:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

```
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).
```

```
x = new h1;  
y = f(x);  
f(v) {  
    u = v;  
    return u;  
}
```

**Parameter passing and return
can be treated as assignments!**

Pointer Analysis in Datalog: Inter-Procedural (4/7)

Input Relations:

```
new(v:V, h:H)  
assign(v:V, u:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

```
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).
```

```
v = x  
u = v  
y = u
```

```
x = new h1;  
y = f(x);  
f(v) {  
    u = v;  
    return u;  
}
```

Pointer Analysis in Datalog: Inter-Procedural (5/7)

Input Relations:

```
new(v:V, h:H) arg(f:F, v:V) ret(f:F, u:V)  
assign(v:V, u:V) call(y:V, f:F, x:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

```
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).
```

```
v = x  
u = v  
y = u
```

```
x = new h1;  
y = f(x);  
f(v) {  
    u = v;  
    return u;  
}
```

`call(y,f,x), arg(f,v), ret(f,u)`

Pointer Analysis in Datalog: Inter-Procedural (6/7)

Input Relations:

```
new(v:V, h:H) arg(f:F, v:V) ret(f:F, u:V)
assign(v:V, u:V) call(y:V, f:F, x:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

```
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
```

```
v = x
u = v
y = u
```

```
x = new h1;
y = f(x);
f(v) {
  u = v;
  return u;
}
```

Wildcard, “don’t care”

Pointer Analysis in Datalog: Inter-Procedural (7/7)

Input Relations:

```
new(v:V, h:H) arg(f:F, v:V) ret(f:F, u:V)
assign(v:V, u:V) call(y:V, f:F, x:V)
```

Output Relations:

```
points(v:V, h:H)
```

Rules:

```
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
points(y, h) :- call(y, f, _), ret(f, u), points(u, h).
```

```
v = x
u = v
y = u
```

```
x = new h1;
y = f(x);
f(v) {
  u = v;
  return u;
}
```

Inter-Procedural Analysis in Soufflé (1/3)

```
x = new h1;
y = f(x);

f(v) {
  u = v;
  return u;
}
```

call1.dl

```
.decl new(v:symbol, h:symbol)
.decl assign(v:symbol, u:symbol)
.decl arg(f:symbol, v:symbol)
.decl ret(f:symbol, u:symbol)
.decl call(y:symbol, f:symbol, x:symbol)
.decl points(v:symbol, h:symbol)
.output points(IO=stdout)

new("x", "h1").
assign("u", "v").
arg("f", "v").  ret("f", "u").
call("y", "f", "x").

points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v),
                  points(x, h).
points(y, h) :- call(y, f, _), ret(f, u),
                  points(u, h).
```

```
$ souffle call1.dl
-----
points
=====
v   h1
u   h1
y   h1
x   h1
=====
```

Inter-Procedural Analysis in Soufflé (2/3)

```
x = new h1;  
y = f(x);  
  
f(v) {  
    u = v;  
    return u;  
}
```

call1.dl

```
.decl new(v:symbol, h:symbol)  
.decl assign(v:symbol, u:symbol)  
.decl arg(f:symbol, v:symbol)  
.decl ret(f:symbol, u:symbol)  
.decl call(y:symbol, f:symbol, x:symbol)  
.decl points(v:symbol, h:symbol)  
.output points(IO=stdout)  
  
new("x", "h1").  
assign("u", "v").  
arg("f", "v").    ret("f", "u").  
call("y", "f", "x").  
  
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).  
points(v, h) :- call(_, f, x), arg(f, v),  
                  points(x, h).  
points(y, h) :- call(y, f, _), ret(f, u),  
                  points(u, h).
```

```
$ souffle call1.dl  
-----  
points  
=====
```

v	h1
u	h1
y	h1
x	h1

```
=====
```

h1 flows $x \rightarrow v \rightarrow u \rightarrow y$.

Inter-Procedural Analysis in Soufflé (3/3)

```
x = new h1;  
y = f(x);  
  
f(v) {  
    u = v;  
    return u;  
}
```

call1.dl

```
.decl new(v:symbol, h:symbol)  
.decl assign(v:symbol, u:symbol)  
.decl arg(f:symbol, v:symbol)  
.decl ret(f:symbol, u:symbol)  
.decl call(y:symbol, f:symbol, x:symbol)  
.decl points(v:symbol, h:symbol)  
.output points(IO=stdout)  
  
new("x", "h1").  
assign("u", "v").  
arg("f", "v").    ret("f", "u").  
call("y", "f", "x").  
  
points(v, h) :- new(v, h).  
points(v, h) :- assign(v, u), points(u, h).  
points(v, h) :- call(_, f, x), arg(f, v),  
                  points(x, h).  
points(y, h) :- call(y, f, _), ret(f, u),  
                  points(u, h).
```

```
$ souffle call1.dl  
-----  
points  
=====
```

v	h1
u	h1
y	h1
x	h1

```
=====
```

h1 flows $x \rightarrow v \rightarrow u \rightarrow y$.

Exercise. Add `new("z", "h2").` and `call("w", "f", "z").` What does y point to now?

Context Sensitivity (1/4)

```
x = new h1;  
z = new h2;  
y = f(x);  
w = f(z);  
f(v) {  
    u = v;  
    return u;  
}
```

Context Sensitivity (2/4)

```
v = x
```

```
u = v
```

```
y = u
```

```
v = z
```

```
u = v
```

```
w = u
```

```
x = new h1;
```

```
z = new h2;
```

```
y = f(x);
```

```
w = f(z);
```

```
f(v) {
```

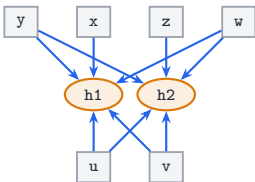
```
    u = v;
```

```
    return u;
```

```
}
```

Naik, CIS 5470, Penn: Constraint-Based Analysis

Context Sensitivity (3/4)

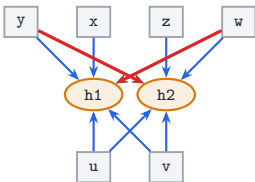


```
v = x
u = v
y = u
v = z
u = v
w = u
```

```
x = new h1;
z = new h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}
```

Naik, CIS 5470, Penn: Constraint-Based Analysis

Context Sensitivity (4/4)



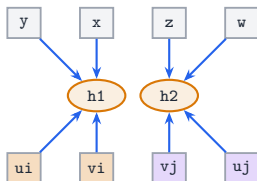
Imprecision!

```
v = x  
u = v  
y = u  
  
v = z  
u = v  
w = u
```

```
x = new h1;  
z = new h2;  
y = f(x);  
w = f(z);  
f(v) {  
    u = v;  
    return u;  
}
```

Naik, CIS 5470, Penn: Constraint-Based Analysis

Cloning-Based Inter-Procedural Analysis (1/2)



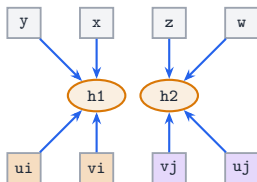
```
vi = x  
ui = vi  
y = ui
```

```
vj = z  
uj = vj  
w = uj
```

```
x = new h1;  
z = new h2;  
y = f(x); // i  
w = f(z); // j  
f(v) {  
    u = v;  
    return u;  
}
```

Achieves context sensitivity by **inlining** procedure calls

Cloning-Based Inter-Procedural Analysis (2/2)



```
vi = x  
ui = vi  
y = ui  
  
vj = z  
uj = vj  
w = uj
```

```
x = new h1;  
z = new h2;  
y = f(x); // i  
w = f(z); // j  
f(v) {  
    u = v;  
    return u;  
}
```

Achieves context sensitivity by **inlining** procedure calls

Cloning depth $\uparrow$: precision $\uparrow$ vs. scalability $\downarrow$

What about Recursion?

```
x = new h1;  
z = new h2;  
y = f(x);  
w = f(z);  
  
f(v) {  
    if (*)  
        v = f(v);  
    return v;  
}
```

Need **infinite** cloning depth to
differentiate the points-to sets
of x, y and w, z!

Summary-Based Inter-Procedural Analysis (1/3)

- Use the incoming program states to differentiate calls to the same procedure

Summary-Based Inter-Procedural Analysis (2/3)

- Use the incoming program states to differentiate calls to the same procedure
 Same **incoming** program states yield same **outgoing** program states for a given procedure

Summary-Based Inter-Procedural Analysis (3/3)

- Use the incoming program states to differentiate calls to the same procedure
Same **incoming** program states yield same **outgoing** program states for a given procedure
- As precise as cloning-based analysis with *infinite* cloning depth!

Further reading: T. Reps, S. Horwitz, M. Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. POPL 1995.

Constraint Languages

Constraint Language	Problem Expressed	Example Solvers
Datalog	Least solution of deductive inference rules	Soufflé, bddbddb
SAT	Boolean satisfiability problem	MiniSAT, Glucose
MaxSAT	Boolean satisfiability problem extended with optimization	Open-WBO
SMT	Satisfiability modulo theories problem	Z3, CVC4, Yices
MaxSMT	Satisfiability modulo theories problem extended with optimization	Z3

Boolean Satisfiability Problem (SAT) (1/3)

- Given a formula F over Boolean variables $x_1, x_2, \dots, x_n$: is F satisfiable?

F is **satisfiable** $\Leftrightarrow$ there exists an assignment to $x_1, x_2, \dots, x_n$ such that F **evaluates to TRUE**

Boolean Satisfiability Problem (SAT) (2/3)

- Given a formula F over Boolean variables $x_1, x_2, \dots, x_n$: is F satisfiable?

F is **satisfiable** $\Leftrightarrow$ there exists an assignment to $x_1, x_2, \dots, x_n$ such that F **evaluates to TRUE**

$$(A \vee B \vee C) \wedge (\neg B \vee \neg C \vee \neg E) \wedge (B \vee D)$$

CNF: **an AND of ORs**

Boolean Satisfiability Problem (SAT) (3/3)

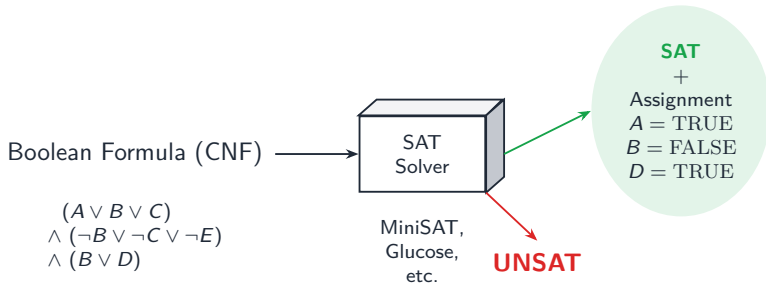
- Given a formula F over Boolean variables $x_1, x_2, \dots, x_n$: is F satisfiable?

F is **satisfiable** $\Leftrightarrow$ there exists an assignment to $x_1, x_2, \dots, x_n$ such that F **evaluates to TRUE**

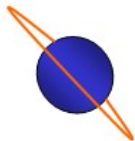
$(A \vee B \vee C) \wedge (\neg B \vee \neg C \vee \neg E) \wedge (B \vee D)$ CNF: **an AND of ORs**

A satisfying assignment: $A = \text{TRUE}, B = \text{FALSE}, D = \text{TRUE}$

SAT Solvers

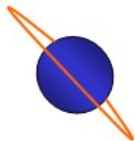


SATURN (1/3)



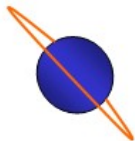
- SAT-based framework for static bug detection
- Stanford, Alex Aiken's group: started 2004; final release 1.2 (2010)

SATURN (2/3)



- SAT-based framework for static bug detection
- Stanford, Alex Aiken's group: started 2004; final release 1.2 (2010)
- Found hundreds of previously unknown bugs in the Linux kernel, with a lower false positive rate than previous efforts

SATURN (3/3)



- SAT-based framework for static bug detection
- Stanford, Alex Aiken's group: started 2004; final release 1.2 (2010)
- Found hundreds of previously unknown bugs in the Linux kernel, with a lower false positive rate than previous efforts



Example: Static Bug-Finding (1/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Example: Static Bug-Finding (2/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 1: bit-blasting x and y

x	<table><tr><td>x_{31}</td><td>$\dots$</td><td>x_0</td></tr></table>	x_{31}	$\dots$	x_0
x_{31}	$\dots$	x_0		
y	<table><tr><td>y_{31}</td><td>$\dots$</td><td>y_0</td></tr></table>	y_{31}	$\dots$	y_0
y_{31}	$\dots$	y_0		

Example: Static Bug-Finding (3/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 1: bit-blasting x and y

x	<table><tr><td>x_{31}</td><td>$\dots$</td><td>x_0</td></tr></table>	x_{31}	$\dots$	x_0
x_{31}	$\dots$	x_0		
y	<table><tr><td>y_{31}</td><td>$\dots$</td><td>y_0</td></tr></table>	y_{31}	$\dots$	y_0
y_{31}	$\dots$	y_0		

Step 2: computing z

z	<table><tr><td>$x_{31} \vee y_{31}$</td><td>$\dots$</td><td>$x_0 \vee y_0$</td></tr></table>	$x_{31} \vee y_{31}$	$\dots$	$x_0 \vee y_0$
$x_{31} \vee y_{31}$	$\dots$	$x_0 \vee y_0$		

Example: Static Bug-Finding (4/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 3: is the denominator ever zero?

$z = 0$ means every bit of z is 0.

$$F \equiv (x_{31} \vee y_{31}) = 0$$

$$\wedge \dots$$

$$\wedge (x_0 \vee y_0) = 0$$

not CNF

Example: Static Bug-Finding (5/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 3: is the denominator ever zero?

$z = 0$ means every bit of z is 0.

$$\begin{array}{ccc} F \equiv (x_{31} \vee y_{31}) = 0 & & F \equiv \neg(x_{31} \vee y_{31}) \\ \wedge \dots & \Rightarrow & \wedge \dots \\ \wedge (x_0 \vee y_0) = 0 & & \wedge \neg(x_0 \vee y_0) \\ \text{not CNF} & & \text{not CNF} \end{array}$$

$a = 0$ is $\neg a$ “ a is false”

Example: Static Bug-Finding (6/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 3: is the denominator ever zero?

$z = 0$ means every bit of z is 0.

$$\begin{array}{ccccc} F \equiv (x_{31} \vee y_{31}) = 0 & & F \equiv \neg(x_{31} \vee y_{31}) & & F \equiv (\neg x_{31} \wedge \neg y_{31}) \\ \wedge \dots & \Rightarrow & \wedge \dots & \Rightarrow & \wedge \dots \\ \wedge (x_0 \vee y_0) = 0 & & \wedge \neg(x_0 \vee y_0) & & \wedge (\neg x_0 \wedge \neg y_0) \\ \text{not CNF} & & \text{not CNF} & & \text{CNF} \end{array}$$

$a = 0$ is $\neg a$ “ a is false”

De Morgan $\neg(a \vee b) \equiv \neg a \wedge \neg b$

CNF AND of ORs of literals; here 64 one-literal clauses

Example: Static Bug-Finding (7/7)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int z = x | y;  
    return 1 / z;  
}
```

Step 4: is F satisfiable?

Yes, a satisfying assignment:

$$x_{31} = \dots = x_0 = 0 \quad y_{31} = \dots = y_0 = 0$$

$\Rightarrow$ bug-inducing input: $x = 0, y = 0$

Satisfiability Modulo Theories (SMT) (1/3)

- A generalization of SAT
 - SAT only handles theory of Booleans

Satisfiability Modulo Theories (SMT) (2/3)

- A generalization of SAT

SAT only handles theory of Booleans

SMT extends it to handle **integers**, **reals**, **arrays**, **strings**, **bit-vectors**, etc.

Satisfiability Modulo Theories (SMT) (3/3)

- A generalization of SAT

SAT only handles theory of Booleans

SMT extends it to handle **integers**, **reals**, **arrays**, **strings**, **bit-vectors**, etc.

- Given a first-order logic formula, does there exist an assignment that makes the formula evaluate to **TRUE**?

SMT: Example Formula (1/2)

- Is there a satisfying assignment for the following formula?

$$\left((\neg p_1 \vee p_2) \vee f(x - y) \leq f(y) \right) \wedge \left(p_1 \vee y < 10.5 \vee e^y > \frac{y}{x} \right)$$

SMT: Example Formula (2/2)

- Is there a satisfying assignment for the following formula?

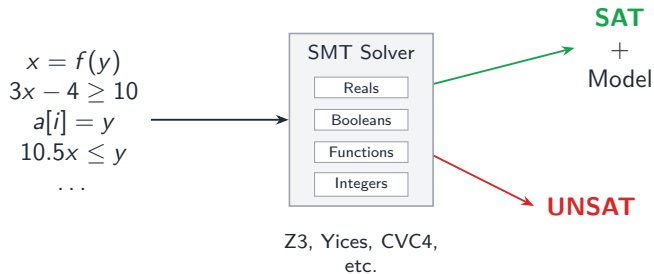
$$\left((\neg p_1 \vee p_2) \vee f(x - y) \leq f(y) \right) \wedge \left(p_1 \vee y < 10.5 \vee e^y > \frac{y}{x} \right)$$

Theory of Booleans

Theory of Uninterpreted Functions

Theory of Reals

SMT Solvers



Example: Static Verification (1/3)

```
int foo(int x, int y) {  
  int u = x + y;  
  int z;  
  if (u == 1)  
    z = 2;  
  else  
    z = u + 1;  
  u = y / z;  
  return u;  
}
```

Does this program divide by zero?

C_1	$u = x + y$	<code>int u = x + y;</code>
C_2	$(u = 1 \wedge z = 2)$	<code>true branch</code>
C_3	$(\neg(u = 1) \wedge (z = u + 1))$	<code>else branch</code>
C_4	$z = 0$	<code>u = y / z;</code>

Example: Static Verification (2/3)

Does this program divide by zero?

```
int foo(int x, int y) {  
    int u = x + y;  
    int z;  
    if (u == 1)  
        z = 2;  
    else  
        z = u + 1;  
    u = y / z;  
    return u;  
}
```

Program paths:

$$C_1 \rightarrow C_2 \rightarrow C_4$$
$$C_1 \rightarrow C_3 \rightarrow C_4$$

Is the conjunction of constraints along some program path satisfiable?

Example: Static Verification (3/3)

Does this program divide by zero?

```
int foo(int x, int y) {  
  int u = x + y;  
  int z;  
  if (u == 1)  
    z = 2;  
  else  
    z = u + 1;  
  u = y / z;  
  return u;  
}
```

$$C_1 \rightarrow C_2 \rightarrow C_4$$

$$\begin{aligned} &(u = x + y) \\ &\wedge (u = 1 \wedge z = 2) \quad \vee \\ &\wedge (z = 0) \end{aligned}$$

divide-by-zero bug!

$$C_1 \rightarrow C_3 \rightarrow C_4$$

$$\begin{aligned} &(u = x + y) \\ &\wedge (\neg(u = 1) \wedge (z = u + 1)) \\ &\wedge (z = 0) \end{aligned}$$

SAT $x = 0, y = -1$