Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 14: Static analysis

Dataflow analysis
Pointer analysis

Where are we in the course

Last lecture was on **static types**

programmer annotates expressions, variables, … with information about their runtime types

Type checker checks these annotations

after all, the programmer could make a mistake;

part of the checks done at runtime (dynamic checks)

Performance: Compilers can use these annotations

generate faster code and smaller object representations

Correctness: static types ensure absence of errors

certain kinds of error can’t happen,
hence software is more secure

Today: properties can be inferred!

Static program analysis

- what is it and why do it

Dataflow analysis

- and partial redundancy elimination

Points-to analysis

- static analysis for understanding how pointer values flow

Andersen’s algorithm for points-to analysis

- via deduction

Andersen’s algorithm via CYK parsing (optional mat.)

- CYK parsing on a graph == CFL-reachability

Static program analysis

Answers questions about program properties

- related to static type inference, which infers types

Static analysis == at *compile time*

- that is, prior to seeing the actual input
- hence, the answer must be correct for all possible inputs

Sample program properties:

Does var x have a constant value (for all inputs)?

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

Motivation for static program analysis (1)

Optimize the program.

Ex: replace `x[i-1]` with `x[1]` if we know that i is always 2.

Constant propagation

what can prevent i = 2 ?

```
i = 0
…
i = i+2
…
if (…) { i++ … }
…
x[i] = x[i-1]
```

a loop

and so on

increment to i in <u>some</u> paths only

static analysis must account for these

Motivation for static program analysis (1)

Optimize the program.

Ex: replace x[i-1] with x[1] if we know that i is always 1.

Constant propagation

i = 0

…

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?

Motivation for static analysis (2)

Find potential security vulnerabilities

Ex: in a server program, can a value flow from POST (untrusted, tainted source) to SQL interpreter (trusted sink) without passing through cgi.escape (a sanitizer)?

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

Dynamic: mark values with a tainted bit. Sanitization clears the bit. An assertion checks that tainted values do not reach the interpreter. 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.

Static analysis must be conservative

When **unsure**, the analysis must answer such that it does not **mislead** the client of the analysis.

- Err on the side of caution. Say, never optimize the program such that it outputs a different value.

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

- Property holds on some but not all execution paths.

- Property holds on some but not all inputs.

Misleading the client:

Constant propagation:

if x is not always a constant but is claimed to be so by the analysis to the client (the optimizer), this would lead to optimization that changes the semantics of the program. The optimizer broke 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 find to find all taintedness bugs, even if the analysis reports many false positives (i.e., many warnings are not bugs).

Data flow analysis

Common subexpression elimination

why subexpression? x+y may be common to bigger expressions, such as A[p+x+y] and B[q+x+y]

z := x + y

w := x + y

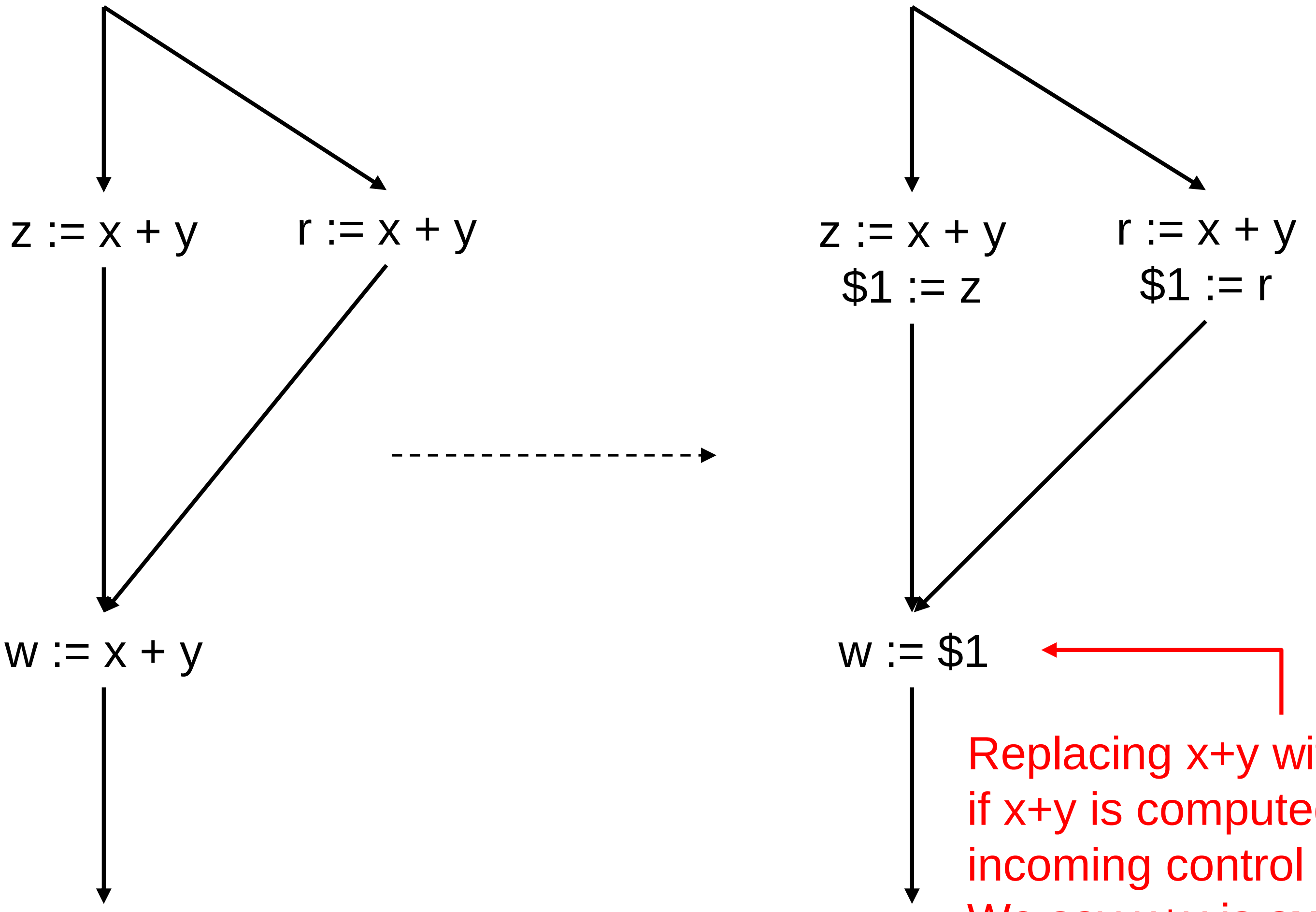
optimization

z := x + y
$1 := z

w := $1

When is CSE legal to perform?

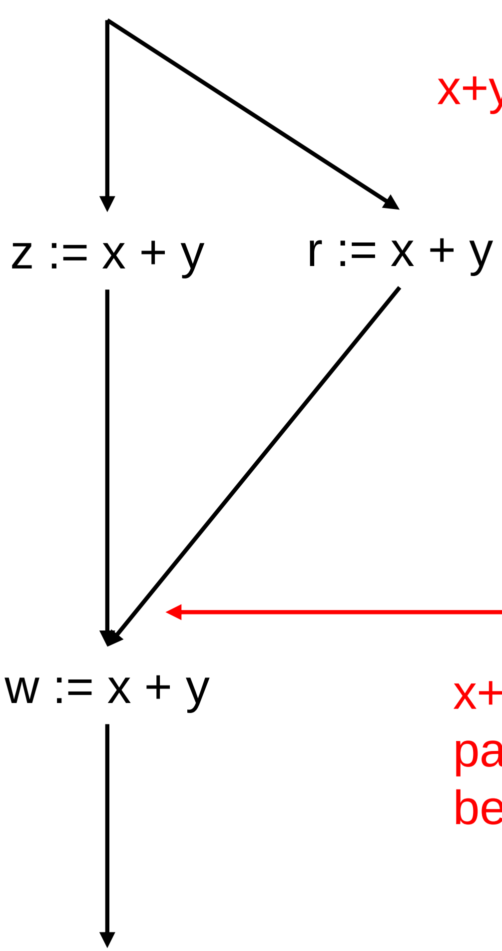
The graph is a control-flow graph (CFG).



Replacing x+y with $1 is legal if x+y is computed on all incoming control flow paths. We say x+y is **available**.

Data flow analysis computes program facts

Our fact of interest is "availability of the value of x+y"



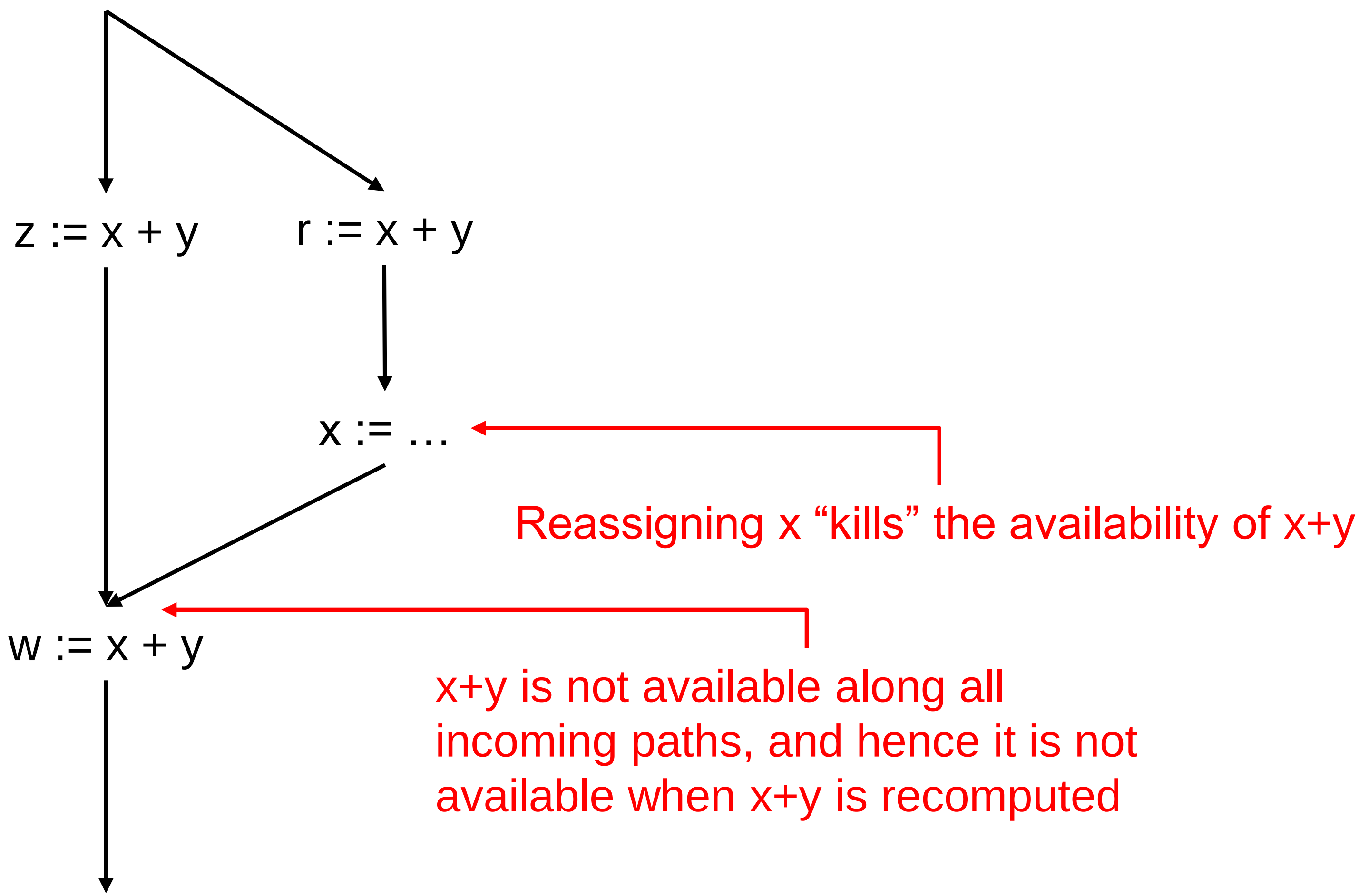
x+y is not available when the program starts

The computation of x+y "generates" the availability of x+y.

x+y is available along all incoming paths, and hence it is available before x+y is recomputed

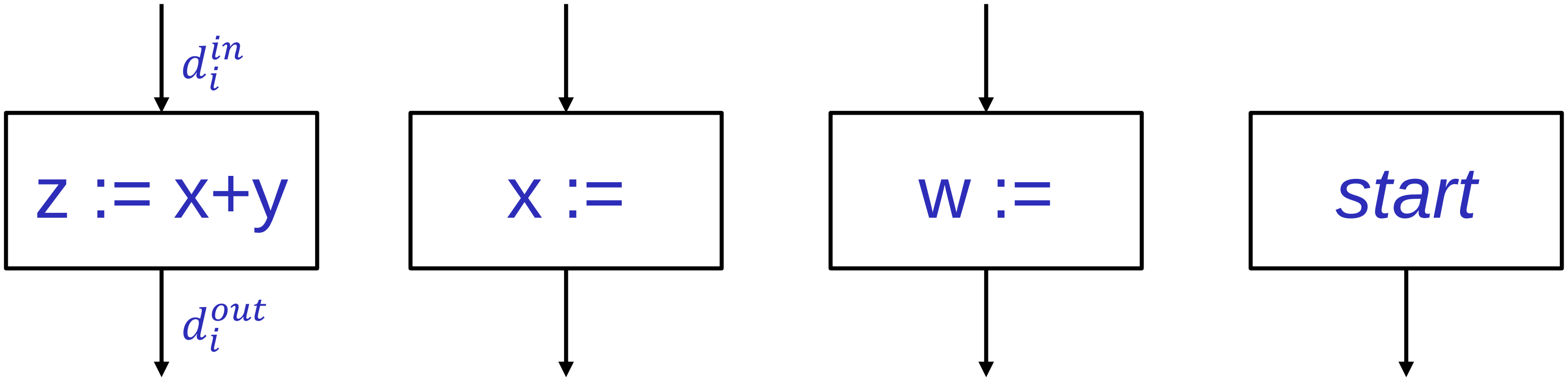
Data flow analysis

GEN statements generate facts. KILL statements remove them.



Rules for computing availability of x+y (1)

Create d_i^{in} and d_i^{out} to denote availability of x+y at the entry/exit of statement i

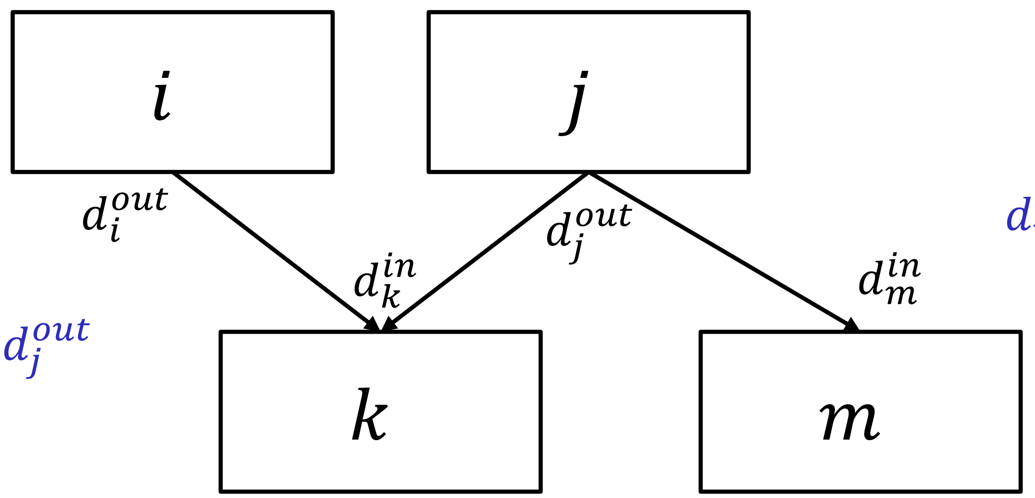


$d_i^{out} \leftarrow t$ $\quad$ $d_i^{out} \leftarrow f$ $\quad$ $d_i^{out} \leftarrow d_i^{in}$ $\quad$ $d_i^{out} \leftarrow f$

These are so-called *transfer functions*. The show how dataflow facts transfer across statements. The first is a GEN statement. The second is a KILL.

Rules for computing availability of x+y (2)

$d_m^{in} \leftarrow d_j^{out}$



$d_k^{in} \leftarrow d_i^{out} \wedge d_j^{out}$

This is so-called *control-flow merge function*. We use AND to ascertain that the property holds along all incoming paths.

Computing a solution to the constraints

Iterative algorithm:

1) Initialize all d values to t.

2) Pick any rule (e.g., $d_k^{in} \leftarrow d_i^{out} \wedge d_j^{out}$) and apply it.

3) Go to 2, stopping when no rule can change the value of any d.

The final solution is a *fixed point*,
i.e., all constraints are satisfied.

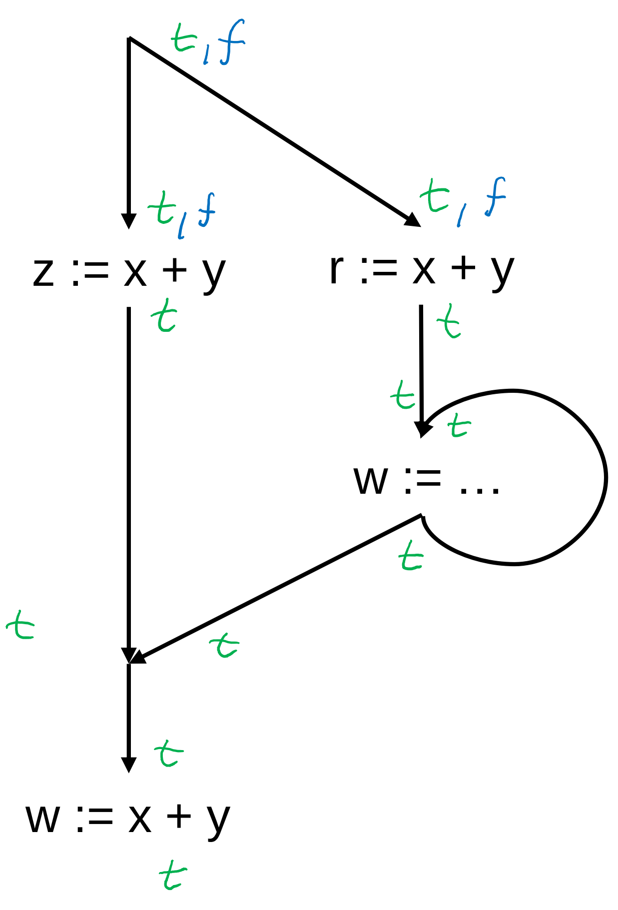
Since we initialized the values to t, the solution is the best fixed point, i.e. the most accurate solution.
The next two slides show why initializing to f yields a less accurate solution.

Example

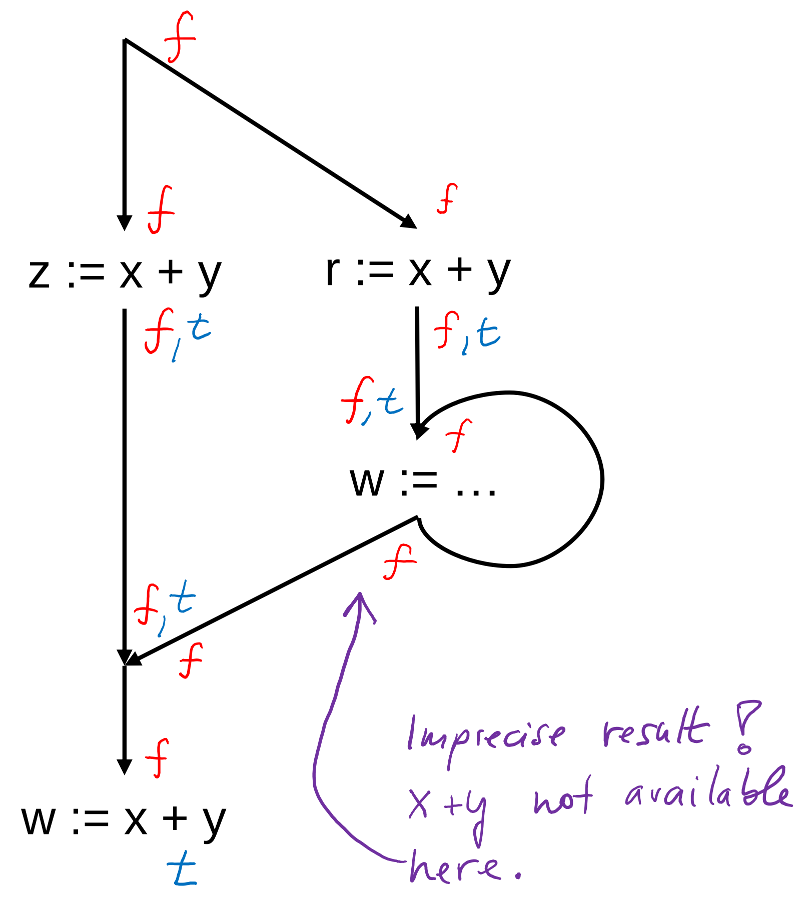
Green and red are initial values.

Blue values changed due to rule application

Initialize to true



Initialize to false



Flow analysis (of pointers)

Flow analysis (1): virtual call optimization

Optimization of virtual calls in Java:

virtual calls are costly, due to method dispatch

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.

How to analyze whether a call has this property?

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

   Foo a = …; a.f() // a could call Foo::f or Bar::f. Cant' tell from def of a

2. By analyzing what values flow to a=… .

   That is, we try to compute the dynamic type of a more precisely than is given by the definition "Foo a".

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);
```

Declared type of a permits a.foo() to call both A::foo and B::foo.

Yet we know only B::foo is the target, which allows optimization.

What program property would reveal that the optimization is possible?

Flow analysis (2): Verification of casts

In Java, casts are checked at run time

- type system not expressive enough to check them statically
- although Java generics help somewhat

The anatomy of a cast check: **(Foo) e** translates to

- **if (dynamic_type_of(e) not compatible with Foo)**

  throw ClassCast Exception
- t1 compatible with t2: t1 = t2 or t1 subclass of t2

Goal: prove that no exception will happen at runtime

- Why do this? The exception prevents any security holes, no?
- Such static verification useful to catch bugs (Mars Rover).

Example

```
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

What property will lead to desired verification?

What analysis that can serve these clients?

Is there a program property useful to these clients?

Yes.

We want to understand how references “flow”

References (pointer values): how are they copied from variable to variable?

Flow from **creation** of an object to its **uses**

that is, flow from new Foo to myFoo.f

Difference from dataflow analysis: pointer values may flow via the heap

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

Common Analysis

The flow analysis can be explained in terms of

- producers (creators of pointer values: new Foo)
- consumers (uses of the pointer value, e.g., 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 may 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 may not be called

Continued..

Client cast verification

Producers are new T

Consumers are cast expressions **(Type) p**.

Question: are casts also producers?

Yes, The cast generates new information! After the cast (T) p we know that the value in p is of type T or compatible with T. Otherwise the program would crash.

p = f() // assume f is declared to return Foo

t = (Bar) p // assume that Bar extends Foo

// we know that t must be of type Bar here.

Assume Java

For now, assume we're analyzing Java

- thanks to class defs, fields of objects are known statically
- (also, assume the Java program does not use reflection)

Flow analysis as a constant propagation

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?
```

Flow analysis as a constant propagation

We (conceptually) translate the program to

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

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

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 “$\mathtt{new}_7$ Foo”

We now consider pointer dereferences

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

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

Can y and w point to same object?

Keeping track of the heap state

Heap state: what objects a variable may point to at a particular program point.

Heap state may change at each statement

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

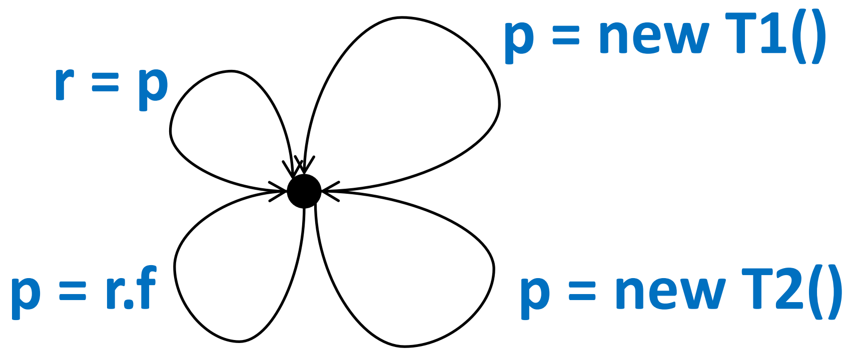
Disregards the control flow of the program

- 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 program 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

Let's develop the analysis! Canonical Stmts

Java pointers give rise to complex expressions:

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

Can we find a small set of canonical statements

- i.e., the core language understood by the analysis
- we'll desugar the rest of the program to these stmts

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

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 $\Leftrightarrow$ *o flowsTo x*

These two relations support our clients

These relations allows determining:

1. target methods of virtual calls
2. verification of casts
3. where objects are used

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

For 3) we need the *o* flowsTo *x* relation

Inference rule (1)

$p = new_i\ T()$ $\qquad$ $o_i\ new\ p$

$$o_i \text{ new } p \rightarrow o_i \textit{ flowsTo } p$$

Inference rule (2)

p = r $\qquad\qquad$ *r assign p*

$$o_i \ \mathit{flowsTo}\ r \wedge r\ \mathit{assign}\ p \rightarrow o_i\ \mathit{flowsTo}\ p$$

Inference rule (3)

p.f = a　　　　　*a pf(f) p*

b = r.f　　　　　*r gf(f) b*

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

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 $\longrightarrow$ 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).
```

How to use the result of the analysis?

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

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

The useful result is when the analysis ***can'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 better optimization, verification, compilation

Same arguments apply to alias, pointsTo relations

- and other static analyses in general

Example of inference

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

Inference Example (3), infering facts

o_1 new x

x assign w

z pf(f) y

o_2 new z

x assign 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

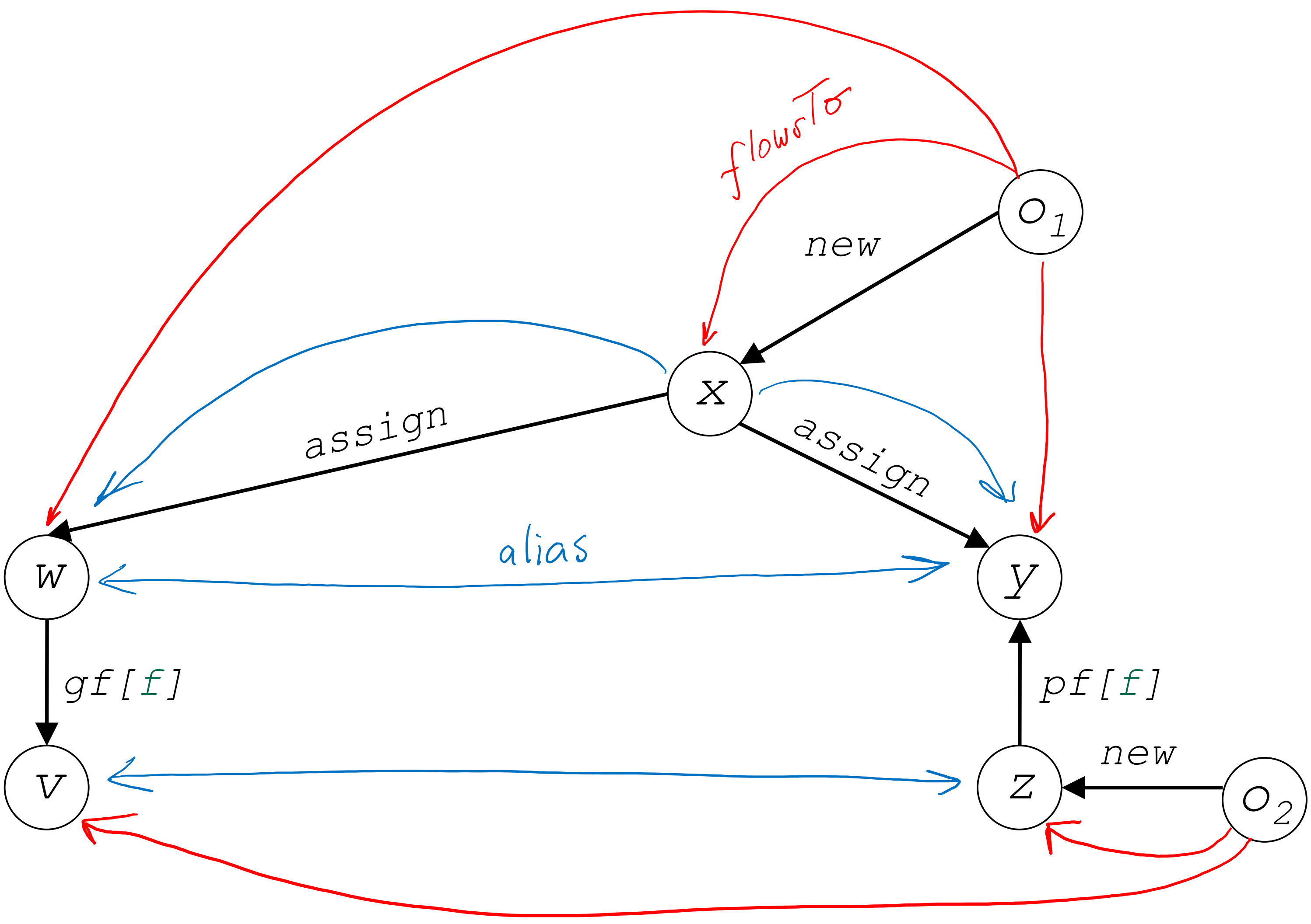
o_1 flowsTo x $\wedge$ x assign y $\rightarrow$ o_1 flowsTo y

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 analysis deductions



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

Extending the analysis for calls and arrays

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
r = new T; s = foo_retval; x = 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 targets are

Suggest two simple methods:

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

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

Summary of Static Analysis

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

“Parsing the graph”

Visualization of inferences on slides 41 and 42 parses the strings in the “graph of binary facts” using the CYK algorithm

The technique

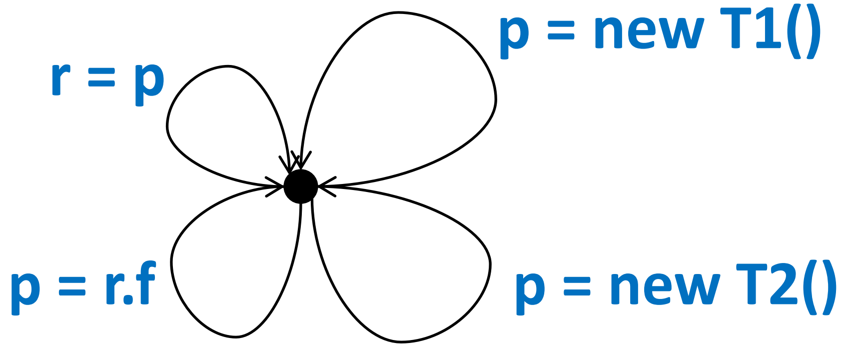
Flow-insensitive analysis:

- collapse into one all program points (ie, stmt entry and exits)
- reduces the amount of analysis state to maintain
- reduces precision, too, of course

Transform this program

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

into this one:



Andersen's algorithm

- Deduces the flowsTo relation from program statements
  - statements are facts
  - analysis is a set of inference rules
  - flowsTo relation is a set of facts inferred with analysis rules
- Statement facts: we'll write them as *x predicateName y*
  - $p = new_i\ T()$ $\quad$ o_i *new p*
  - p = r $\quad$ *r assign p*
  - p = r.f $\quad$ *r gf(f) p*
  - p.f = r $\quad$ *r pf(f) p*

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?

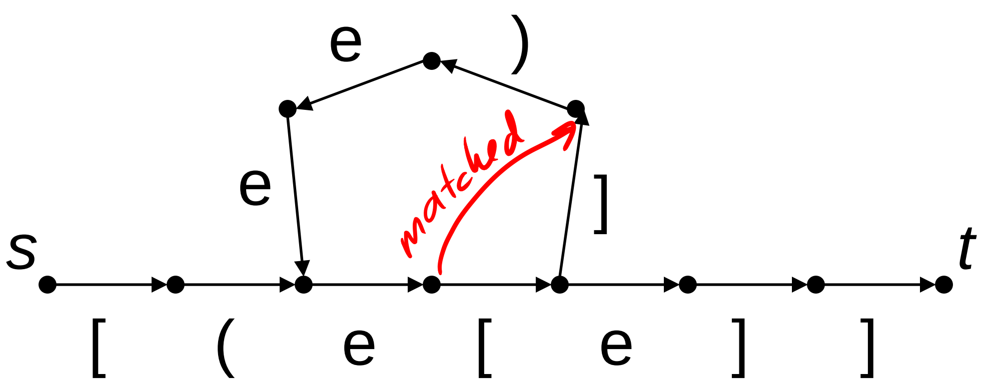
depth-first search, complexity O(N+E)

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

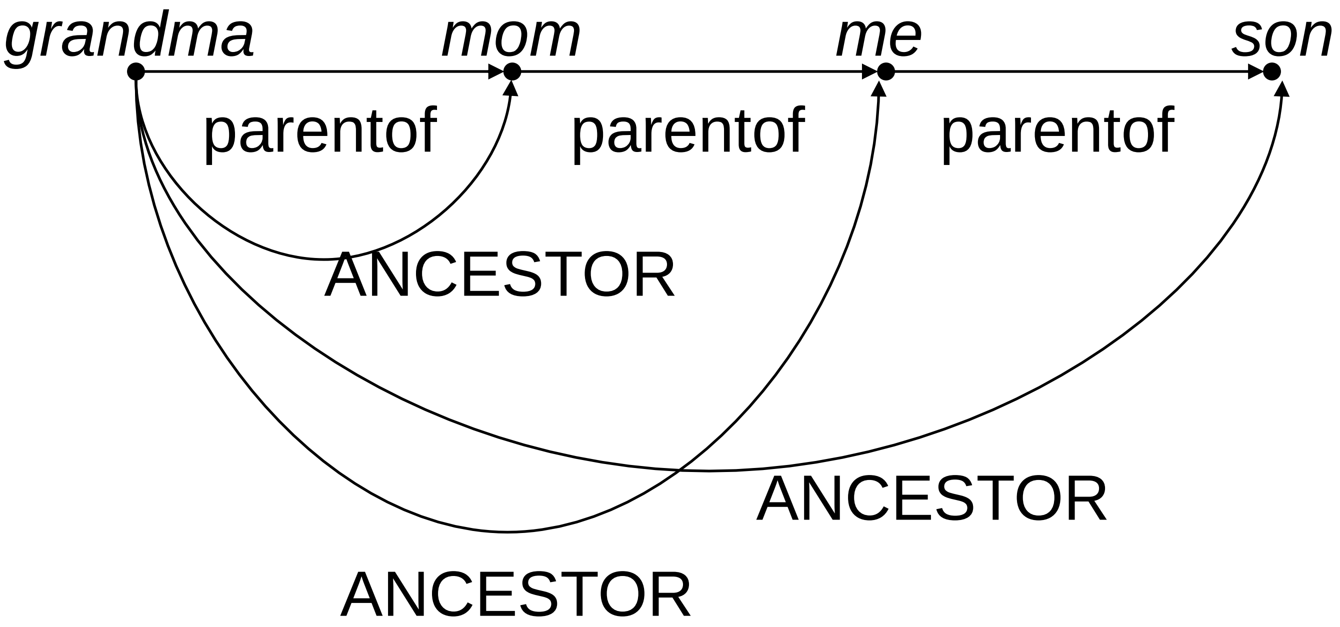
Notes:

- 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"?



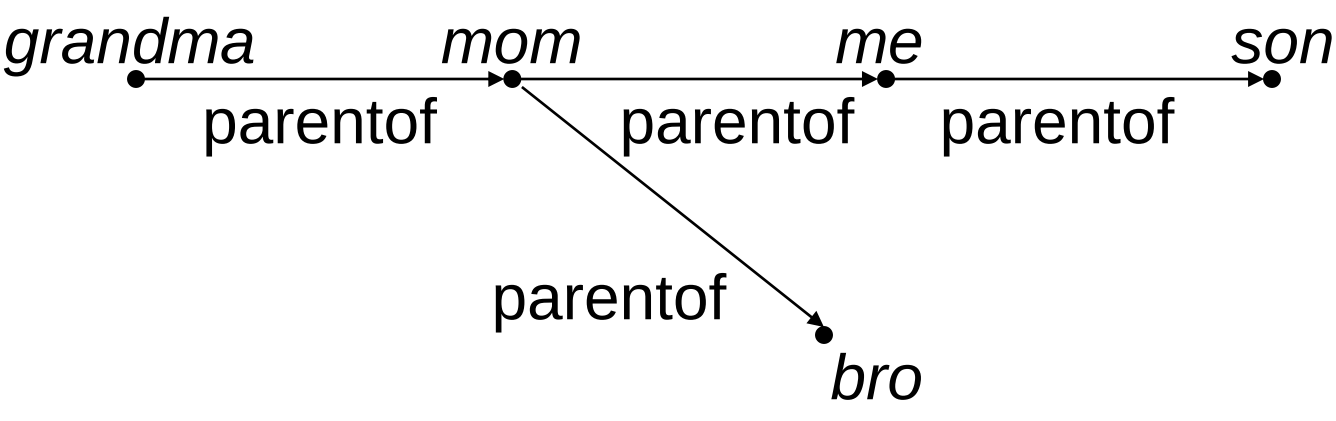
What rules can we convert to CFL-rechability?

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 $s \in L(\text{flowsTo})$

$\Leftrightarrow$

there exists a path from x to o labeled with $s' \in L(\text{pointsTo})$.

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

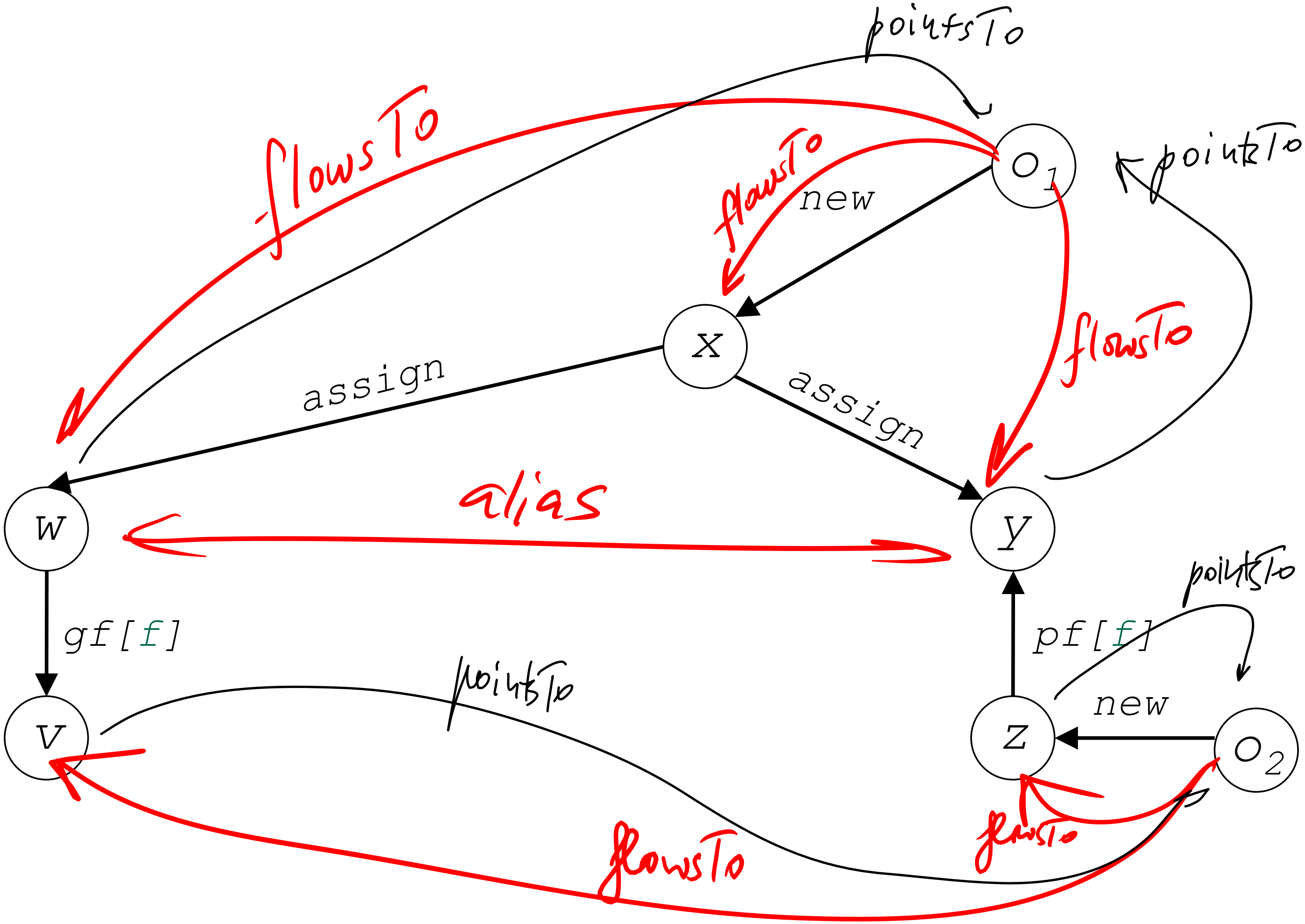
All rules are chain rules now

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

$$
\begin{aligned}
\text{flowsTo} &\rightarrow \text{new} \\
\text{flowsTo} &\rightarrow \text{flowsTo assign} \\
\text{flowsTo} &\rightarrow \text{flowsTo pf[f] alias gf[f]} \\
\text{pointsTo} &\rightarrow \text{new}^{-1} \\
\text{pointsTo} &\rightarrow \text{assign}^{-1} \text{ pointsTo} \\
\text{pointsTo} &\rightarrow \text{gf[f]}^{-1} \text{ alias pf[f]}^{-1} \text{ pointsTo} \\
\text{alias} &\rightarrow \text{pointsTo flowsTo}
\end{aligned}
$$

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