

Constraint-Based Analysis

Mayur Naik



```
s.close()
for i in range(1, 1000):
    attack()

import os
print os.getpid()
print id = os.fork()
def attack():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #pid = os.fork()
    s.connect(("10.0.0.1", 80))
    print ">> GET /" + sys.argv[1]
    s.send("GET /" + sys.argv[1])
    s.send("Host: " + sys.argv[2])
    s.close()
for i in range(1, 1000):
```

The field of software analysis is highly diverse: there are many different approaches each with their own strengths and limitations in aspects such as soundness, completeness, scalability, and applicability.

We will learn about a dominant approach to software analysis called constraint-based analysis.

Constraint-based analysis follows a declarative paradigm: it is concerned with expressing “what” the analysis computes rather than “how” the analysis computes it.

In other words, constraint-based analysis is concerned with the specification of the analysis, rather than the implementation of the analysis.

The analysis specification takes the form of constraints over program facts, while the analysis implementation involves solving these constraints using an off-the-shelf constraint solver.

This separation of concerns has many benefits: it simplifies the design and understanding of the analysis, it allows to rapidly prototype analyses, and it enables to leverage continual performance improvements in constraint solvers.

We will illustrate these benefits using different constraint languages, including Datalog, SAT, and SMT.

LESSON

Introduction

SEGMENT

Motivation and Background

Motivation

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
- ...

Designing an efficient program analysis is a challenging task. It involves dealing with both the specification of the analysis -- that is, what information the analysis must compute -- and the implementation of the analysis -- that is, the details of how the analysis should compute that information efficiently.

For example, in a null-pointer dereference checking analysis, the specification might be, "No null pointer is dereferenced along any path in the program." As for the implementation, there are several design choices that affect the efficiency of the analysis, such as whether to use a forward vs. a backwards traversal of the program, whether to use symbolic vs. explicit representations of the program's state, and many others.

CIS 547 - Software Analysis

Motivation

Designing an efficient program analysis is challenging

Program Analysis = Specification + Implementation

Nontrivial!

Consider null pointer dereference analysis:

- No null pointer assignments
($v = \text{null}$): forward is best
- No pointer dereferences
($v \rightarrow \text{next}$): backward is best

"How"

Many design choices:

- forward vs. backward traversal
- symbolic vs. explicit representation
- ...

Penn Engineering Property of Penn Engineering | 5

Even the first choice, whether to traverse the program forward or backward, is a non-trivial decision.

Consider for instance a null-pointer dereference checking analysis.

A forward traversal involves starting at locations in the program where pointers are set to null and checking if they can flow to locations in the program where pointers are dereferenced. A backward traversal involves doing the opposite, that is, starting at locations in the program where pointers are dereferenced, and checking if locations where pointers are set to null can reach them.

It is easy to see that, if a program does not set any pointers to null, then forward traversal is more efficient. On the other hand, if the program does not dereference any pointers, then backward traversal is more efficient. In practice, programs contain a mix of both null pointer assignments and pointer dereferences, making it challenging to determine the most efficient traversal strategy.

CIS 547 - Software Analysis

What Is Constraint-Based Analysis?

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

Penn Engineering Property of Penn Engineering | 6

In constraint-based analysis, the analysis designer defines the specification of the program analysis using what is called a constraint language, and a constraint solver automates the implementation of the analysis.

CS 547 - Software Analysis

Benefits of Constraint-Based Analysis

- 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

Penn Engineering Property of Penn Engineering | 7

This approach to program analysis has several benefits.

Because the analysis specification is separated from the implementation, analysis designers can focus their efforts on specifying what information the analysis must compute, rather than implementing how the analysis should compute that information efficiently.

Another benefit of constraint-based analysis is that it yields natural program specifications: just like types in a type system, constraints are usually defined locally, and solving their conjunction captures global properties about the program.

Finally, the modularization of the program analysis task into a specification and an implementation sub-problem allows the specification to be agnostic of the implementation. In other words, we can leverage powerful, off-the-shelf constraint solvers in a “plug-and-play” manner, giving us flexibility that would otherwise not be available.

CS 547 - Software Analysis

Outline

- A constraint language: Datalog
- Two static analyses in Datalog:
 - Intra-procedural analysis: computing reaching definitions
 - Inter-procedural analysis: computing points-to information
- Other constraint languages: SAT and SMT

Penn Engineering Property of Penn Engineering | 8

Here are the topics we will consider in the remainder of this module.

Next, you will learn a language called Datalog that can be used to specify constraint-based analyses.

Once you have learned the basics of Datalog, you will see how to use it to specify two kinds of static analyses:

First, you will see how to specify an intra-procedural analysis in Datalog, that is, an analysis that is restricted to a single procedure. In particular, you will see how to specify computing reaching definitions.

Then, you will see how to define an inter-procedural analysis in Datalog, that is, an analysis of a program involving multiple procedures. In particular, you will see how to specify computing points-to information. You will also see the extra complexities inherent in defining these types of analysis.

Lastly, we will look at two other constraint languages, SAT and SMT, and how they can be used to check assertions.

CS 547 - Software Analysis

QUIZ: Specification & Implementation

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.

Penn Engineering Property of Penn Engineering | 9

{QUIZ SLIDE}

To illustrate the difference between the specification and the implementation of a program analysis, let's look at the following quiz. Consider a dataflow analysis such as live variables analysis. If this analysis is expressed as a constraint-based analysis, which of the following must the analysis designer still decide upon?

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 intersections or unions at merge points

Check all that apply.

CS 547 - Software Analysis

QUIZ: Specification & Implementation

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.

Penn Engineering Property of Penn Engineering | 10

{SOLUTION SLIDE}

Recall that, when using a constraint-based analysis, the user only needs to decide aspects of the specification, not the implementation. Therefore, the answers to this quiz are those that are part of the specification of live variables analysis instead of its implementation. Let's consider each statement in turn.

The order in which statements should be processed: this is an implementation aspect, as changing the order in which statements are processed would not change the outcome of the analysis. Therefore this is an aspect the constraint solver would determine, so the analysis designer does not need to decide this.

What the gen and kill sets for each kind of statement are: this is a specification aspect. Choosing different gen and kill sets would affect the outcome of the analysis. Therefore the analysis designer needs to decide on this aspect.

In what language to implement the chaotic iteration algorithm: this choice won't affect the final outcome of the analysis, so it's another implementation aspect that the analysis designer is not responsible for.

Whether to take the intersection or union at merge points: switching between intersection and union changes the type of analysis that is being done, so this is a specification detail that the analysis designer needs to decide.



LESSON

An Overview of Datalog



SEGMENT

Datalog Syntax and Semantics

A Constraint Language: Datalog

- A declarative logic programming language
- **Not Turing-complete**: 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: Souffle, bddbldb, IRIS, ...

Datalog is a declarative logic programming language.

It is not a Turing-complete language: it can be viewed as a subset of Prolog, or as SQL with recursion. Efficient algorithms exist to evaluate programs in these languages, so there exist efficient algorithms to evaluate Datalog programs.

Datalog originated as a query language for deductive databases. It was later applied in many other domains, including software analysis, data mining, networking, security, knowledge representation, and cloud computing among others.

There are many implementations of Datalog. Some of the implementations available include Souffle, bddbldb, and IRIS.

You can learn more about Datalog using the following resources:

<http://www.utdallas.edu/~gupta/courses/acl/papers/datalog-paper.pdf>

Online book: <http://webdam.inria.fr/Alice/>

Syntax 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).

We will now present the syntax of Datalog by means of an example program that computes reachability in a directed graph.

The problem of graph reachability is to determine all pairs of nodes in a graph that are connected by a path.

To express this problem as a program in Datalog, we need to define three things:

the form of the input to the Datalog program,
the form of the output of the Datalog program, and
the rules of inference comprising the Datalog program that compute the output from the input.

CS 547 - Software Analysis

Syntax of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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

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

Penn Engineering Property of Penn Engineering | 15

A Datalog program's inputs and outputs are defined in terms of relations, which are declarative statements that some number of objects are related in some way.

A relation is similar to a table in a relational database, and a tuple in a relation is similar to a row in the table: it asserts that the relation holds among some number of objects.

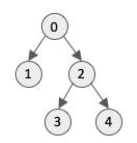
CS 547 - Software Analysis

Syntax of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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


→

edge	
n	m
0	1
0	2
2	3
2	4

Penn Engineering Property of Penn Engineering | 16

For the graph-reachability problem, the input is a single binary relation called $\text{edge}(n:N, m:N)$, where n and m are variables of type Node, denoted by N , the set of all nodes. This relation encodes the edges in the input graph. For example, for the graph shown here, the edge relation contains tuples $(0,1)$ and $(2,3)$, but not tuples $(3,4)$, $(0,3)$, or $(2,0)$.

The four tuples $(0,1)$, $(0,2)$, $(2,3)$, and $(2,4)$ are sufficient to establish the entire structure of the graph.

The output of this Datalog program is a single binary relation called $\text{path}(n:N, m:N)$, which is true iff there is a directed path in the graph from n to m . So, for the graph shown, the path relation should contain tuples $(0,4)$ and $(0,3)$, but not tuples $(3,0)$ or $(1,4)$.

CS 547 - Software Analysis

Syntax 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).`

Deductive rules that hold universally
 (i.e., variables like x, y, z can be
 replaced by any constant).
 Specify "if ... then ..." logic.

Penn Engineering Property of Penn Engineering | 17

In order for the Datalog program to compute the output relations from the input relations, we must provide rules of inference. These are deductive rules that hold universally. They specify logical "if-then" statements.

CS 547 - Software Analysis

Syntax 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).`

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

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

Penn Engineering Property of Penn Engineering | 18

The rules of inference that we will define for this problem are (in English):

First: There is always a path from each node x to itself, which in Datalog syntax takes the form

`path(x, x).`

Second: If there is a path from node x to node y and an edge from node y to node z, then there is a path from node x to node z. In Datalog syntax, this rule takes the form

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

The rules of inference are written in the opposite order that they are typically written in: the hypothesis of an implication is written on the right-hand side, and the conclusion is written on the left-hand side. Relations separated by a comma are ANDed together. The first inference rule, because it has no hypotheses, acts as an axiomatic statement. Finally, a period is used to end each inference rule.

CS 547 - Software Analysis

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

```

Penn Engineering Property of Penn Engineering | 19

Now that you're familiar with the syntax of Datalog programs, I will illustrate the semantics of Datalog programs, using the graph-reachability example. Conceptually, we start out with the empty path relation, and apply each of these two rules, growing the path relation with each application. We stop when the path relation stops growing.

A slight variant of this algorithm is depicted here [point to box]. It starts out by applying the first rule, which involves adding to the path relation each tuple (x, x) for each node x in the graph, capturing the intent of this rule that there exists a path from each node to itself. It then repeatedly applies the second rule, which involves adding to the path relation each tuple (x, z) whenever there exists a node y such that tuple (x, y) exists in the current path relation and tuple (y, z) exists in the input edge relation. This captures the intent of the second rule, that there exists a path from node x to node z if there exists a path from node x to some node y, and there exists an edge from that node y to node z.

This naive algorithm is essentially the chaotic iteration algorithm used for dataflow analyses and pointer analysis. In practice, Datalog solvers have much more efficient algorithms for computing the output relations from the input relations and inference rules. The key is that if there are multiple rules, the order in which the rules are applied does not matter.

Additionally, the result of the algorithm, like that of chaotic iteration, is the least solution: the smallest path relation that satisfies all the rules. The least solution

typically corresponds to what the user wants to compute in many problems. An example of a non-least solution to this problem would be that path(x,y) holds for all nodes x and y. While this relation doesn't violate any rules, it contains many nonsensical paths that would not be desired by a user.

CS 547 - Software Analysis

Semantics of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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

```

graph TD
    0((0)) --> 1((1))
    0((0)) --> 2((2))
    1((1)) --> 3((3))
    2((2)) --> 4((4))
  
```

Input Tuples:
 $\text{edge}(0, 1), \text{edge}(0, 2), \text{edge}(2, 3), \text{edge}(2, 4)$

Output Tuples:
 $\text{path}(0, 0), \text{path}(1, 1), \text{path}(2, 2), \text{path}(3, 3), \text{path}(4, 4), \text{path}(0, 1), \text{path}(0, 2), \text{path}(2, 3), \text{path}(2, 4), \text{path}(0, 3), \text{path}(0, 4)$

Penn Engineering Property of Penn Engineering | 20

Let's look at a run of this Datalog program on an example input.

Suppose the input is the following directed graph, encoded by the following edge relation, which contains four tuples: (0,1), (0,2), (2,3) and (2,4).

The output of this Datalog program on this input is as follows:

CS 547 - Software Analysis

Semantics of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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

```

graph TD
    0((0)) --> 1((1))
    0((0)) --> 2((2))
    1((1)) --> 3((3))
    2((2)) --> 4((4))
  
```

Input Tuples:
 $\text{edge}(0, 1), \text{edge}(0, 2), \text{edge}(2, 3), \text{edge}(2, 4)$

Output Tuples:
 $\text{path}(0, 0), \text{path}(1, 1), \text{path}(2, 2), \text{path}(3, 3), \text{path}(4, 4), \text{path}(0, 1), \text{path}(0, 2), \text{path}(2, 3), \text{path}(2, 4), \text{path}(0, 3), \text{path}(0, 4)$

Penn Engineering Property of Penn Engineering | 21

Applying the first rule, $\text{path}(x,x)$, produces all paths of length 0, represented by the following tuples in the path relation: (0,0), (1,1), (2,2), (3,3), and (4,4).

CIS 547 - Software Analysis

Semantics of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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

```

graph TD
    0((0)) --> 1((1))
    0((0)) --> 2((2))
    2((2)) --> 3((3))
    2((2)) --> 4((4))
  
```

Input Tuples:
 $\text{edge}(0, 1), \text{edge}(0, 2), \text{edge}(2, 3), \text{edge}(2, 4)$

Output Tuples:
 $\text{path}(0, 0), \text{path}(1, 1), \text{path}(2, 2), \text{path}(3, 3), \text{path}(4, 4), \text{path}(0, 1), \text{path}(0, 2), \text{path}(2, 3), \text{path}(2, 4), \text{path}(0, 3), \text{path}(0, 4)$

Penn Engineering Property of Penn Engineering | 22

Applying the second rule at this time yields all paths of length 1, represented by the following tuples in the path relation: (0,1), (0,2), (2,3), and (2,4).

CIS 547 - Software Analysis

Semantics of Datalog: Example

Input Relations:
 $\text{edge}(n:N, m:N)$

Output Relations:
 $\text{path}(n:N, m:N)$

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

```

graph TD
    0((0)) --> 1((1))
    0((0)) --> 2((2))
    2((2)) --> 3((3))
    2((2)) --> 4((4))
  
```

Input Tuples:
 $\text{edge}(0, 1), \text{edge}(0, 2), \text{edge}(2, 3), \text{edge}(2, 4)$

Output Tuples:
 $\text{path}(0, 0), \text{path}(1, 1), \text{path}(2, 2), \text{path}(3, 3), \text{path}(4, 4), \text{path}(0, 1), \text{path}(0, 2), \text{path}(2, 3), \text{path}(2, 4), \text{path}(0, 3), \text{path}(0, 4)$

Penn Engineering Property of Penn Engineering | 23

Applying the second rule again yields all paths of length two: (0,3) and (0,4).

Because the path relation doesn't change after applying either of these rules again, the algorithm terminates, yielding the least solution seen here.

CIS 547 - Software Analysis

QUIZ: Computation Using Datalog

Check each of the below Datalog programs that computes in relation **scc** exactly those pairs of nodes $(n1, n2)$ such that $n1$ is reachable from $n2$, and $n2$ is reachable from $n1$.

- `scc(n1, n2) :- edge(n1, n2), edge(n2, n1).`
- `scc(n1, n2) :- path(n1, n2), path(n2, n1).`
- `scc(n1, n2) :- path(n1, n3), path(n3, n2), path(n2, n4), path(n4, n1).`
- `scc(n1, n2) :- path(n1, n3), path(n2, n3).`

Penn Engineering Property of Penn Engineering | 24

{QUIZ SLIDE}

Let's work on expressing another computation in Datalog in the form of a quiz. Suppose we want to compute the relation **scc** (standing for strongly connected component) on a directed graph from the input relations **edge** and **path** (as we defined them earlier), and suppose we want our Datalog program to output **scc**($n1, n2$) if and only if $n2$ is reachable from $n1$ and $n1$ is reachable from $n2$.

Select each of the inference rules below that will compute the correct output:

`scc(n1, n2) :- edge(n1, n2), edge(n2, n1).`
`scc(n1, n2) :- path(n1, n2), path(n2, n1).`
`scc(n1, n2) :- path(n1, n3), path(n3, n2), path(n2, n4), path(n4, n1).`
`scc(n1, n2) :- path(n1, n3), path(n2, n3).`

CIS 547 - Software Analysis

QUIZ: Computation Using Datalog

Check each of the below Datalog programs that computes in relation **scc** exactly those pairs of nodes $(n1, n2)$ such that $n1$ is reachable from $n2$, and $n2$ is reachable from $n1$.

- `scc(n1, n2) :- edge(n1, n2), edge(n2, n1).`
- `scc(n1, n2) :- path(n1, n2), path(n2, n1).`
- `scc(n1, n2) :- path(n1, n3), path(n3, n2), path(n2, n4), path(n4, n1).`
- `scc(n1, n2) :- path(n1, n3), path(n2, n3).`

Penn Engineering Property of Penn Engineering | 25

{SOLUTION SLIDE}

Two of these inference rules---the second and third---correctly compute the relation **scc**. The second rule is minimal in its expression of the **scc** relation, but the third rule still computes the same relation. To see this, recall that **path**(x,x) holds for all nodes x ; therefore, by taking $n3$ equal to $n1$ and $n4$ equal to $n2$, the hypothesis is true if and only if **path**($n1,n2$) and **path**($n2,n1$) are true.

While the first rule will not produce any incorrect tuples in relation **scc**, it will fail to produce **scc**($n1,n2$) for any two nodes that are reachable from each other but which are not adjacent to each other.

Finally, the last rule could potentially produce incorrect tuples: the fact that there exists some node $n3$ such that there is a path from $n1$ to $n3$ and a path from $n2$ to $n3$ is neither a necessary nor a sufficient condition for nodes $n1$ and $n2$ to belong to a strongly connected component.

LESSON

Analysis Using Datalog

SEGMENT

Reaching Definitions Analysis

CS 547 - Software Analysis

Dataflow Analysis in Datalog

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

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

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

Penn Engineering Property of Penn Engineering | 28

Now that we have seen the syntax and semantics of Datalog programs, we will consider how to use Datalog to specify an intra-procedural dataflow analysis; specifically, reaching definitions analysis.

The specification of reaching definitions analysis is as follows:

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

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

where $\text{KILL}[n]$ is the set of definitions killed at program point n , $\text{GEN}[n]$ is the set of definitions generated at program point n , and $\text{predecessors}(n)$ is the set of program points that immediately precede program point n in the input procedure's control-flow graph.

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

Input Relations:

Output Relations:

Rules:

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

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

Penn Engineering Property of Penn Engineering | 29

Let us describe the form of the input and output relations as well as the inference rules that would be used to specify reaching definitions analysis in Datalog.

The input relations for the analysis should capture all the information from the input procedure's control-flow graph that is relevant to computing the IN and OUT sets for each program point. While we haven't yet formally defined the inference rules for the analysis, by looking at the specification we see that in order to compute $\text{OUT}[n]$, we need to know $\text{KILL}[n]$ and $\text{GEN}[n]$, and in order to compute $\text{IN}[n]$, we need to know $\text{predecessors}(n)$. Therefore, the input relations should give Datalog's constraint solver knowledge of the contents of the KILL, GEN, and predecessors sets. Moreover, all three of these relations can be computed from the control-flow graph of the procedure to be analyzed.

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

Input Relations:
 $\text{kill}(n:N, d:D)$

Definition d is killed by statement n .

Output Relations:

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

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

Rules:

Penn Engineering

Property of Penn Engineering | 30

Let us define the relation $\text{kill}(n:N, d:D)$ to mean that the definition d is in the KILL set of program point n ...

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

Input Relations:
 $\text{kill}(n:N, d:D)$
 $\text{gen}(n:N, d:D)$

Definition d is generated by statement n .

Output Relations:

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

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

Rules:

Penn Engineering

Property of Penn Engineering | 31

the relation $\text{gen}(n:N, d:D)$ to mean that the definition d is in the GEN set of program point n ...

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

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

Output Relations:

Rules:

Statement **m** is an immediate successor of statement **n**.

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

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

Penn Engineering Property of Penn Engineering | 32

and the relation next(n:N, m:N) to mean that program point m is an immediate successor of program point n, or equivalently, that program point n is an immediate predecessor of program point m.

(In these relations, N denotes the set of all program points and D denotes the set of all definitions in the given control-flow graph.)

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

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:

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

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

Definition **d** may reach the program point **just before** statement **n**.

Penn Engineering Property of Penn Engineering | 33

In reaching definitions analysis, we want to compute the IN and OUT sets for each program point. So let us say that in(n:P, d:D) is the relation that asserts that the definition d is a member of the IN set of program point n---that is, definition d may reach the program point just before n ...

CIS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

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:

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

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

Definition d may reach the program point just after statement n.

Penn Engineering Property of Penn Engineering | 34

... and let us define out(n:P, d:D) to mean that definition d is a member of the OUT set of program point n---that is, definition d may reach the program point just after n.

CIS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

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).

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

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

Penn Engineering Property of Penn Engineering | 35

Lastly, we specify three rules of inference to compute the IN and OUT sets. These will be based on the formulas for OUT[n] and IN[n] shown here.

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

The first two rules map to the first rule in the specification, which is $OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$. We use two separate rules to reflect the union of $(IN[n] - KILL[n])$ with $GEN[n]$, and we use the '!' character to mean the relation kill(n,d) does not hold. This represents the fact that $IN[n] - KILL[n]$ is the intersection of $IN[n]$ with the complement of $KILL[n]$.

CS 547 - Software Analysis

Reaching Definitions Analysis in Datalog

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).

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

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

Penn Engineering Property of Penn Engineering | 36

Finally, the third rule maps to the second rule in the specification, which is $IN[n] = \bigcup_{n' \in \text{predecessors}[n]} OUT[n']$. Because this expression is a union, we only need a single inference rule to ensure all definitions are correctly added to the appropriate IN set. For each predecessor n of program point m, each definition d in the OUT set of that predecessor n will satisfy the hypothesis of this third inference rule.

CS 547 - Software Analysis

Reaching Definitions Analysis: Example

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).

```

graph TD
    1[1: entry] --> 2[2: x = 8]
    2 --> 3[3: (x != 1)?]
    3 -- true --> 4[4: x = x - 1]
    3 -- false --> 5[5: exit]
    4 --> 3
    
```

Penn Engineering Property of Penn Engineering | 37

Let's look at an example run of our reaching definitions analysis specified in Datalog. Consider this control-flow graph:

Program point 1 is the entry point to the procedure. It has a single transition to program point 2, which contains the statement $x = 8$. Program point 2 has a single transition to program point 3, which is a test of the boolean expression $(x \neq 1)$. If this expression is true, control flows to program point 4, which contains the definition $x = x - 1$ and then transitions to program point 3 again. If the boolean expression at program point 3 is false, control flows to program point 5, which is the exit point of the procedure.

CS 547 - Software Analysis

Reaching Definitions Analysis: Example

```

graph TD
    1[1: entry] --> 2[2: x = 8]
    2 --> 3[3: (x != 1)?]
    3 -- true --> 4[4: x = x - 1]
    3 -- false --> 5[5: exit]
    4 --> 3
  
```

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

Input Tuples:
 $kill(4, 2),$
 $gen(2, 2), gen(4, 4),$
 $next(1, 2), next(2, 3),$
 $next(3, 4), next(3, 5),$
 $next(4, 3)$

Penn Engineering Property of Penn Engineering | 38

Recall that the inputs to reaching definitions analysis are the relations $kill(n, d)$, $gen(n, d)$, and $next(n, m)$. Because each definition is associated with a program point, we will label the definitions by the number of the program point they appear at (so the definition $x = 8$ will be denoted by letting d equal 2 in the relations, and the definition $x = x - 1$ will be denoted by letting d equal 4). Let's look at the contents of each of these relations in turn for this control-flow graph, starting with the gen relation.

There are only two tuples in the gen relation for this control-flow graph: $(2, 2)$ and $(4, 4)$, as no other program points establish any variable definitions.

The $next$ relation can be computed from the directed edges in the control-flow graph. Each edge corresponds to a tuple in this relation. This relation thus contains the following tuples: $(1, 2)$, $(2, 3)$, $(3, 4)$, $(3, 5)$, and $(4, 3)$.

The $kill$ relation is a bit more difficult to compute. A definition d is in the KILL set of a program point n if there is a definition other than d generated at n and there is a directed path from the point where d is generated to n . Using the graph-reachability analysis we described earlier (for example), we can compute that the only such tuple that is applicable for this procedure is $(4, 2)$, capturing the fact that the definition associated with program point 2 is killed at program point 4.

CS 547 - Software Analysis

Reaching Definitions Analysis: Example

```

graph TD
    1[1: entry] --> 2[2: x = 8]
    2 --> 3[3: (x != 1)?]
    3 -- true --> 4[4: x = x - 1]
    3 -- false --> 5[5: exit]
    4 --> 3
  
```

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

Input Tuples:
 $kill(4, 2),$
 $gen(2, 2), gen(4, 4),$
 $next(1, 2), next(2, 3),$
 $next(3, 4), next(3, 5),$
 $next(4, 3)$

Output Tuples:
 $in(3, 2), in(3, 4), in(4, 2),$
 $in(4, 4), in(5, 2), in(5, 4),$
 $out(2, 2), out(3, 2), out(3, 4),$
 $out(4, 4), out(5, 2), out(5, 4)$

Penn Engineering Property of Penn Engineering | 39

Given these input relations and the inference rules described previously, the output relations produced are as follows.

The in relation contains the following tuples: $(3, 2)$, $(3, 4)$, $(4, 2)$, $(4, 4)$, $(5, 2)$, and $(5, 4)$.

The out relation contains the following tuples: $(2, 2)$, $(3, 2)$, $(3, 4)$, $(4, 4)$, $(5, 2)$, and $(5, 4)$.

You can verify the contents of these relations for yourself by using, for example, the chaotic iteration algorithm.

CS 547 - Software Analysis

QUIZ: Live Variables Analysis

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:

:- .

:- , ! .

:- , .

Penn Engineering Property of Penn Engineering | 40

{QUIZ SLIDE}

To practice specifying a program analysis in Datalog yourself, in the following quiz define the inference rules needed to compute a live variables analysis.

CS 547 - Software Analysis

QUIZ: Live Variables Analysis

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) .

Penn Engineering Property of Penn Engineering | 41

{SOLUTION SLIDE}

The three rules used for live variables analysis are very similar to those for reaching definitions analysis. In fact, the only change needed is that the IN and OUT sets have swapped places. The three rules needed are:

```

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

```

The order of the two hypotheses in your third rule may have varied but it should be equivalent to this rule.

SEGMENT

Intra-Procedural Pointer Analysis

Pointer Analysis in Datalog

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

```
(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
```

Let's now consider a flow-insensitive may-alias analysis on programs in the following language.

A program in this language consists of functions that have a single argument variable and whose body is a set of simple statements s_1 , through s_n . For convenience of presentation, we presume that the flow-insensitivity approximation has already been applied to the body of the function, enabling us to view it as a set of statements rather than a control-flow graph.

Each statement is either an object allocation statement, a copy assignment, a return statement, or a call to a function f with actual argument u whose return result is assigned to v .

Since we are dealing with a pointer analysis, we are only concerned with pointer typed variables, and we presume that each object allocation site is associated with a unique label.

For simplicity, we do not allow field-reads or field-writes in this language.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Intra-Procedural

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

```

(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
  
```


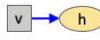
Penn Engineering Property of Penn Engineering | 44

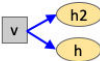
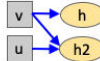
Let's first look at the intra-procedural aspects of the pointer analysis for our language. For this part of the analysis, we can ignore the function call and return statements, though we'll come back to them when we consider the inter-procedural aspects of the analysis.

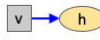
CS 547 - Software Analysis

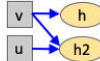
Pointer Analysis in Datalog: Intra-Procedural

Recall the specification:

Before:  

After:  

Before: 

After: 

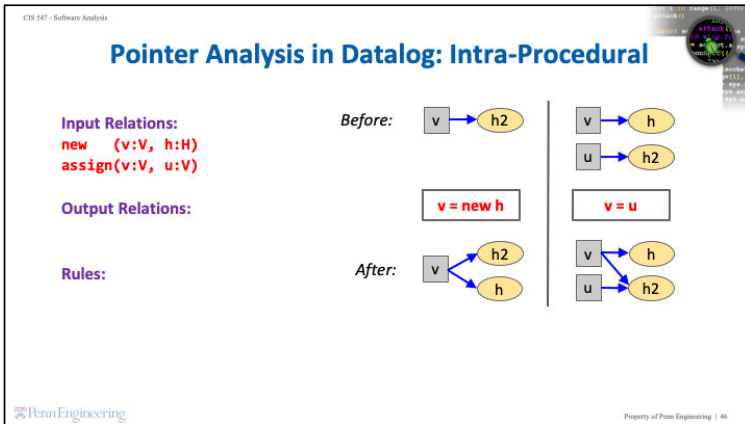
Penn Engineering Property of Penn Engineering | 45

Let's recall the specification of the pointer analysis rules associated with the object allocation and copy assignment statements as depicted in this diagram.

Before analyzing an object allocation statement `v = new h`, if the variable `v` points to an allocation site labeled `h2`, then after analyzing this statement, `v` may point to both allocation sites `h` and `h2`.

Likewise, before analyzing a copy assignment statement `v = u`, if the variable `v` points to an allocation site labeled `h` and variable `u` points to an allocation site labeled `h2`, then after analyzing this statement, `v` may point to both `h` and `h2`.

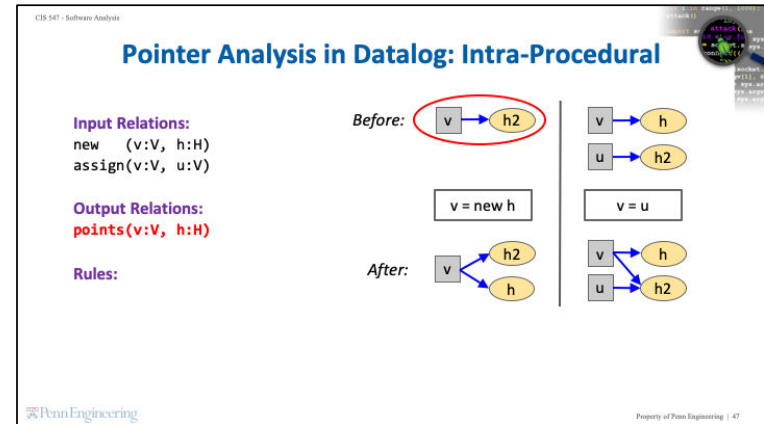
Note that in both cases, we accumulate points-to facts of variables rather than overwriting them, as this particular pointer analysis performs weak updates rather than strong updates, given its flow-insensitive nature.



The input relations to the pointer analysis correspond to object allocation and copy assignment statements.

The first one is $\text{new}(v:V, h:H)$, meaning that the object allocation statement $v = \text{new } h$ appears in the program being analyzed, and the second is $\text{assign}(v:V, u:V)$, meaning that the copy assignment statement $v = u$ appears in the program.

(The letter V denotes the set of all pointer-typed variables and the letter H denotes the set of labels of all object allocation statements.)



The output relation generated by the pointer analysis is of the form $\text{points}(v:V, h:H)$, meaning that the variable v may point to an object allocated at the site labeled h.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Intra-Procudural

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:

Penn Engineering | 48

Finally, the inference rules used to compute the points-to information are defined to reflect the diagrams depicting the specification of the pointer analysis.

The rule for object allocation statements is:

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

CS 547 - Software Analysis

Pointer Analysis in Datalog: Intra-Procudural

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:

After:

Penn Engineering | 49

and the rule for copy assignment statements is:

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

SEGMENT

Inter-Procedural Pointer Analysis

Pointer Analysis in Datalog: **Inter-Procedural**

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

```
(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
```

Now let us remove the restriction on function call and return statements, which will allow us to conduct inter-procedural pointer analysis.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

```

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

```

Penn Engineering Property of Penn Engineering | 52

Suppose we have the following program to analyze. The program has two statements: an object allocation statement $x = \text{new } h1$ and a function call statement $y = f(x)$. Function f takes a single argument v and has two statements in its body: $u = v$ and $\text{return } u$.

To analyze the program, we need to introduce rules for handling function calls and returns.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

Input Relations:
 $\text{new}(v:V, h:H)$
 $\text{assign}(v:V, u:V)$

Output Relations:
 $\text{points}(v:V, h:H)$

Rules:
 $\text{points}(v, h) \text{ :- new}(v, h).$
 $\text{points}(v, h) \text{ :- assign}(v, u), \text{points}(u, h).$

Parameter passing and return can be treated as assignments!

```

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

```

Penn Engineering Property of Penn Engineering | 53

So far, we know how to address the object allocation and copy assignment statements via the input relations $\text{new}(v, h)$ and $\text{assign}(v, u)$.

But how do we handle function calls and returns?

The trick is to treat parameter passing and return statements as copy assignments.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

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;`
`}`

Property of Penn Engineering | 54

More concretely, the line `y = f(x)` can be effectively replaced by three lines:

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

where the first line assigns the value of the passed argument `x` to the variable `v`, the second line is the body of the function `f`, which in this case is just the statement `u = v`, and the third line assigns the value of variable `u` that would be returned to variable `y` receiving the output of the function call `f(x)`.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

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;`
`}`

Property of Penn Engineering | 55

To implement the inter-procedural version of the pointer analysis, we need to add input relations capturing function calls, function definitions, and return statements:

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

where the letter `F` denotes the set of all functions in the program being analyzed.

`arg(f:F, v:V)` means that the function `f` is defined with `v` as its argument variable.

`ret(f:F, u:V)` means that the function `f` returns the value of the variable `u`.

And `call(y:V, f:F, x:V)` means that the function `f` is called with argument variable `x` and that its output is assigned to the variable `y`.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

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).`

```

call(y,f,x)
  x = new h1;
  y = f(x);
  f(v) {
    u = v;
    return u;
  }
arg(f,v)
ret(f,u)
  
```

Penn Engineering Property of Penn Engineering | 56

So, for our example program, we would include the following tuples: tuple (f,v) in the arg relation, tuple (f,u) in the ret relation, and tuple (y,f,x) in the call relation.

CS 547 - Software Analysis

Pointer Analysis in Datalog: Inter-Procedural

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"
  
```

Penn Engineering Property of Penn Engineering | 57

Finally, we add new rules of inference to compute points-to information from these relations. The first rule is

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

to reflect the fact that the variable v in the definition of f may point to the same allocation site as the variable x passed into f at the call.

The underscore character refers to a "wildcard": it doesn't matter what variable is present in that slot, as the output is unaffected by it.

Pointer Analysis in Datalog: Inter-Procedural

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;
}
```

And the second new inference rule is:

```
points(y, h) :- call(y, f, _), ret(f, u), points(u, h).
```

to reflect that if variable y receives the output of a call to f , and f returns the variable u , then y may point to the same allocation site as u .

QUIZ: Querying Pointer Analysis

Check each of the below Datalog programs that computes in relation **mustNotAlias** each pair of variables (u, v) such that u and v do not alias in any run of the program.

- `mustNotAlias(u, v) :- points(u, h1), points(v, h2), h1 != h2.`
- `mayAlias(u, v) :- points(u, h), points(v, h).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`
- `mayAlias(u, v) :- points(u, _), points(v, _).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`
- `common(u, v, h) :- points(u, h), points(v, h).`
`mayAlias(u, v) :- common(u, v, _).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`

{QUIZ SLIDE}

Suppose you want to compute the relation `mustNotAlias` for a program in our toy language, where `mustNotAlias(u,v)` holds if and only if u and v do not alias in any run of the program.

Select each of the Datalog programs below that will compute the correct output.

QUIZ: Querying Pointer Analysis

Check each of the below Datalog programs that computes in relation `mustNotAlias` each pair of variables `(u, v)` such that `u` and `v` do not alias in any run of the program.

- `mustNotAlias(u, v) :- points(u, h1), points(v, h2), h1 != h2.`
- `mayAlias(u, v) :- points(u, h), points(v, h).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`
- `mayAlias(u, v) :- points(u, _), points(v, _).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`
- `common(u, v, h) :- points(u, h), points(v, h).`
`mayAlias(u, v) :- common(u, v, _).`
`mustNotAlias(u, v) :- !mayAlias(u, v).`

Penn Engineering Property of Penn Engineering | 40

{SOLUTION SLIDE}

The two programs which correctly compute the `mustNotAlias` relation are the second and fourth.

The first program does not correctly compute the relation. A variable may point to more than one allocation site at once, so it is not sufficient merely to check that there exist distinct allocation sites that are pointed to by `u` and `v`.

The second program does correctly compute the `mustNotAlias` relation. `mayAlias(u,v)` holds whenever `u` and `v` may point to the same allocation site, and `mustNotAlias` is the logical negation of `mayAlias`.

The third program does not correctly compute the relation. Because of the wildcard character, it would generate the tuple `mayAlias(u,v)` if there is any points tuple with `u` and any points tuple with `v`, even if the allocation sites in those tuples are not the same.

Finally, the fourth program does correctly compute `mustNotAlias`. The first two rules are logically equivalent to the rule `mayAlias(u,v) :- points(u,h), points(v,h)`, so this program computes the same result as the second program.

SEGMENT

Context-Sensitive Analysis

Penn Engineering Property of Penn Engineering | 41

CS 547 - Software Analysis

Context Sensitivity

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).`

```

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

```

Property of Penn Engineering | 62

The rules we've defined so far correspond to what is called a context-insensitive pointer analysis -- that is, an analysis that conflates points-to information across different calls to the same function. This in turn results in a loss of precision. To achieve a more precise analysis, we introduce context sensitivity.

For example, consider this new program. The program has four statements: two object allocation statements `x = new h1` and `z = new h2`, and two function call statements `y = f(x)` and `w = f(z)`. Function `f` is as before: it takes a single argument `v` and has two statements in its body: `u = v` and `return u`.

CS 547 - Software Analysis

Context Sensitivity

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

```

```

v = z
u = v
w = u

```

```

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

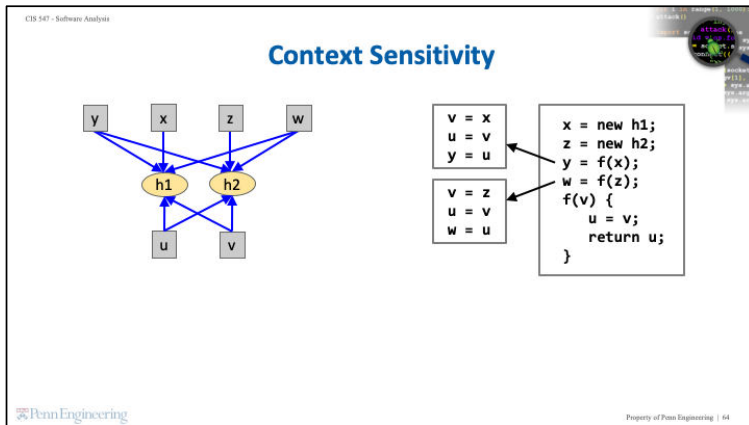
```

Property of Penn Engineering | 63

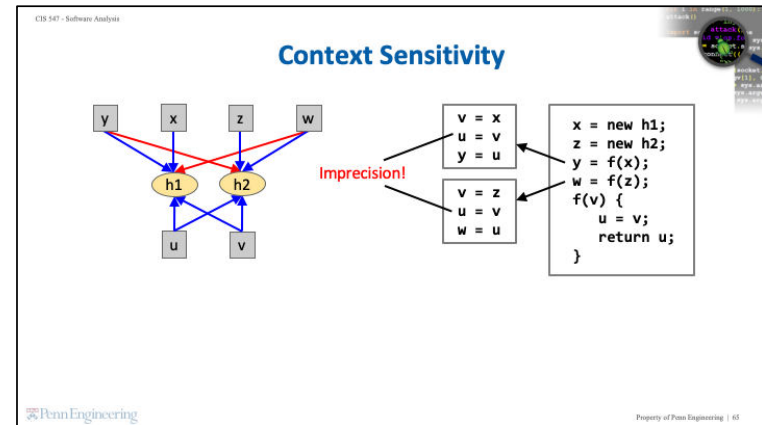
As before, we could try replacing the function calls with assignment statements:

`y = f(x);` would be replaced by `v = x; u = v; y = u;` and

`w = f(z);` would be replaced by `v = z; u = v; w = u;`



Let's build the points-to graph that would correspond to the pointer analysis as we've defined it so far.



Notice that w may point to h1 and y may point to h2 in this points-to graph. This introduces imprecision into the pointer analysis we've defined so far: w can never point to the object allocated at h1, and y can never point to the object allocated at h2.

Cloning-Based Inter-Procedural Analysis

Achieves context sensitivity by **inlining** procedure calls

Cloning depth ↑ : precision ↑ vs. scalability ↓

One way to add context sensitivity to the analysis is through what is called "cloning". It achieves context sensitivity by reproducing the bodies of the procedure in-line with distinguished variable names.

For example, in this program, instead of replacing $y = f(x)$; by $v = x$; $u = v$; $y = u$; and $w = f(z)$; by $v = z$; $u = v$; $w = u$;, we could introduce different copies of the variables v and u (say, v_i and u_i versus v_j and u_j) for each call to f . In this way, we avoid imprecisely claiming that w may point to h_1 or that y may point to h_2 . Instead, we would have an equivalent program for which pointer analysis would generate a precise points-to graph.

We can achieve greater precision by allowing cloning to be used for more levels in the call stack. However, the tradeoff for precision via cloning is scalability. The deeper we allow function calls to be cloned, the more space and time we need to allow for the resulting analysis. If each function calls just two other functions, the resources needed for a precise analysis becomes exponential in the depth of the stack of nested function calls.

What about Recursion?

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

In fact, if there are recursive function calls, as in the following program, which is similar to the previous one except that the body of function f recursively calls f , then an infinite cloning depth is needed to differentiate the points-to set of x and y from that of w and z .

You can learn more about cloning-based context sensitivity by following the link: <http://suif.stanford.edu/papers/pldi04.pdf>

CIS 547 - Software Analysis

Summary-Based Inter-Procedural Analysis

- 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!

Penn Engineering Property of Penn Engineering | 68

Cloning is not the only way to perform context-sensitive analysis. Another popular approach, called the summary-based approach, uses the incoming program states to differentiate between different calls to the same procedure.

Since in general, there may be infinitely many different such concrete states, the analysis designer must apply a suitable abstraction that conflates them enough to enable the analysis to be scalable and terminate, yet make enough distinctions to enable the analysis to be precise.

Then, the same incoming program states yield the same outgoing program states for a given procedure, and these input-output pairs of program states are called summaries.

Summary-based analysis is as precise as cloning-based analysis with infinite cloning depth.

You can read more about summary-based inter-procedural analysis by following the link: <https://research.cs.wisc.edu/wpis/papers/pop195.pdf>

CIS 547 - Software Analysis

LESSON

Analysis Using SAT and SMT

Penn Engineering Property of Penn Engineering | 69

CS 547 - Software Analysis

SEGMENT

Constraint Languages

Property of Penn Engineering | 70

CS 547 - Software Analysis

Constraint Languages

Constraint Language	Problem Expressed	Example Solvers
Datalog	Least solution of deductive inference rules	Souffle, bddbdb
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

Property of Penn Engineering | 71

Thus far in this module, we focused on a particular constraint language, Datalog. However, there are several other constraint languages with different expressiveness and performance characteristics. Here are some of the popular languages along with the kind of problem that one can express using each of them, and some example solvers for each of those problems.

In Datalog, recall that one must specify the analysis in terms of the problem of finding the least solution of deductive inference rules, and example solvers for this problem are bddbdb and Souffle.

SAT is the well-known Boolean satisfiability problem. Using SAT, one must specify the analysis in terms of the problem of determining whether a set of Boolean constraints is satisfiable. Example solvers for this problem are MiniSat and Glucose.

MaxSAT is an optimization extension of the Boolean satisfiability problem. One can specify not only what Boolean constraints must be satisfied but also an objective function to minimize or maximize. This optimization aspect of MaxSAT can be used to express, for instance, various tradeoffs in the analysis. Example MaxSAT solvers are open-wbo and SAT4j.

SMT, the Satisfiability Modulo Theories problem, is an extension of the SAT problem. It allows one to specify not just constraints over booleans, but also constraints over integers and pointers. Example SMT solvers include Z3 from

Microsoft and Yices.

Finally, MaxSMT is an optimization extension of the SMT problem, similar to how MaxSAT is to the SAT problem. An example MaxSMT solver is Z3.

In this lesson, we will give an overview of using SAT and SMT for program analysis in general, and assertion checking in particular.

The slide is titled "SEGMENT" in blue text at the top center. Below the title, the main heading "Analysis Using SAT" is displayed in black text. In the top right corner, there is a small circular graphic with a globe and some text. In the bottom left corner, the "Penn Engineering" logo is visible. In the bottom right corner, the text "Property of Penn Engineering | 72" is present. The slide is framed by a black border.

CS 547 - Software Analysis

Boolean Satisfiability Problem (SAT)

- 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**

$(A \vee B \vee C) \wedge (\neg B \vee \neg C \vee \neg E) \wedge (B \vee D)$ ——— Conjunctive Normal Form (CNF)
 an AND of ORs

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

Penn Engineering Property of Penn Engineering | 73

The Boolean satisfiability problem, or SAT for short, is a decision problem that asks if a given formula F over boolean variables x_1 through x_n is satisfiable. F is satisfiable if and only if there exists an assignment of boolean values to x_1 through x_n such that F evaluates to true.

As an example, suppose F is the following formula over boolean variables A, B, C, D , and E . The symbol \vee denotes the binary OR operator or disjunction, the symbol inverted \vee denotes the binary AND operator or conjunction, and this symbol denotes the unary negation operator.

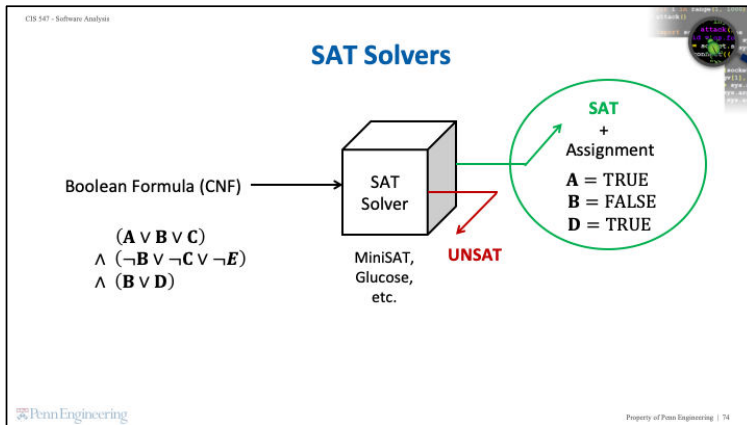
Any boolean formula can be expressed in a specialized form called Conjunctive Normal Form, or CNF for short. This example formula is in CNF. In this normal form, the formula is a conjunction of clauses, where each clause is a disjunction of literals, where a literal is either an atom such as A or its negation such as not B .

This example formula is satisfiable because it evaluates to true under the assignment where A is true, B is false, and D is true. To show that a CNF formula evaluates to true under an assignment, we must show that each clause of the formula evaluates to true under the assignment. A clause evaluates to true if any literal of the clause evaluates to true. Let's look at each clause of our example formula in turn.

The first clause evaluates to true since A is true in the given assignment.
 The second clause evaluates to true since B is false in the given assignment, which

makes the literal “not B ” evaluate to true.
 The third clause evaluates to true since D is true in the given assignment.

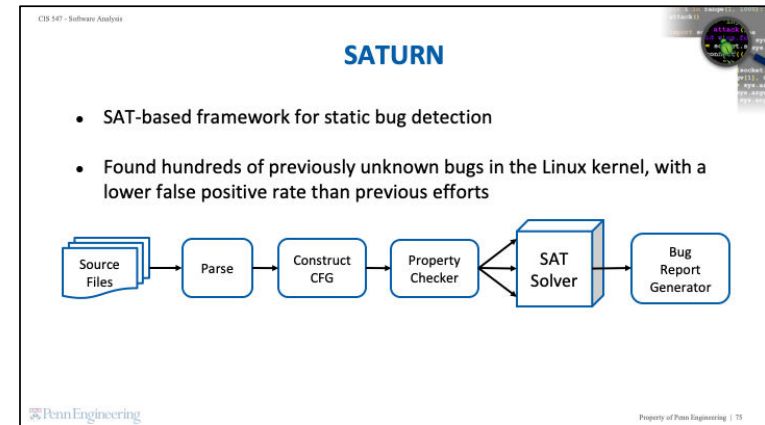
Thus, we have shown that each clause of the formula evaluates to true, thus the formula evaluates to true, and is therefore satisfiable.



A wide variety of industrial applications can be encoded as instances of the boolean satisfiability problem. General-purpose SAT solvers such as MiniSAT and Glucose, to name a few, have therefore been developed to solve this problem.

The SAT solver takes as input an arbitrary boolean formula in CNF format, and outputs either that the formula is satisfiable, along with a satisfying assignment, or that the formula is unsatisfiable, meaning that there is no satisfying assignment. For the example we just illustrated, the input and output are as follows.

The SAT problem was the first problem to be proven NP-complete in 1971. Despite this complexity, algorithmic advances have enabled modern SAT solvers to solve very large practical instances comprising tens of thousands of variables and millions of clauses in seconds.



The remarkable practical success of SAT solvers has led to their widespread use in domains ranging from planning and scheduling to program analysis. As an example, the SATURN project from Stanford University in the 2000's developed a SAT-based framework for static bug detection. SATURN found hundreds of previously unknown bugs in the Linux kernel, with a lower false positive rate than previous efforts.

The SATURN pipeline, depicted here, generates boolean constraints in three steps: it first parses the given program's source code, it then constructs its control-flow graph, and finally encodes the semantic information in the control-flow graph as a boolean formula with the goal of checking a property of interest, such as assertions or memory leaks. The boolean formula is fed to a SAT solver. If the solver outputs that the formula is satisfiable, then the satisfying assignment is used to generate a bug report in the form of a program input that exhibits the violation of the property.

SATURN scales to very large programs like the Linux kernel by using a form of summary-based inter-procedural analysis that we discussed earlier in this module. Each SAT formula thus encodes the control-flow graph of a single function rather than the entire program. The encodings of different functions in the program exchange information via summaries similar to type signatures. Let's look at an example program and property next.

Example: Static Bug-Finding

Does this program divide by zero?

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

Suppose the program is the C function `foo` shown here and the property to check is divide-by-zero bugs. The program computes the bitwise OR of arguments `x` and `y` and stores the result in `z`. It then returns the reciprocal of `z`. Let us next see how the problem of checking whether this program divides by zero can be encoded as a boolean satisfiability problem.

Example: Static Bug-Finding

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	x_{31}	...	x_0
y	y_{31}	...	y_0

To check whether this program divides by zero, we wish to find values of input variables `x` and `y`, if any, under which this program divides by zero. So the first step in the encoding is to represent `x` and `y` as variables in our boolean formula. Since the formula only allows boolean variables, whereas `x` and `y` are integer variables, we represent each bit of `x` and `y` as a separate boolean variable. In C, an integer variable is 32 bits in size, so we introduce 32 boolean variables `x0` thru `x31` to represent the integer variable `x`, and similarly another 32 boolean variables `y0` thru `y31` to represent the integer variable `y`. This technique of eagerly translating richer data types such as integers into booleans for handing over to a SAT solver is called bit-blasting.

CS 547 - Software Analysis

Example: Static Bug-Finding

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	x_{31}	...	x_0
y	y_{31}	...	y_0

Step 2: computing z

z	$x_{31} \vee y_{31}$...	$x_0 \vee y_0$
---	----------------------	-----	----------------

Penn Engineering Property of Penn Engineering | 78

The next step is to capture the computation of the bitwise OR of x and y, whose result is stored in integer variable z. Since the operation is bitwise OR, the encoding is straightforward: each of the 32 bits of z is simply the disjunction of the corresponding bits of x and y. The encoding is more complex for other operations in a C program, such as addition or multiplication, and approximations are typically used.

CS 547 - Software Analysis

Example: Static Bug-Finding

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?

$F \equiv (z = 0)$ **Bit-blasting** $F \equiv (x_{31} \vee y_{31}) = 0$
 $\wedge \dots$
 $\wedge (x_1 \vee y_1) = 0$
 $\wedge (x_0 \vee y_0) = 0$

$F \equiv \neg(x_{31} \vee y_{31})$
 $\wedge \dots$
 $\wedge \neg(x_1 \vee y_1)$
 $\wedge \neg(x_0 \vee y_0)$

$F \equiv (\neg x_{31} \wedge \neg y_{31})$
 $\wedge \dots$
 $\wedge (\neg x_1 \wedge \neg y_1)$
 $\wedge (\neg x_0 \wedge \neg y_0)$ **CNF Form**

Penn Engineering Property of Penn Engineering | 79

Our ultimate goal is to check whether z can be 0. Since the integer constant 0 corresponds to a bit pattern of 32 0's, the boolean formula that we construct equates each of the 32 bits of z with 0. This formula is not in CNF, but it can be easily transformed into one in two additional steps. We first replace each clause "xi OR yi equals 0" with the equivalent clause "not of (xi OR yi)". It is easy to see that the former clause is true if and only if the latter clause is true. We then apply de Morgan's law that the negation of a disjunction is the conjunction of the negations. At this point, we have a CNF formula which is satisfiable if and only if the original program divides by zero on some input x and y.

CIS 547 - Software Analysis

Example: Static Bug-Finding

Does this program divide by zero?

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

Step 4: is F satisfiable?

$$F \equiv (\neg x_{31} \wedge \neg y_{31}) \wedge \dots \wedge (\neg x_1 \wedge \neg y_1) \wedge (\neg x_0 \wedge \neg y_0)$$

Yes, a satisfying assignment bug-inducing input:

$x_{31} = \dots = x_0 = 0$	➔	$x = 0$
$y_{31} = \dots = y_0 = 0$		$y = 0$

Penn Engineering Property of Penn Engineering | 80

In the next step, we feed the formula to a SAT solver. The solver returns a satisfying assignment, namely, one where all 32 boolean variables corresponding to each of x and y are false. This satisfying assignment corresponds to a bug-inducing input where x and y are zero. Indeed, it is easy to see that this program divides by zero on that input.

The lecture handout points to further reading on how SATURN handles a variety of other constructs such as loops, procedures, and pointers. The handling of most of these constructs, particularly loops, is approximate, rendering SATURN neither sound nor complete.

CIS 547 - Software Analysis

SEGMENT

Analysis Using SMT

Penn Engineering Property of Penn Engineering | 81

Satisfiability Modulo Theories (SMT)

- 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**?

Next, we will introduce Satisfiability Modulo Theories, or SMT for short. The SMT problem is a generalization of the SAT problem. SAT only handles the theory of booleans while SMT extends it to handle theories involving reasoning about integers, reals, and various data structures such as arrays, strings, and bit-vectors. This avoids the need for bit blasting which in turn can improve both accuracy and efficiency of the solver. In particular, the SMT problem asks, given a first-order logic formula, whether there exists an assignment that makes the formula evaluate to true. Let's look at an example.

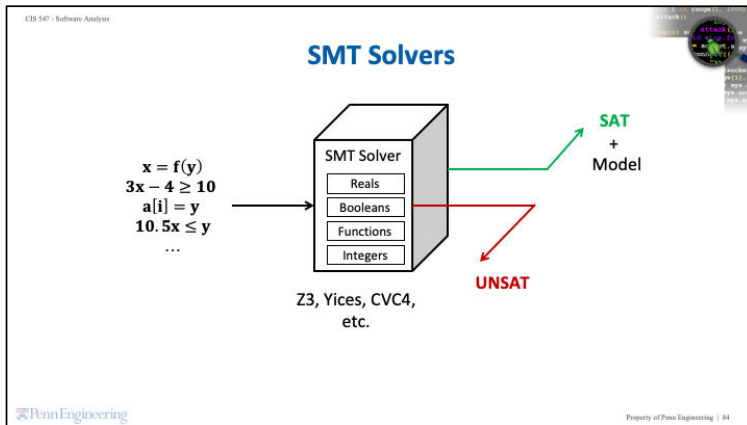
SMT: Example Formula

- Is there a satisfying assignment for the following formula?

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

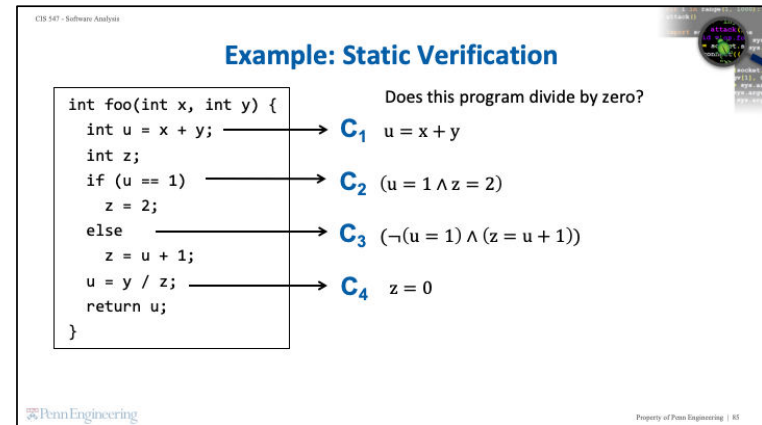
Theory of Booleans
Theory of Reals
Theory of Uninterpreted Functions

Here is an example SMT formula. Determining whether this formula is satisfiable involves reasoning about three different theories: the theory of booleans for the parts involving boolean variables p_1 and p_2 , the theory of reals for the parts involving real-valued variables x and y , and the theory of uninterpreted functions for the parts involving function variable f .



General-purpose SMT solvers such as Z3, Yices, and CVC4, to name a few, enjoy widespread use. They take an arbitrary SMT formula in SMT-LIB format and output either that the formula is satisfiable, along with a satisfying assignment called a model, or that the formula is unsatisfiable, meaning that there is no satisfying assignment.

Similar to SAT solvers, SMT solvers have many applications in software analysis and beyond. Let's revisit our problem of statically checking for divide-by-zero errors, this time using SMT instead of SAT.



As in any constraint-based analysis approach, the first step is to generate constraints. Suppose our goal to statically check whether this C program has divide-by-zero errors. For this objective, the constraints to be generated are called verification conditions. These verification conditions are generated from the program's source code in a modular fashion.

The first constraint in this example, denoted C1, captures the effect of the assignment $u = x + y$ by asserting that u must be equal to x plus y .

The second constraint, denoted C2, captures the effect of the true branch, by asserting that it is taken when u is equal to 1, and that z is equal to 2 after this branch.

Likewise, the third constraint, denoted C3, captures the effect of the false branch, by asserting that it is taken when u is not equal to 1, and that z is equal to u plus 1 after this branch.

Since our goal is to check whether the program might divide by zero, our final constraint, denoted C4, asserts that the denominator of the division operation, z is zero.

CS 547 - Software Analysis

Example: Static Verification

```

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?

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?

Penn Engineering

Property of Penn Engineering | 86

To check whether this program could divide by zero, it suffices to check whether some program path might divide by zero. There are only two paths in this program. A path divides by zero if and only if the conjunction of constraints along that path is satisfiable. These constraints are C_1 , C_2 , and C_4 along the path that follows the true branch, and C_1 , C_3 , and C_4 along the path that follows the else branch.

CS 547 - Software Analysis

Example: Static Verification

```

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?

$C_1 \rightarrow C_2 \rightarrow C_4$

$C_1 \rightarrow C_3 \rightarrow C_4$

$(u = x + y)$

$\wedge (u = 1 \wedge z = 2) \vee \wedge (\neg(u = 1) \wedge (z = u + 1))$

$\wedge (z = 0)$

$\wedge (z = 0)$

divide-by-zero bug!

SAT $x = 0, y = -1$

Penn Engineering

Property of Penn Engineering | 87

The final formula that is fed to the SMT solver is as follows: it is a disjunction of the conjunction of constraints along each of the two paths. The constraints arising from the first path turns out to be unsatisfiable since the true branch sets z to 2. On the other hand, the constraints arising from the second path are satisfiable; in this case, the SMT solver also provides a model – a satisfying assignment such as $x = 0$ and $y = -1$, which serves as a bug-inducing input for debugging the divide-by-zero error.

As an exercise, think how this program can be fixed, and convince yourself that the SMT formula generated from the resulting program is indeed unsatisfiable.

More generally, this approach can be used to check arbitrary assertions, and for programs with richer constructs such as loops and procedures. However, these expressive features pose significant challenges for automatic verification; overcoming these challenges requires human assistance in the form of additional specifications. For instance, reasoning about loops requires loop invariants, and reasoning about procedures requires pre- and post- conditions.



LESSON

Review



SEGMENT

What Have We Learned?

What Have We Learned?

- Constraint-based analysis and its benefits
- The Datalog constraint language
- How to express static analyses in Datalog
 - Analysis logic == constraints in Datalog
 - Analysis inputs and outputs == relations of tuples
- Context-insensitive and context-sensitive inter-procedural analysis
- Introduction to SAT and SMT

In this module, we have looked at the benefits of constraint-based analysis and how it separates the specification of an analysis from its implementation.

We have also seen how to use a constraint language (particularly Datalog) to set up and solve static analysis problems. The key points to remember are:

the mapping between the logic of the analysis and the constraints or rules of inference in Datalog, and

the mapping between the inputs and outputs of the analysis and the relations by which facts are asserted in Datalog.

We have also explored the difficulties and potential solutions that arise in extending pointer analysis to context-insensitive and context-sensitive forms of inter-procedural analysis.

Lastly, we introduced two other constraint languages, SAT and SMT, and showed how they can be used to check assertions.