Lecture 19

Flow Analysis

flow analysis in prolog;
applications of flow analysis

Ras Bodik
Ali and Mangpo

Hack Your Language!
CS164: Introduction to Programming
Languages and Compilers, Spring 2013
UC Berkeley

Today

Static program analysis

- what it computes and how are its results used

Points-to analysis

- analysis for understanding flow of pointer values

Andersen's algorithm

- approximation of programs: how and why

Andersen's algorithm in Prolog

- points-to analysis expressed via just four deduction rules

Andersen's algorithm via CYK parsing (optional)

- expressed as CYK parsing of a graph representing the pgm

Static Analysis of Programs

definition and motivation

Static program analysis

Computes program properties

used by a range of clients: compiler optimizers, static debuggers, security audit tools, IDEs, …

Static analysis == at compile time

- that is, prior to seeing the actual input

==> analysis answer must be correct for all possible inputs

Sample program properties:

does variable x has a constant value (for all inputs)?

does foo() return a table (whenever called, on all inputs)?

Client 1: Program Optimization

Optimize program by finding constant subexpressions

- Ex: replace *x[i]* with *x[1]* if we know that *i* is always 1.
- This optimizations saves the address computation.

This analysis is called *constant propagation*

```
i = 2
…
i = i+2
…
if (…) { … }
…
x[i] = x[i-1]
```

Q: is i a constant no matter which side of if is taken?

Q: is i always a constant here?

Q: what constant?

Client 2: Find security vulnerabilities

One specific analysis: find broken sanitization

In a web server program, as whether a value can flow from POST (untrusted source) to the SQL interpreter (trusted sink) without passing through the cgi.escape() sanitizer?

This is *taint analysis*. Can be dynamic or static.

Dynamic: Outside values are marked with a tainted bit. Sanitization clears the bit. An assertion checks that values reaching the interpreter are not tainted.
http://www.pythonsecurity.org/wiki/taintmode/

Static: a compile-time variant of this analysis. Proves that no input can ever make a tainted value flow to trusted sink.

Client 3: Optimization of virtual calls

Java virtual calls look up their target fun at run time

even though sometimes we know the fun at compile time

Analysis idea:

Determine the target function of the call statically.

If we can prove that the call has a single target, it is safe to rewrite the virtual call so that it calls the target directly.

Two ways to analyze whether a call has known target:

1. Based on declared (static) types of pointer variables:

   `Foo a = …; a.f()` // Here, a.f() could call Foo::f or Bar::f.

   // in another program, the static type may identify the unique target

2. By analyzing what values flow to rhs of a=....

   That is, we try to compute the dynamic type of var a more precisely than is given by its static type.

Example

```
class A            { void foo() {…} }
class B extends A  { void foo() {…} }
void bar(A a) { a.foo() }  // can we optimize this call?
B myB = new B();
A myA = myB;
bar(myA);
```

The declared (static) type of a permits a.foo() to call both A::foo and B::foo.

Yet we know the target must be B::foo because bar is called with a B object.

This knowledge allows optimization of the virtual call.

Client 4: Verification of casts

In Java, casts include dynamic type checks

- type system is not expressive enough to check them statically
- although Java generics help in many cases

What happens in a cast? **(Foo) e** translates to

- **if (dynamic_type_of(e) not compatible with Foo)**
  throw ClassCast Exception
- t1 compatible with t2 means t1 = t2 or t1 subclass of t2

This dynamic checks guarantees type safety but

- it incurs run time overhead
- we can't be sure the program will not throw the exception

The goal and an example

Goal: prove that no exception will happen at runtime

- this proves absence of certain class of bugs (here, class cast bugs)
- useful for debugging of high-assurance sw such that in Mars Rover

```
class SimpleContainer { Object a;
    void put (Object o) { a=o; }
    Object get() { return a; }     }
SimpleContainer c1 = new SimpleContainer();
SimpleContainer c2 = new SimpleContainer();
c1.put(new Foo()); c2.put("Hello");
Foo myFoo = (Foo) c1.get();  // verify that cast does not fail
```

Note: analysis must distinguish containers c1 and c2.

- otherwise c1 will appear to contain string objects

Property computed by analysis (clients 1-4)

Constant propagation:

- Is a variable constant at a given program point?
- If yes, what is the constant value?

Taint analysis:

- Is every possible value reaching a sensitive call untainted?
- Values coming from untrusted input are tainted.

Virtual call optimization:

- What is the possible set of dynamic types of a variable?

Cast verification:

- Same property as for virtual call optimization. A cast (Foo)t is verified as correct if the set of dynamic types of t is compatible with Foo.

Properties of Static Analysis

Static analysis must be conservative

When **unsure**, the analysis must give answer that does not **mislead** its client

ie, err on the side of caution (ie be conservative, aka sound)

Examples:

- don’t allow optimization based on incorrect assumptions, as that might change what the program computes
- don’t miss any security flaws even if you must report some false alarms

Several ways an analysis can be **unsure:**

Property holds on some but not all execution paths.

Property holds on some but not all inputs.

Examples of misleading the client:

Constant propagation:

if x is not always a constant but were claimed to be so by the analysis, this would lead to optimization that changes the semantics of the program. The optimizer would brake the program.

Taintedness analysis:

Saying that a tainted value cannot flow may lead to missing a bug by the security engineer during program review. Yes, we want to find all taintendness bugs, even if the analysis reports many false positives (ie many warnings are not bugs).

Some paths vs. all paths

Constant propagation:

analysis must report that x is a constant at some program point only if it is that constant along **all** paths leading to p.

Cast verification:

report that a variable t may be of type Foo if t is of type Foo along at least one path leading to t (need not be all paths).

Flow Analysis

What analysis can serve clients 1-4?

Is there a generic property useful to all these clients?

Yes, *flow of values.*

Value flow: how values propagate through variables

this notion covers both integer constants and objects

Points-to analysis

Points-to analysis: a special kind of value flow analysis

for pointer values (useful for clients 2-4)

The analysis answers the question:

what objects can a pointer variable point to?

It tracks flow from **creation** of an object to its **uses**

that is, flow from new Foo to myFoo.f

Note: the pointer value may flow via the heap

- that is, a pointer may be stored in an object's field
- ... and later read from this field, and so on

More on Value Flow Analysis

The flow analysis can be explained in terms of

- producers (creators of pointer values: new Foo)
- consumers (uses of the pointer value, eg, a call p.f())

Client virtual call optimization

For a given call **p.f()** we ask which expressions **new T()** produced the values that ***may*** flow to p.

we are actually interested in which values will definitely not flow

Knowing producers will tells us possible dynamic types of p.

… and thus also the set of target methods

and thus also the set of target methods which will not be called

Continued..

Client cast verification

Same, but consumers are expressions **(Type) p**.

Are they also produces?

Client 164compilation

- For each producer **new Foo** find if all consumers $e_1[e_2]$ such that the producer flows to e_1
- If there are no such consumers, Foo can be implemented as a struct.

Static Analysis Approximates the Program

Assume Java

For now, assume we're analyzing Java

- thanks to class defs, fields of objects are known statically

Also, assume the program does not use reflection

- this allows us to assume that the only way to read and write into object fields is via p.f and p.f=…

We will generalize our analysis to JS later

Why program approximation

Analyzing programs precisely is too expensive

and sometimes undecidable (ie impossible to design a precise analysis algorithm)

Hence we simplify the problem

by transforming the program into a simpler one (we say that we abstract the program)

The approximation must be conservative

- must not lose “dangerous” behavior of the original pgm
- eg: if x is not a constant in the original, it must not be a constant in the approximated program

Points-to analysis for simple programs

Initially we'll only handle new and assignments p=r:

```
if (…) p = new T1()
else   p = new T2()
r = p
r.f()       // what are possible dynamic types of r?
```

Problem for precise analysis. What may r point to?

- if the above code is in a loop, unboundedly many T1 and T2 objects could be created for some large inputs. How does the analysis keep track of them all?

Solution: abstract objects

We (conceptually) translate the program to

```
if (…) p = o₁
else   p = o₂
r = p
r.f()       // what are possible symbolic constant values r?
```

o_1 is an abstract object

- ie, a symbolic constant standing for all objects created at that allocation

Abstract objects

The o_i constants are called abstract objects

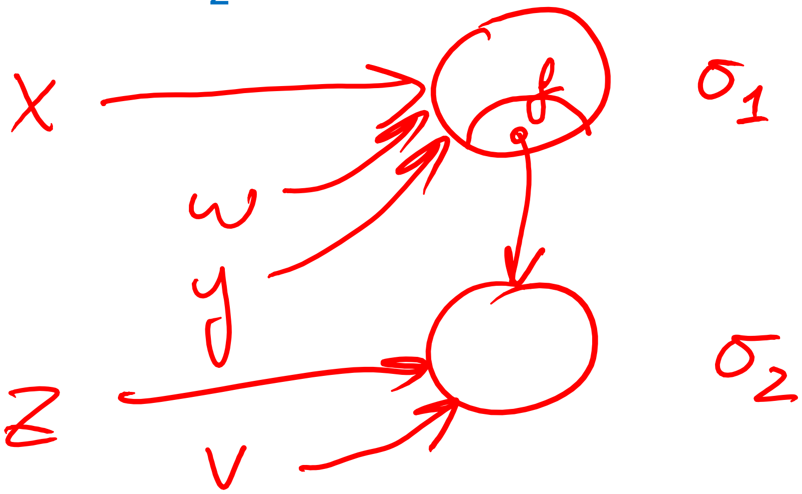
- an abstract object o_i stands for any and all dynamic objects allocated at the allocation site with number i
- allocation site = a new expression
- each new expression is given a number i
- you can think of the abstract object as the result of collapsing all objects from this allocation site into one

When the analysis says a variable p may have value o_7

- we know that p may point to any object allocated in the expression "new_7 Foo"

We now consider pointer dereferences p.f

```
✓ x = new Obj();      // o1
✓ z = new Obj();      // o2
✓ w = x;
✓ y = x;
  y.f = z;
  v = w.f;
```



To determine abstract objects that v may reference, what new question do we need to answer?

Q: can y and w point to same object?

Keeping track of the heap state

Heap state:

1) what abstract objects a variable may point to
2) what objects may fields of abstract objects point to.

The heap state may change after each statement

may be too expensive to track

Analyses often don't track state at each point separately

- to save space, they collapse all program points into one
- consequently, they keep a single heap state

This is called flow-insensitive analysis

why? see next slide

Flow-Insensitive Analysis

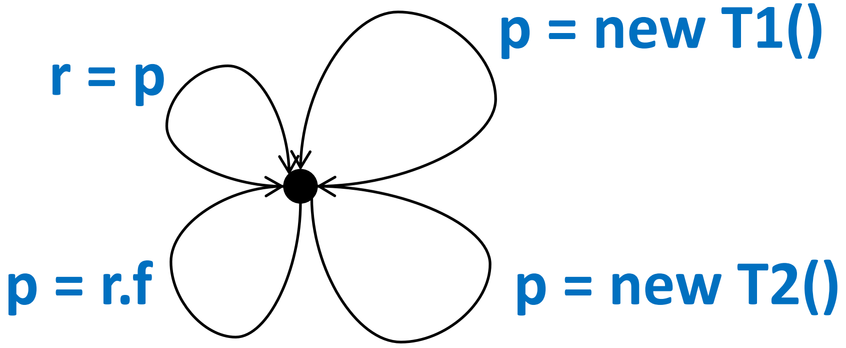
Disregards the control flow of the program

- assumes that statements can execute in any order …
- … and any number of times

Effectively, flow-insensitive analysis transforms this

```
if (...) p = new T1(); else p = new T2();
r = p; p = r.f;
```

into this control flow graph:



Flow-Insensitive Analysis

Motivation:

- there is a single program point,
- and hence a single “version” of heap state

Is flow-insensitive analysis sound?

- yes: each execution of the original program is preserved
- and thus will be analyzed and its effects reflected

But it may be imprecise

1) it adds executions not present in the original program
2) it does not distinguish value of p at distinct pgm points

Summary

Approximations we made to make analysis feasible:

- Abstract objects: collapse objects
- flow-insensitive: collapse program points

Representing the Program in a Small Core Langauge

Canonical Stmts

Java programs contain complex expressions:

- ex: **`p.f().g.arr[i] = r.f.g(new Foo()).h`**

Can we find a small set of canonical statements?

- ie, a core language understood by the analysis
- we'll desugar the rest of the program to these stmts

Turns out we only need four canonical statements:

p = new T()	*new*
p = r	*assign*
p = r.f	*getfield*
p.f = r	*putfield*

Canonical Statements, discussion

Complex statements can be canonized

```
p.f.g = r.f
   →
t1 = p.f
t2 = r.f
t1.g = t2
```

Can be done with a syntax-directed translation

- like translation to byte code in PA2

Algorithm for Flow Analysis

Andersen's Algorithm

For flow-insensitive flow analysis:

Goal: compute two binary relations of interest:

- *x* **pointsTo** *o:* holds when x may point to abstract object o
- *o* **flowsTo** *x:* holds when abstract object o may flow to x

These relations are inverses of each other

x **pointsTo** *o* <==> *o* **flowsTo** *x*

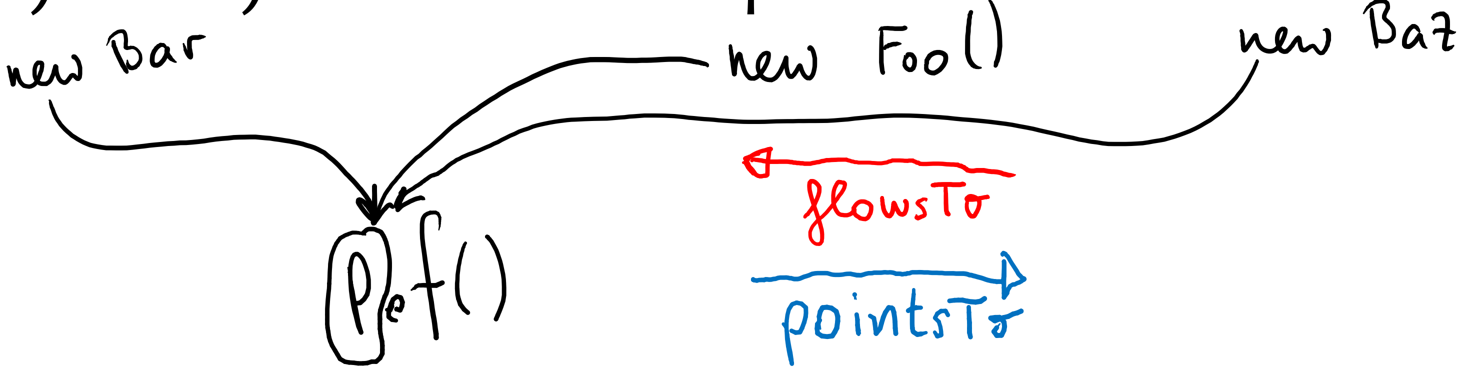
These two relations support our clients

These relations allows determining:

- target methods of virtual calls
- verification of casts
- how JavaScript objects are used (see later in slides)

For the last one, we need the flowsTo relation

For 1) and 2) we need the x pointsTo o relation



Inference rule (1)

p = new_i T()

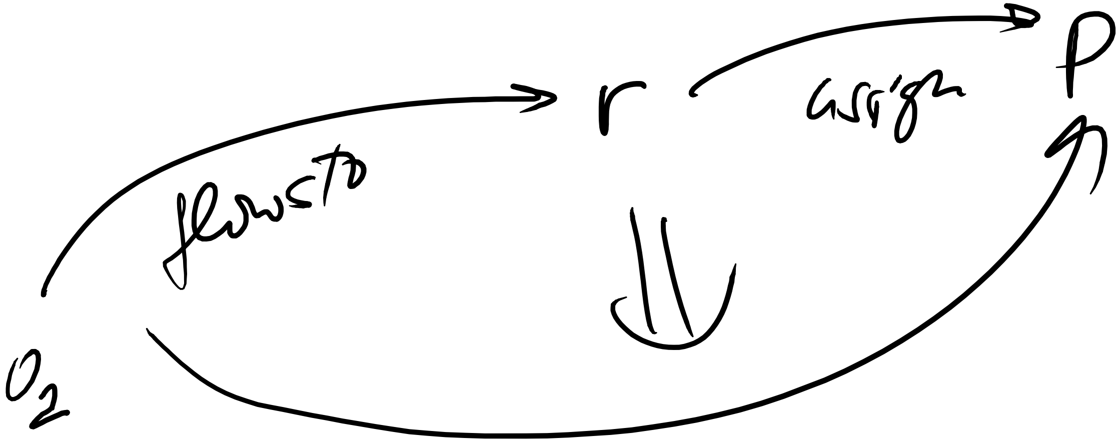
o_i *new p*

o_i new $p \longrightarrow o_i$ *flowsTo* p

Inference rule (2)

p = r *r assign p*

o_i flowsTo r ∧ r assign p → *o_i flowsTo p*



Inference rule (3)

p.f = a $\quad\quad$ $a \;\; pf(f) \;\; p$

b = r.f $\quad\quad$ $r \;\; gf(f) \;\; b$

$$o_i \; \mathit{flowsTo} \; a \;\; \wedge \;\; a \; pf(f) \; p \;\; \wedge \;\; p \; \mathit{alias} \; r \;\; \wedge \;\; r \; gf(f) \; b$$

$$\rightarrow \quad o_i \;\; \mathit{flowsTo} \;\; b$$

Inference rule (4)

it remains to define *x alias y*

(x and y may point to same object):

o_i flowsTo x ∧ o_i flowsTo y → x alias y

Prolog program for Andersen algorithm

```
new(o1,x).          % x=new_1 Foo()
new(o2,z).          % z=new_2 Bar()
assign(x,y).        % y=x
assign(x,w).        % w=x
pf(z,y,f).          % y.f=z
gf(w,v,f).          % v=w.f

flowsTo(O,X) :- new(O,X).
flowsTo(O,X) :- assign(Y,X), flowsTo(O,Y).
flowsTo(O,X) :- pf(Y,P,F), gf(R,X,F), aliasP,R), flowsTo(O,Y).

alias(X,Y)   :- flowsTo(O,X), flowsTo(O,Y).
```

Java program

How to conservatively use result of analysis?

When the analysis infers *o* flowsTo *y*, what did we prove?

- nothing useful, usually, since *o* flowsTo *y* does not imply that there definitely is a program input for which o will definitely flow to y.

The useful result is when the analysis **doesn't** infer *o* flowsTo *y*

- then we have proved that o **cannot** flow to y for any input
- this is useful information!
- it may lead to optimization, verification, compilation

Same arguments apply to alias, pointsTo relations

- and other static analyses in general

Example

Inference Example (1)

The program:

```
x = new Foo(); // o1
z = new Bar(); // o2
w = x;
y = x;
y.f = z;
v = w.f;
```

Inference Example (2):

The program is converted to six facts:

o_1 new x

x assign w

z pf(f) y

o_2 new z

x assign y

w gf(f) v

Inference Example (3), infering facts

o_1 new x | o_2 new z
x assign w | x assign y
z pf(f) y | w gf(f) v

The inference:

o_1 new x $\rightarrow$ o_1 flowsTo x

o_2 new z $\rightarrow$ o_2 flowsTo z

o_1 flowsTo x $\wedge$ x assign w $\rightarrow$ o_1 flowsTo w

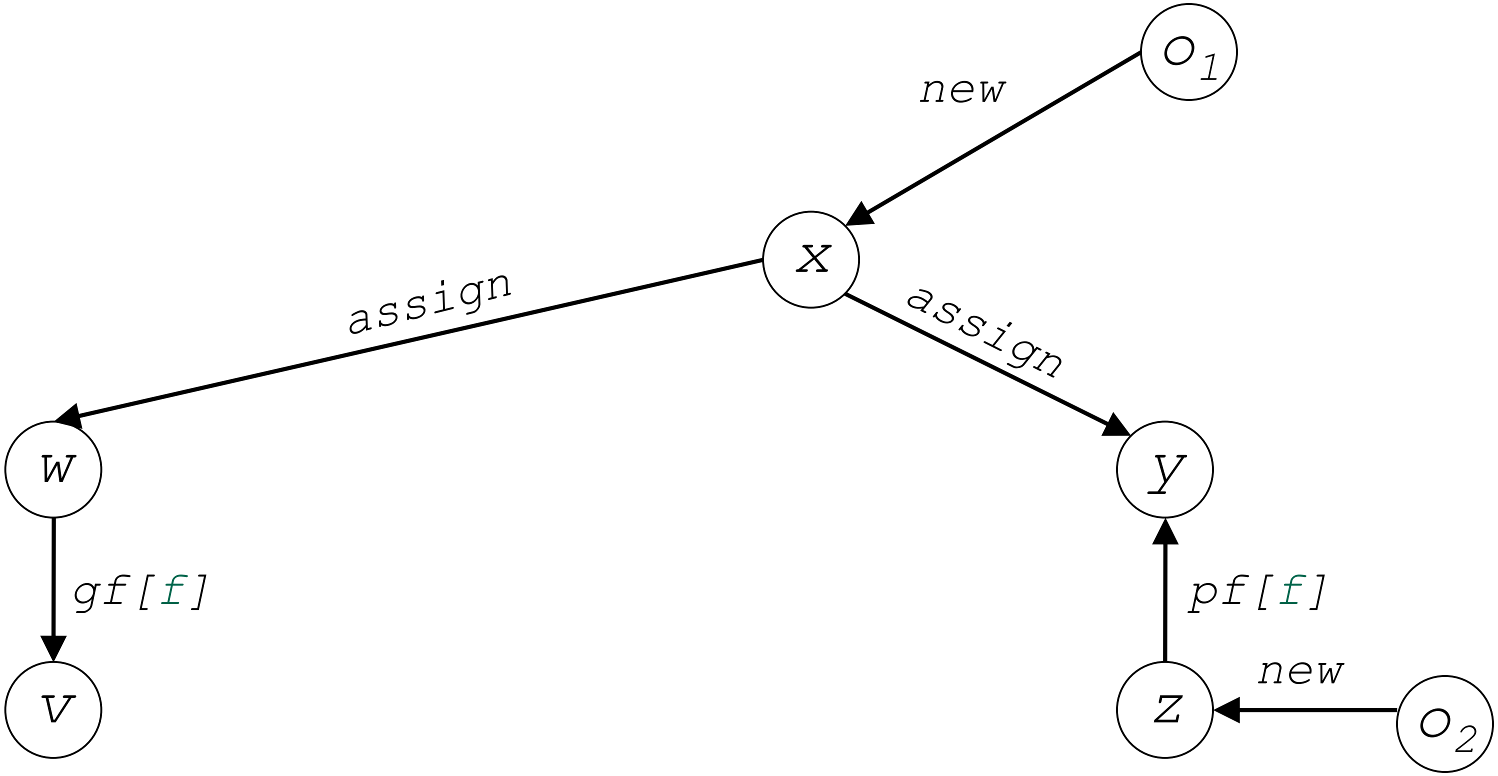
o_1 flowsTo x $\wedge$ x assign y $\rightarrow$ o_1 flowsTo y

o_1 flowsTo y $\wedge$ o_1 flowsTo w $\rightarrow$ y alias w

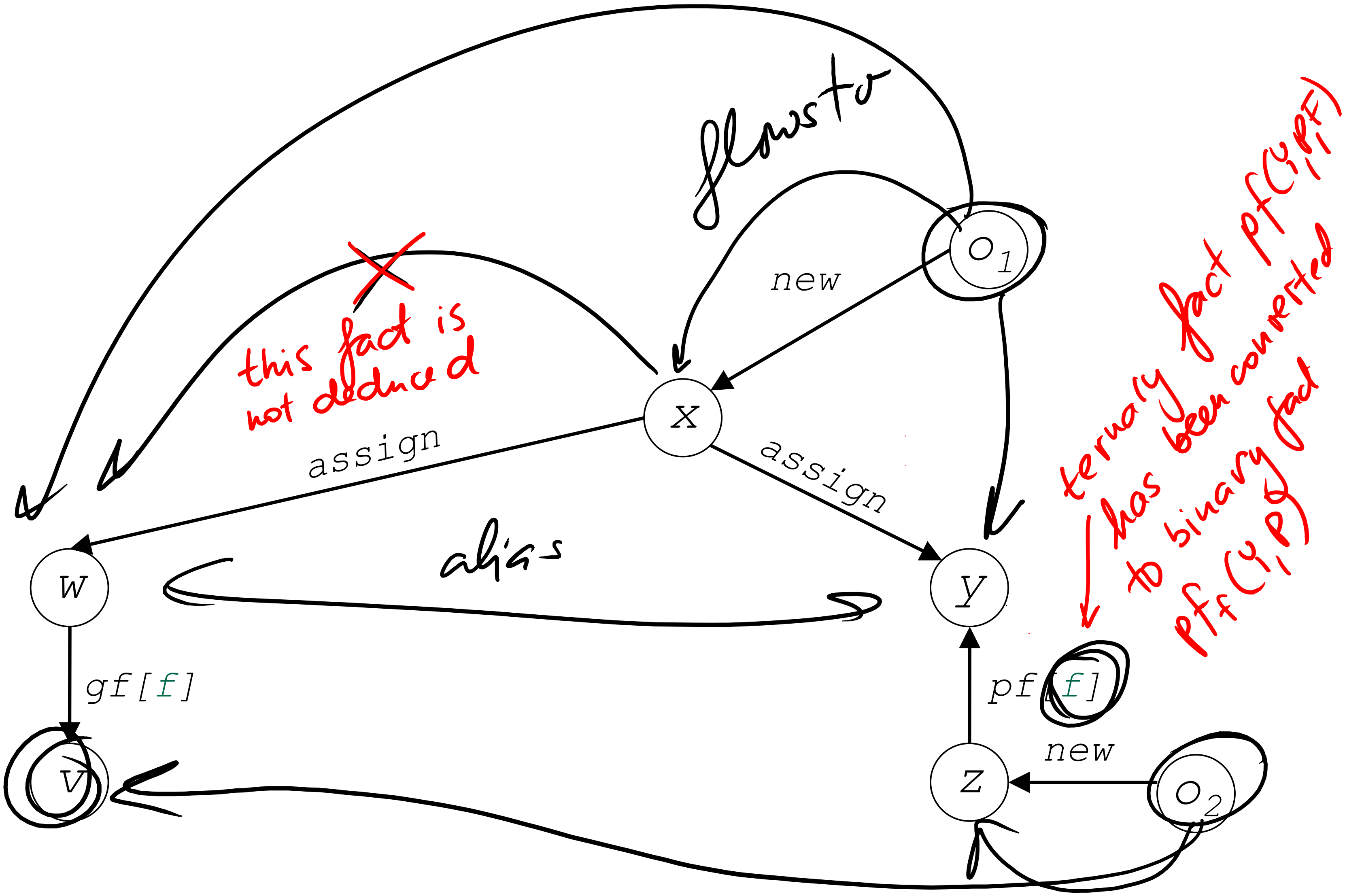
o_2 flowsTo z $\wedge$ z pf(f) y $\wedge$ y alias w $\wedge$ w gf(f) v $\rightarrow$
o_2 flowsTo v

…

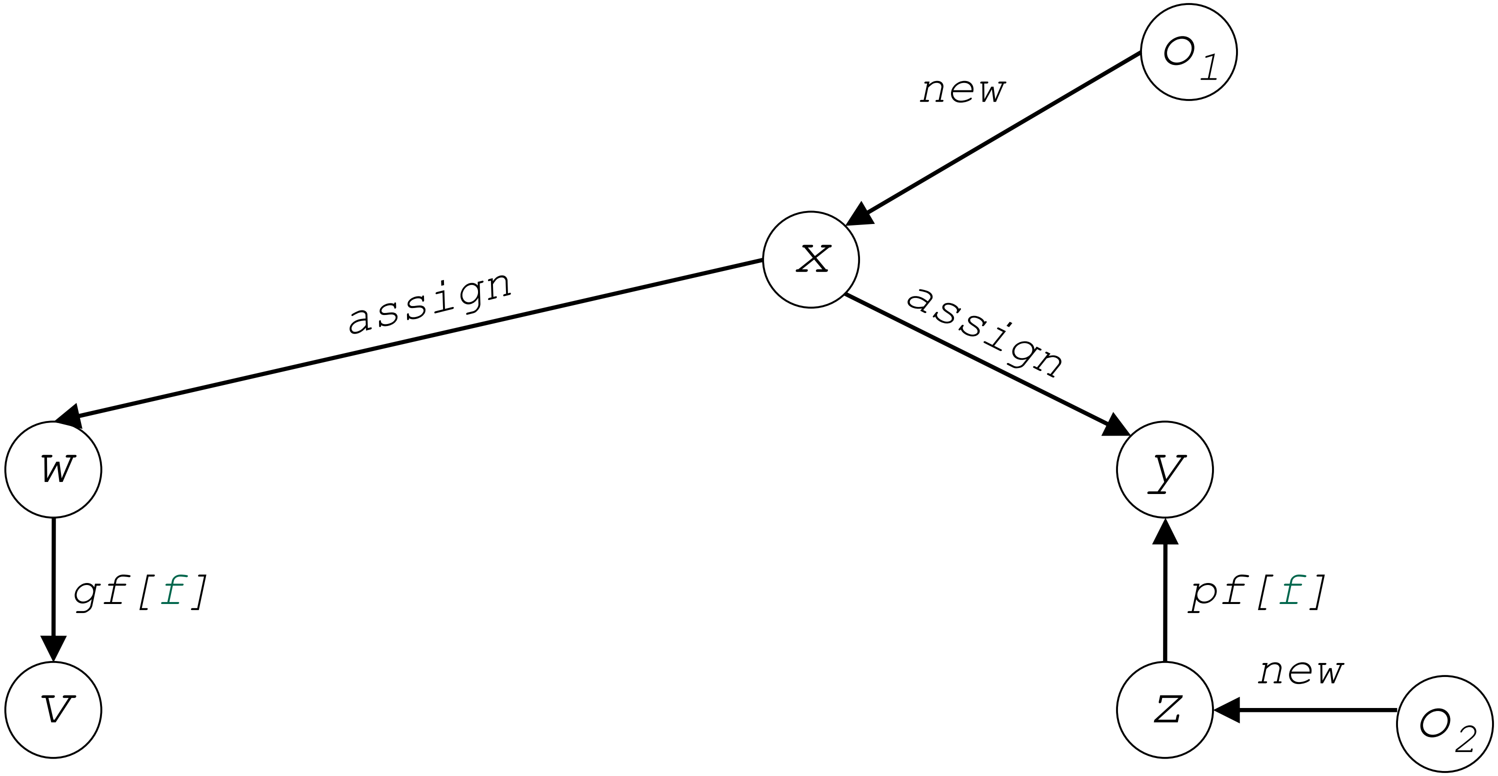
Example: visualizing Prolog deductions



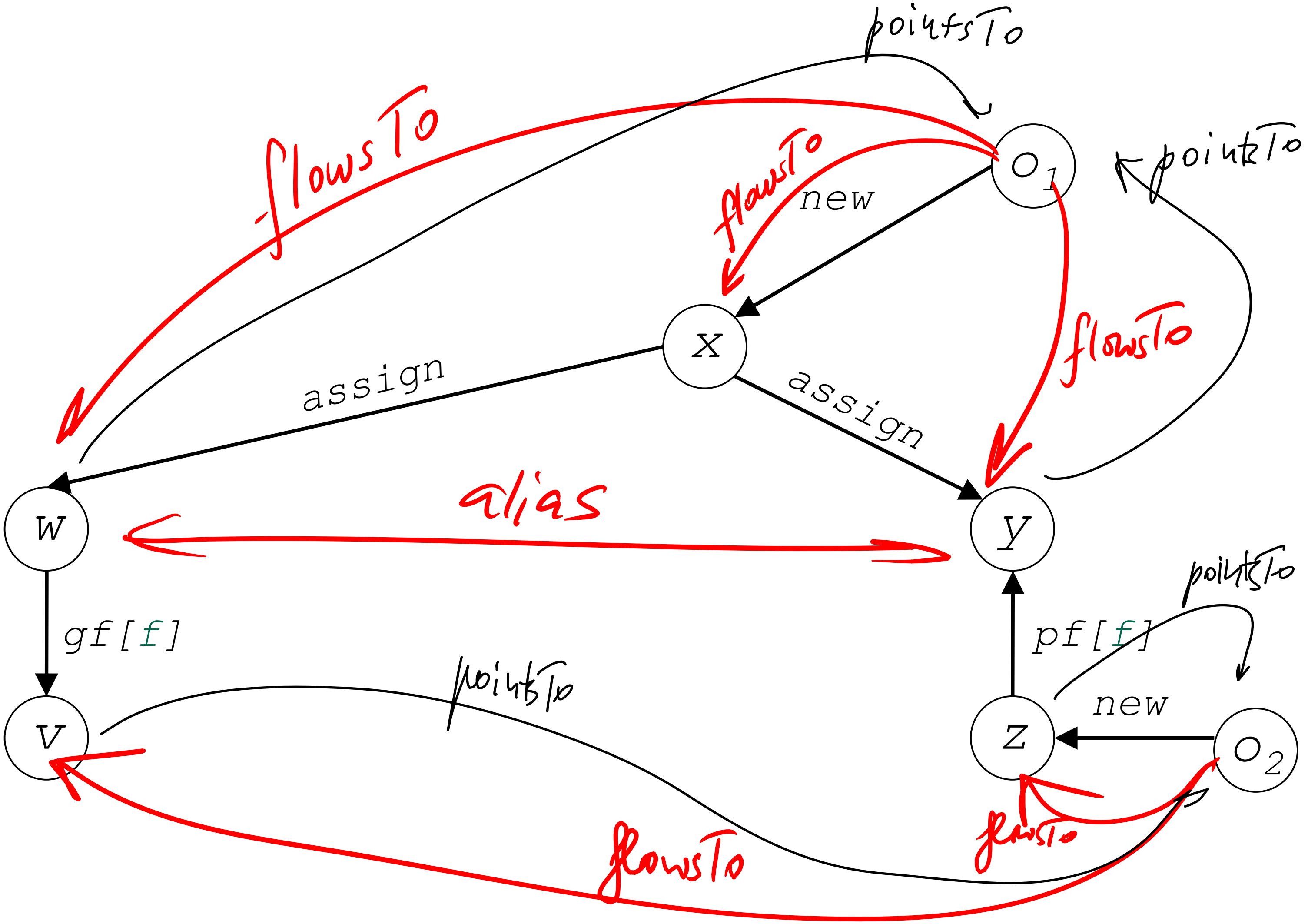
Example: visualizing Prolog deductions



Example: visualizing Prolog deductions



Example, deriving the relations



Example (4):

Notes:

- inference must continue until no new facts can be derived
- only then we know we have performed sound analysis

Conclusions from our example inference:

- we have inferred o_2 flowsTo *v*
- we have NOT inferred o_1 flowsTo *v*
- hence we know v will point only to instances of Bar
- (assuming the example contains the whole program)
- thus casts (Bar) v will succeed
- similarly, calls v.f() are optimizable

Important Odds and Ends

Handling of method calls

Issue 1: Arguments and return values:

- these are translated into assignments of the form p=r

Example:

```
Object foo(T x) { return x.f }
r = new T
s = foo(r.g)
```

is translated into

```
foo_retval = x.f    // Object foo(T x) { return x.f }
r = new T
s = foo_retval; x = r.g   // s = foo(r.g)
```

Handling of method calls

Issue 2: targets of virtual calls

- call p.f() may call many possible methods
- to do the translation shown on previous slide, must determine what these target methods are

Suggest two simple methods:

- examine (static) type hierarchy (classes and subclasses)
- compute (with our flow analysis) possible dynamic types of p

see another example in the section notes

Handling of arrays

We collapse all array elements into one element

- this array element will be represented by a field `arr`
- ex: **p.g[i] = r** becomes **p.g.arr = r**

Adaptation for JavaScript

to read more about the practical issues, see
"Fast and Precise Hybrid Type Inference for JavaScript"
by Brian Hackett and Shu-yu Guo from Mozilla

Adaptation for JavaScript

We developed the analysis for Java.

- Java objects are instances of classes
- their set of fields is fixed and known at compile time

In JS, objects are implemented as dictionaries

their fields can be added, even removed, during execution

We need to handle more language constructs:

attribute read: $e_1[e_2]$ // note e_2 is an expr, not a literal

attribute write: $e_1[e_2] = e_3$

Client 5: Compilation of objects in Lua/JS

Goal: compile 164 expression p.f1 into efficient code.

- If p refers only to tables that contains the attributed f1, we can represent the table as a struct and compile p[“f1”] into an (efficient) instruction “load from address in p + 4 bytes”.
  - A few additional conditions must be met before this optimization can be performed. (See the next slide)

Analysis needed for this optimization:

- Determine at compile time what fields the objects referred to by p might contain at run time.

- We hope the analysis will answer that all objects referred to by p will contain attribute f1.

Client 5 in more detail

Our approach:

- for each object constructor C, determine expressions E accessing objects created in C (Q_1)
- if expressions in E are all of the form p.field (not p[e]), we can have C allocate structs rather than as dicts … (Q_2)
- … provided expressions in E do not refer to objects not from C (Q_3)

Q_1 and Q_3 can be answered with points-to analysis

Q_2 is a simple syntactic check

Example

A JS program

```
var p  = new Foo;   // line 1
var r  = p.field;
var s  = {};
s[r.f] = p;
var t  = s[input()];
   t.g = …
```

Consider the Foo objects created in line 1:

- We want to determine at compile time what fields these Foo objects will contain during their lifetime?
- Is it possible to determine in this program a precise set of fields in Foo? Can we compute a safe superset of fields?

Continued

If Foo objects were not accessed via e[e], then we can compute at least (a superset of) Foo fields.

So, can we tell if this program access Foo’s via e[e]?

Let’s do a manual analysis

- our goal is to illustrate the issues with e[e] in the analysis
- lets’ denote fields(Foo) the superset of fields in Foo’s

```
var p   = new Foo;     // fields(Foo) = {}
var r   = p.field;     // fields(Foo) = {field}
var s   = {};          // no change to fields(Foo)
s.a     = p;           // no change to fields(Foo)
                       // s.a a Foo object
var t   = s[input()]; // s[input()] could be a Foo
    t.g = …            // fields(Foo) = {field, g}
```

The optimization

We perform the optimization for each allocation site C

$Q_1(V_C,C)$:

find set v of variables V_C such that C flowsTo v.

$Q_2(V_C)$:

if any variable v in V_C is used in expression v[e]
then we cannot optimize C;
if v is used in v.f, add f to fields(C)

$Q_3(V_C)$:

if any v in V_C pointsTo a C' such that C' != C
then we cannot optimize C

If C can be optimized:

- create a struct with fields fields(C), allocate it at C

Notes

Success of this analysis depends on

- the precision of the analysis and
    - there are analyses more accurate then Andersen
- on the nature of the program
    - some JS objects can’t be compiled this way because the set of their fields varies at runtime

The analysis rule

A conservative rule (conservative=sufficient but not necessary):

Compute, at compile time:

- the set of fields are added to the table using stmt e.ID=e
- the table’s fields must not be written or read through operator e[e] (only through e.ID)

Notes

“Parsing the graph”

Visualization of inferences on slides 47 and 49 parses the strings in the “graph of binary facts” using the CYK algorithm (Lecture 8)

Details on this style of inference are in the rest of the slide, under CFL-reachability (optional material)

Summary

Determine run-time properties of programs statically

- example property: “is variable x a constant?”

Statically: without running the program

- it means that we don’t know the inputs
- and thus must consider all possible program executions

We want *sound* analysis: err on the side of caution.

- allowed to say x is not a constant when it is
- not allowed to say x is a constant when it is not

Static analysis has many clients

- optimization, verification, compilation

CFL-Reachability

deduction via parsing of a graph

Inference via graph reachability

Prolog's search is too general and expensive.

may in general backtrack (exponential time)

Can we replace it with a simpler inference algorithm?

possible when our inference rules have special form

We will do this with CFL-rechability

it's a generalized graph reachability

(Plain) graph reachability

Reachability Def.:

- Node x is **reachable** from a node y in a directed graph G if there is a path p from y to x.

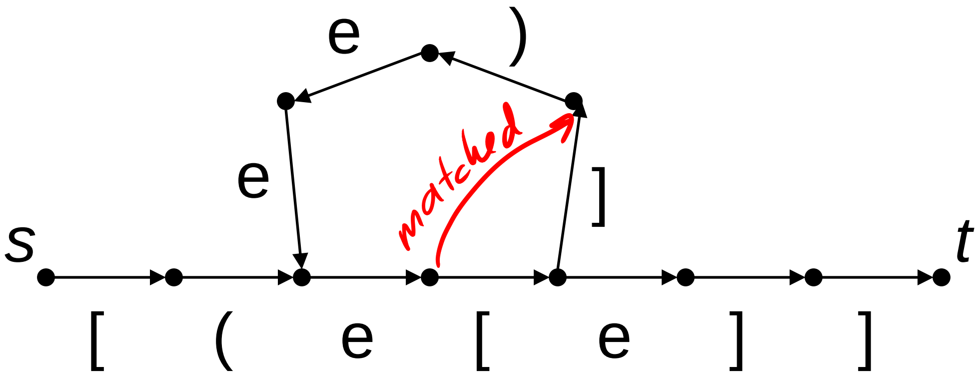
How to compute reachability?

- depth-first search, complexity O(N+E)

Context-Free-Language-Reachability

CFL-Reachability Def.:

Node x is ***L*-reachable** from a node y in a directed labeled graph G if
- there is a path p from y to x, and
- path p is labeled with a string from a context free language L.



The context-free language L:

matched → *matched matched*
| (*matched*)
| [*matched*]
| *e*
| ε

Is t reachable from s according to the language L?

Computing CFL-reachability

Given

- a labeled directed graph P and
- a grammar G with a start nonterminal S,

we want to compute whether x is S-reachable from y

- for all pairs of nodes x,y
- or for a particular x and all y
- or for a given pair of nodes x,y

We can compute CFL-reachability with CYK parser

- x is S-reachable from y if CYK adds an S-labeled edge from y to x
- $O(N^3)$ time

Convert inference rules to a grammar

The inference rules

```
ancestor(P,C) :- parentof(P,C).
ancestor(A,C) :- ancestor(A,P), parentof(P,C).
```

Language over the alphabet of edge labels

```
ANCESTOR  ::=  parentof
            |   ANCESTOR  parentof
```

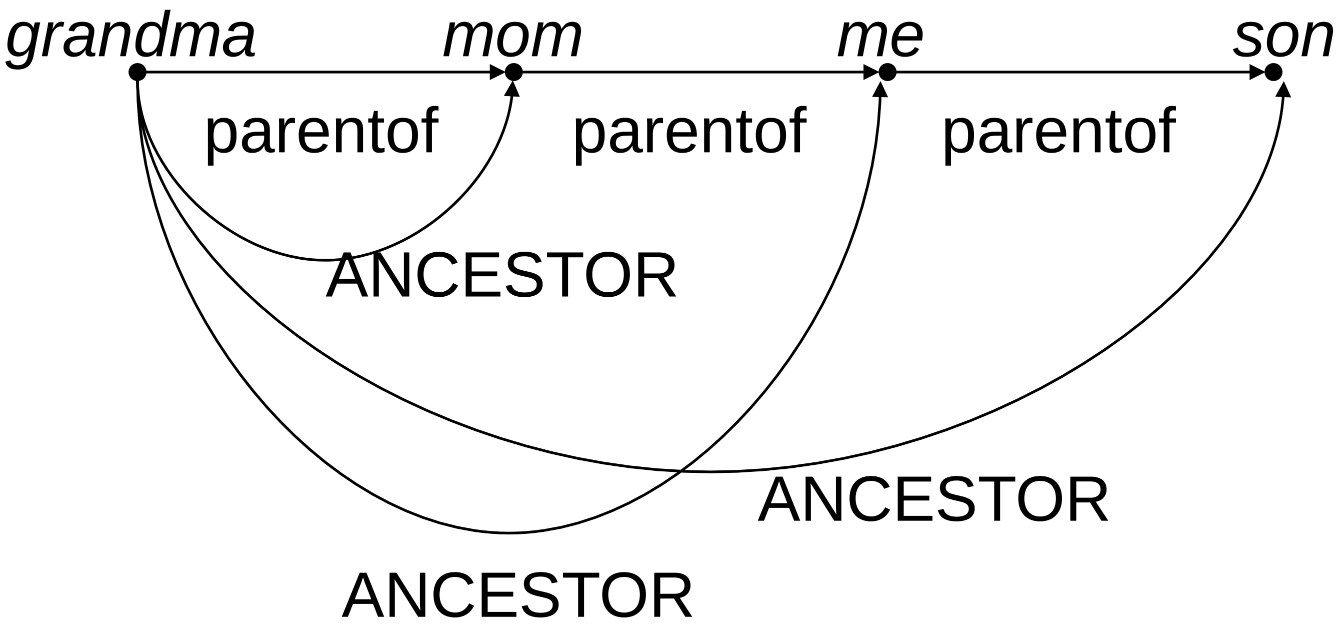
Notes:

- initial facts are terminals (perentof)
- derived facts are non-terminals (ANCESTOR)

So, which rules can be converted to CFL-reachability?

ANCESTOR ::= parentof | ANCESTOR parentof

Is “son” ANCESTOR-reachable from “grandma”?



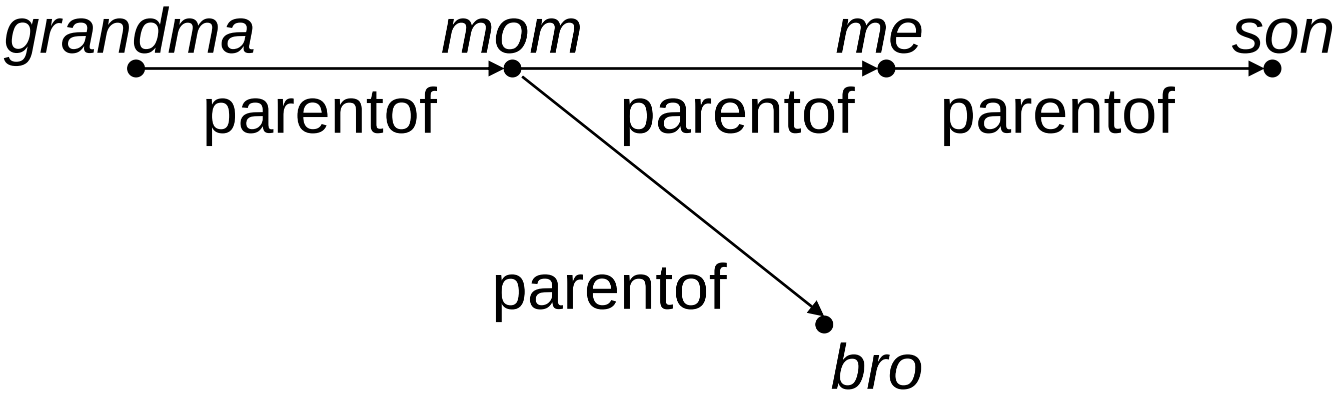
What rules can we convert to CFL-rechability?

Let’s add a rule for SIBLING:

ANCESTOR ::= parentof | ANCESTOR parentof

SIBLING ::= ???

We want to ask whether “bro” is SIBLING-reachable from “me”.



Conditions for conversion to CFL-rechability

- Not all inference rules can be converted
- Rules must form a "chain program"
- Each rule must be of the form:

  `foo(A,D) :- bar(A,B), baz(B,C), baf(C,D)`
- Ancestor rules have this form

  `ancestor(A,C) :- ancestor(A,P), parentof(P,C).`
- But the Sibling rules cannot be written in chain form
  - why not? think about it also from the CFL-reachability angle
  - no path from x to its sibling exists, so no SIBLING-path exists
    - no matter how you define the SIBLING grammar

Andersen’s Algorithm with Chain Program

converts the analysis into a graph parsing problem

Back to Andersen’s analysis

Rules in logic programming form:

```
flowsTo(O,X) :- new(O,X).
flowsTo(O,X) :- flowsTo(O,Y), assign(Y,X).
flowsTo(O,X) :- flowsTo(O,Y), pf(Y,P,F), alias(P,R),
    gf(R,X,F).
alias(X,Y)   :- flowsTo(O,X), flowsTo(O,Y).
```

Problem: some predicates are not binary

Andersen's algorithm inference rules

Translate to binary form

put field name into predicate name,

must replicate the third rule for each field in the program

```
flowsTo(O,X) :- new(O,X).
flowsTo(O,X) :- flowsTo(O,Y), assign(Y,X).
flowsTo(O,X) :- flowsTo(O,Y), pf[F](Y,P),
                        alias(P,R), gf[F](R,X).
alias(X,Y)   :- flowsTo(O,X), flowsTo(O,Y).
```

Andersen’s algorithm inference rules

Now, which of these rules have the chain form?

flowsTo(O,X) :- new(O,X).	yes
flowsTo(O,X) :- flowsTo(O,Y), assign(Y,X).	yes
flowsTo(O,X) :- flowsTo(O,Y), pf[F](Y,P), alias(P,R), gf[F](R,X).	yes
alias(X,Y) :- flowsTo(O,X), flowsTo(O,Y).	no

Making alias a chain rule

We can easily make alias a chain rule with pointsTo. Recall:

```
flowsTo(O,X)  :-  pointsTo(X,O)
pointsTo(X,O) :- flowsTo(O,X)
```

Hence

```
alias(X,Y)   :- pointsTo(X,O), flowsTo(O,Y).
```

If we could derive **chain** rules for pointsTo, we would be done. Let’s do that.

Idea: add terminal edges also in opposite direction

For each edge o new x, add edge x new^{-1} o

- same for other terminal edges

Rules for pointsTo will refer to the inverted edges

- but otherwise these rules are analogous to flowsTo

What it means for CFL reachability?

there exists a path from o to x labeled with s $\in$ L(flowsTo)

$\Leftrightarrow$

there exists a path from x to o labeled with s’$\in$L(pointsTo).

Inference rules for pointsTo

$p = new_i\ T()$ o_i new p p new^{-1} o_i

o_i new p → o_i flowsTo p Rule 1

p new^{-1} o_i → p pointsTo o_i Rule 5

p = r r assign p p $assign^{-1}$ r

o_i flowsTo r and r assign p → o_i flowsTo p Rule 2

p $assign^{-1}$ r and r pointsTo o_i → p pointsTo o_i Rule 6

Inference rules for pointsTo (Part 2)

We can now write *alias* as a chain rule.

p.f = a	a pf(f) p	p pf(f)$^{-1}$ a
b = r.f	r gf(f) b	b gf(f)$^{-1}$ r

o_i flowsTo a □ a pf(f) p □ p alias r □ r gf(f) b → o_i flowsTo b
b gf(f)$^{-1}$ r □ r alias p □ p pf(f)$^{-1}$ a □ a flowsTo o_i → b pointsTo o_i
Rules 3, 7

Both flowsTo and pointsTo use the same alias rule:
x pointsTo o_i □ o_i flowsTo y → x alias y Rule 8

The reachability language

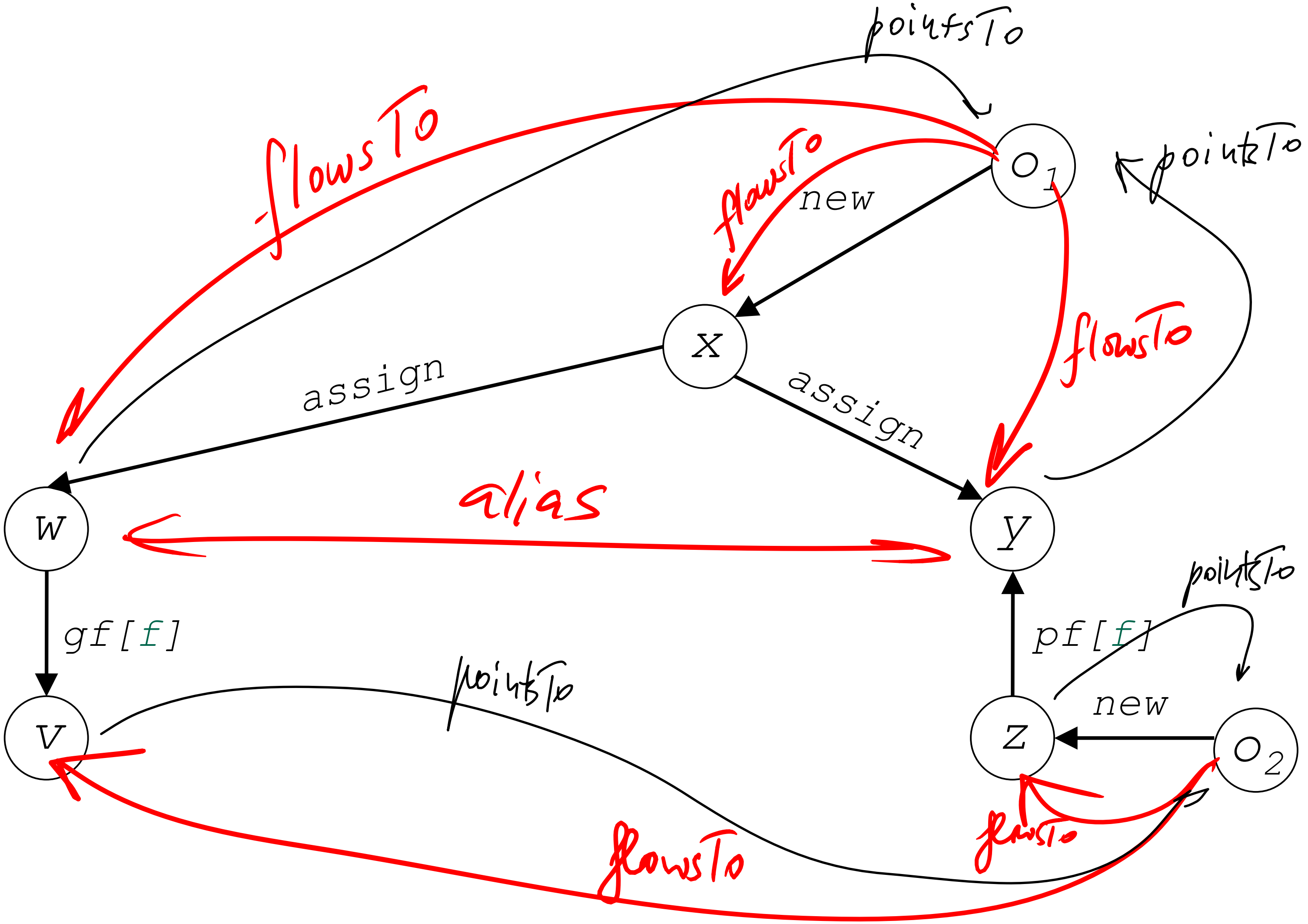
All rules are chain rules now

- directly yield a CFG for flowsTo, pointsTo via CFL-reachability :

flowsTo → new
flowsTo → flowsTo assign
flowsTo → flowsTo pf[f] alias gf[f]
pointsTo → new^{-1}
pointsTo → assign^{-1} pointsTo
pointsTo → gf[f]$^{-1}$ alias pf[f]$^{-1}$ pointsTo
alias → pointsTo flowsTo

Example: computing pointsTo-, flowsTo-reachability

Inverse terminal edges not shown, for clarity.



Summary (Andersen via CFL-Reachability)

The pointsTo relation can be computed efficiently

- with an O(N^3) graph algorithm

Surprising problems can be reduced to parsing

- parsing of graphs, that is

CFL-Reachability: Notes

The context-free language acts as a filter

- filters out paths that don't follow the language

We used the filter to model program semantics

- we filter out those pointer flows that cannot actually happen

What do we mean by that?

- consider computing x *pointsTo o* with "plain" reachability
  - plain = ignore edge labels, just check if a path from x to o exists
- is this analysis sound? yes, we won't miss anything
  - we compute a *superset* of pointsTo relation based on CFL-reachability
- but we added infeasible flows, example:
  - wrt plain reachability, pointer stored in p.f can be read from p.g