Static Program Analysis: What is it and how it is used

S. C. Kothari
CPRE 556:Lectures 9-10, 2006
Electrical and Computer Engineering Dept.
Iowa State University

Static analysis

- Without executing the program, answer questions such as:
  - does the variable x always have the same value?
  - will the value of x be read in the future?
  - can the pointer p be null?
  - which variables can p point to?
  - is the variable x initialized before it is read?
  - is the value of the integer variable x always positive?
  - what is a lower and upper bound on the value of the integer variable x?
  - at which program points could x be assigned its current value?
  - do p and q point to disjoint structures in the heap?

Describe all the errors

```
int main() {

        char *p,*q;
        p = NULL;
        printf("%s",p);
        q = (char *)malloc(100);
        p = q;
        free(q);
        *p = 'x';
        free(p);
        p = (char *)malloc(100);
        p = (char *)malloc(100);
        q = p;
        strcat(p,q);
}
```

The standard tools such as gcc -Wall and lint detect no errors.

Bad news ..

- **Rice's theorem** (also known as **The Rice-Myhill-Shapiro theorem - 1953**) is an important result in the theory of recursive functions. A property of partial functions is *trivial* if it holds for all partial recursive functions or for none. Rice's theorem states that, for any non-trivial property of partial functions, the question of whether a given algorithm computes a partial function with this property is undecidable. http://en.wikipedia.org/wiki/Rice's_theorem
- Implication of Rice's theorem: all interesting questions about the behavior of programs are *undecidable*.

Rice's theorem: an example

- Assume for example the existence of an analyzer that decides if a variable in a program has a constant value.
- This analyzer to also decide the halting problem by using as input the program:

  x = 17; if (TM(j)) x = 18;

- Here x has a constant value if and only if the j'th Turing machine halts on empty input.

Using static analysis as an engineering tool

- We want to solve practical problems like:
  - finding bugs in the program
  - identifying security holes
  - making the program run faster or use less space etc.
- The solution is to settle for *approximations* that are still precise enough to address our applications.
- Most often, such approximations are *conservative*, meaning that all errors lean to the same side.
- Engineering challenge: Be correct as often as possible while obtaining a reasonable performance.
- Also it may be able to define preconditions (to be satisfied by the program) under which we can argue that the analysis is in fact completely precise.

TIP: Tiny Imperative Programming Language

- The presentation uses a tiny imperative programming language, called *TIP*.
- It is designed to have a minimal syntax and yet to contain all the constructions that make static analyses interesting and challenging.

TIP: Expressions

The basic expressions all denote integer values:

$$
\begin{aligned}
E &\rightarrow \mathit{intconst} \\
&\rightarrow \mathit{id} \\
&\rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid E \texttt{ > } E \mid E \texttt{ == } E \\
&\rightarrow \texttt{(} E \texttt{)} \\
&\rightarrow \texttt{input}
\end{aligned}
$$

The `input` expression reads an integer from the input stream. The comparison operators yield 0 for false and 1 for true. Pointer expressions will be added later.

TIP: Statements

The simple statements are familiar:

$$
\begin{aligned}
S \rightarrow\ & id\ \texttt{=}\ E\ \texttt{;} \\
\rightarrow\ & \texttt{output}\ E\ \texttt{;} \\
\rightarrow\ & S\ S \\
\rightarrow\ & \texttt{if}\ \texttt{(}E\texttt{)}\ \texttt{\{}\ S\ \texttt{\}} \\
\rightarrow\ & \texttt{if}\ \texttt{(}E\texttt{)}\ \texttt{\{}\ S\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ S\ \texttt{\}} \\
\rightarrow\ & \texttt{while}\ \texttt{(}E\texttt{)}\ \texttt{\{}\ S\ \texttt{\}} \\
\rightarrow\ & \texttt{var}\ id_1, \ldots, id_n\texttt{;}
\end{aligned}
$$

In the conditions we interpret 0 as false and all other values as true. The `output` statement writes an integer value to the output stream. The `var` statement declares a collection of uninitialized variables.

TIP: Functions

Functions take any number of arguments and return a single value:

$$F \rightarrow id\ (\ id, \ldots, id\ )\ \{\ \texttt{var}\ id, \ldots, id;\ S\ \texttt{return}\ E;\ \}$$

Function calls are an extra kind of expression:

$$E \rightarrow id\ (\ E, \ldots, E\ )$$

TIP: Pointers

Finally, to allow dynamic memory, we introduce pointers into a heap:

$$
\begin{aligned}
E &\rightarrow \texttt{\&}id \\
&\rightarrow \texttt{malloc} \\
&\rightarrow \texttt{*}E \\
&\rightarrow \texttt{null}
\end{aligned}
$$

The first expression creates a pointer to a variable, the second expression allocates a new cell in the heap, and the third expression dereferences a pointer value. In order to assign values to heap cells we allow another form of assignment:

$$S \rightarrow \texttt{*}id \texttt{ = } E \texttt{;}$$

TIP: Function Pointers

Note that pointers and integers are distinct values, so pointer arithmetic is not permitted. It is of course limiting that `malloc` only allocates a single heap cell, but this is sufficient to illustrate the challenges that pointers impose.

We also allow function pointers to be denoted by function names. In order to use those, we generalize function calls to:

$$E \rightarrow (E)(\ E, \ldots, E\ )$$

Function pointers serve as a simple model for objects or higher-order functions.

TIP: Programs

A program is just a collection of functions:

$$P \rightarrow F \ldots F$$

The final function is the main one that initiates execution. Its arguments are supplied in sequence from the beginning of the input stream, and the value that it returns is appended to the output stream. We make the notationally simplifying assumption that all declared identifiers are unique in a program.

TIP: Program Examples

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

```
rec(n) {
  var f;
  if (n==0) { f=1; }
  else { f=n*rec(n-1); }
  return f;
}
```


Factorial program written in TIP

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = malloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

```
main() {
   var n;
   n = input;
   return foo(&n,foo);
}
```

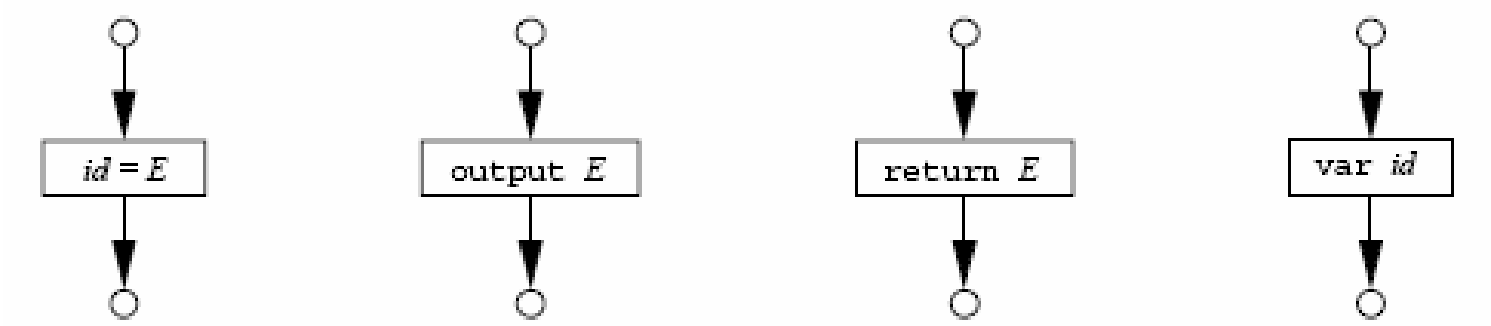
Flow Sensitive Analysis

- Type analysis starts with the syntax tree of a program and defines constraints over variables assigned to nodes.
- It is a *flow insensitive analysis*, in the sense that the results remain the same if a statement sequence *S*1*S*2 is permuted into *S*2*S*1.
- Analyses that are *flow sensitive* use a *control flow graph*, which is a different representation of the program source.

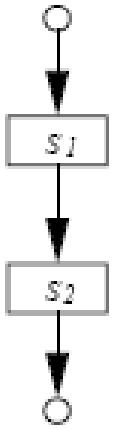
Control Flow Graphs

- A control flow graph (CFG) is a directed graph, in which *nodes* correspond to program points and *edges* represent possible flow of control.
- A CFG always has a single point of entry, denoted *entry*, and a single point of exit, denoted *exit*.
- If *v* is a node in a CFG then *pred*(*v*) denotes the set of predecessor nodes and *succ*(*v*) the set of successor nodes.

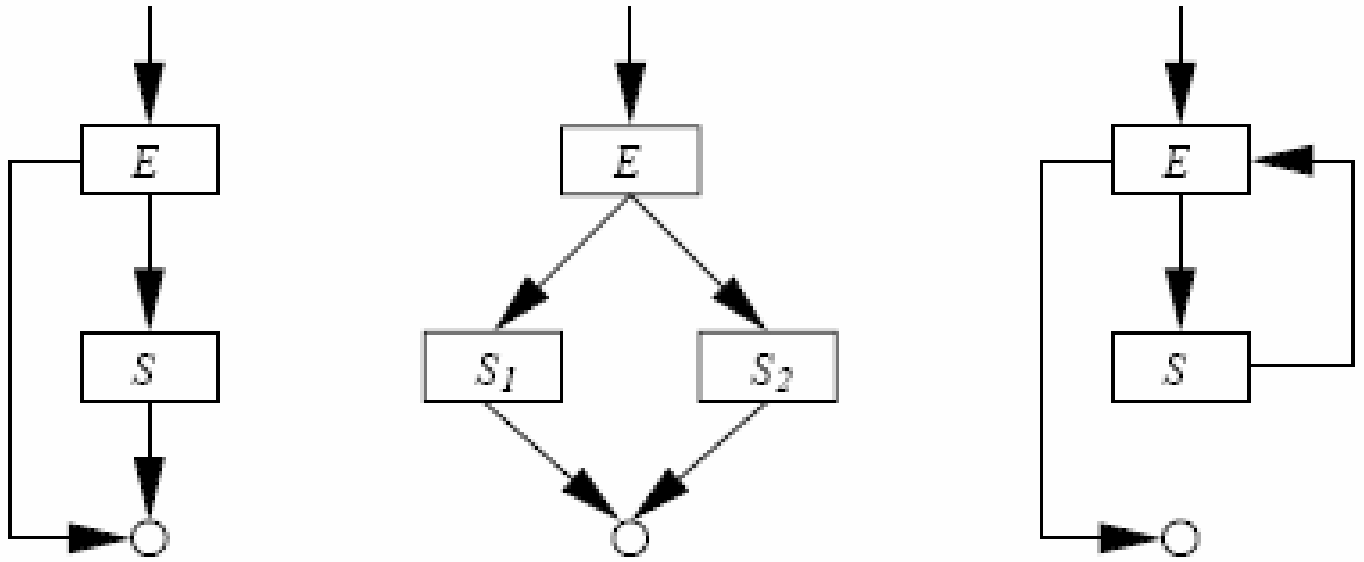
Control Flow Graphs for Statements



For the sequence S_1 S_2, we eliminate the exit node of S_1 and the entry node of S_2 and glue the statements together:

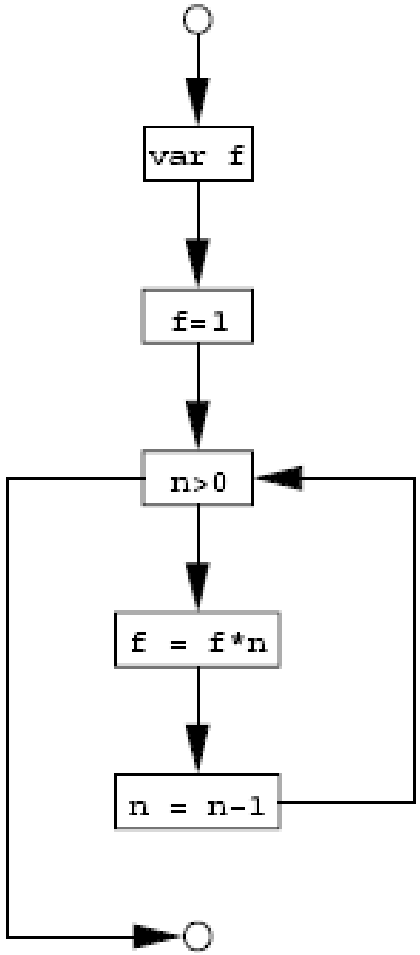


Control Flow Graphs for Control Structures



Example: Program and its CFG

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```



Dataflow Analysis

- To every node *v* in the CFG, we assign a set of variables [[*v*]].
- For each construction in the programming language, we then define a *dataflow constraint* that relates the value of the variable of the corresponding node to those of other nodes (typically the neighbors).
- Instead of a formal theory of dataflow analysis, we will discuss a set of interesting examples.

Live Variables

- A variable is *live* at a program point if its current value may be read during the remaining execution of the program.

Example: Liveness and its use

- y and z are never live at the same time, and that the value written in the assignment z=z-1 is never read. Thus, the program may safely be optimized.

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```



```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```


Analysis to Compute Live Variables

Reaching Definitions

- The *reaching definitions* for a given program point are those assignments that may have defined the current values of variables.

Notation

- For every CFG node *v,* let [[*v*]] denote the set set of assignments that may define values of variables at the program point *after* the node.
- Join(v) = Union of [[w]], w is a predecessor of v.
- If v is an assignment statement, let v(id) denote the ID of the variable assigned at v.

Computing the Reaching Definitions

- Constraint equation for assignment:

  [[*v*]] = Join(v) – { assignments-to v(id)} + {v}

- Constraint equation for other statements:

  [[*v*]] = Join(v)

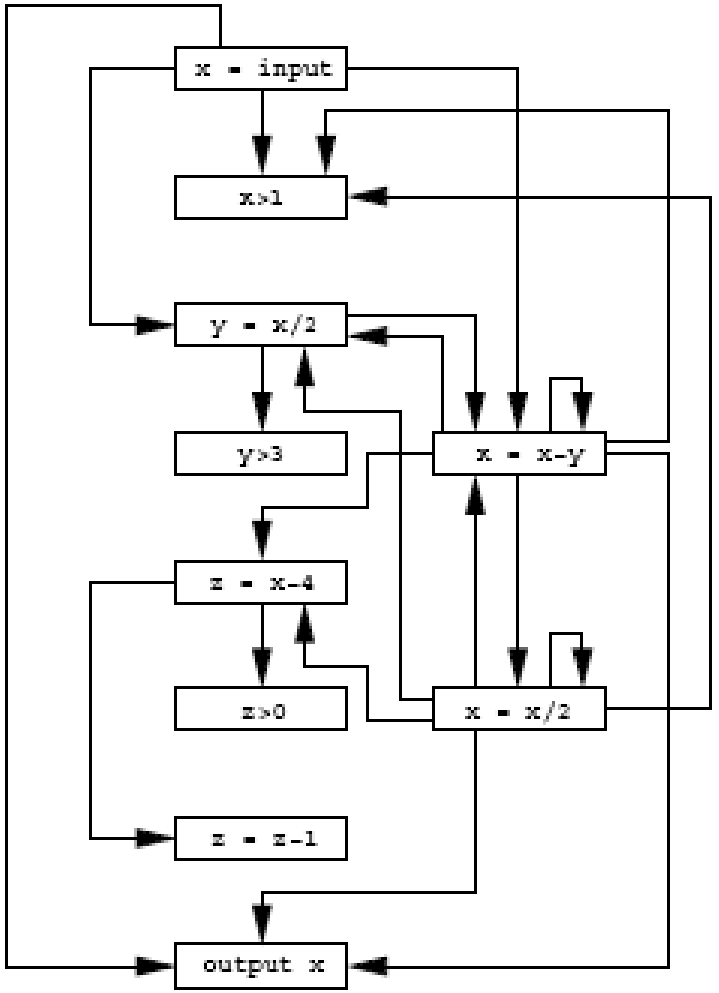
Def-use Graph

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

In a def-use graph, edges go
from definitions to possible uses.

Def-use Graph

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```



In a def-use graph, edges go from definitions to possible uses.

Analyses Types

- A *forwards* analysis is one that for each program point computes information about the *past* behavior.
  - Characterized by the right-hand sides of constraints only depending on *predecessors* of the CFG node.
- A *backwards* analysis is one that for each program point computes information about the *future* behavior.
  - Characterized by the right-hand sides of constraints only depending on *successors* of the CFG node

Analyses Types

- A *may* analysis is one that describes information that may possibly be true and, thus, computes an *upper* approximation
  - Characterized by the right-hand sides of constraints using a *union* operator to combine information.
- A *must* analysis is one that describes information that must definitely be true and, thus, computes a *lower* approximation.
  - Characterized by the right-hand sides of constraints using a *intersection* operator to combine information.

Pointer Analysis

- Pointer manipulations:
  - *id* = malloc
  - *id* 1 = &*id 2*
  - *id* 1 = *id 2*
  - *id* 1 = **id 2*
  - **id* 1 = *id 2*
  - *id* = null

Pointer Constraints

id = malloc:	$\{\texttt{malloc-i}\} \subseteq [\![id]\!]$
$id_1 = \&id_2$:	$\{\&id_2\} \subseteq [\![id_1]\!]$
$id_1 = id_2$:	$[\![id_2]\!] \subseteq [\![id_1]\!]$
$id_1 = *id_2$:	$\&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!]$
$*id_1 = id_2$:	$\&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!]$

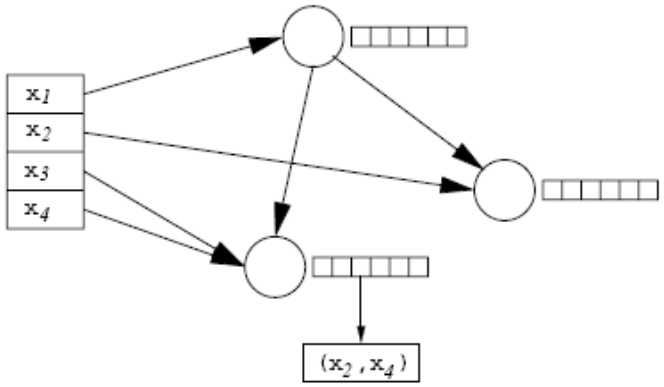
The last two constraints are generated for every variable named *id*, but we need in fact only consider those whose addresses are actually taken in the given program. The null assignment corresponds to the constraint:

$$\emptyset \subseteq [\![id]\!]$$

Cubic Algorithm

We have a set of *tokens* $\{t_1, \ldots, t_k\}$ and a collection of *variables* $x_1, \ldots, x_n$ whose values are subsets of token. Our task is to read a sequence of *constraints* of the form $\{t\} \subseteq x$ or $t \in x \Rightarrow y \subseteq z$ and produce the minimal solution.

The algorithm is based on a simple data structure. Each variable is mapped to a node in a directed acyclic graph (DAG). Each node has an associated bitvector belonging to $\{0,1\}^k$, initially defined to be all 0's. Each bit has an associated list of pairs of variables, which is used to model conditional constraints. The edges in the DAG reflect inclusion constraints. The bitvectors will at all times directly represent the minimal solution. An example graph may look like:



Constraints are added one at a time. A constraint of the form $\{t\} \subseteq x$ is handled by looking up the node associated with x and setting the corresponding bit to 1. If its list of pairs was not empty, then an edge between the nodes corresponding to y and z is added for every pair (y, z). A constraint of the form $t \in x \Rightarrow y \subseteq z$ is handled by first testing if the bit corresponding to t in the node corresponding to x has value 1. If this is so, then an edge between the nodes corresponding to y and z is added. Otherwise, the pair (y, z) is added to the list for that bit.

If a newly added edge forms a cycle, then all nodes on that cycle are merged into a single node, which implies that their bitvectors are unioned together and their pair lists are concatenated. The map from variables to nodes is updated accordingly. In any case, to reestablish all inclusion relations we must propagate the values of each newly set bit along all edges in the graph.

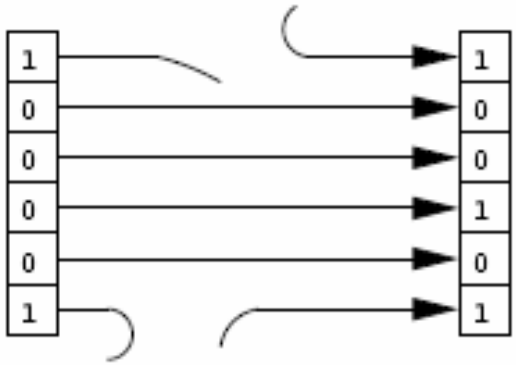
Analysis of the algorithm

To analyze this algorithm, we assume that the numbers of tokens and constraints are both $O(n)$. This is clearly the case when analyzing programs, where the numbers of variables, tokens, and constraints all are linear in the size of the program.

Merging DAG nodes on cycles can be done at most $O(n)$ times. Each merger involves at most $O(n)$ nodes and the union of their bitvectors is computed in time at most $O(n^2)$. The total for this part is $O(n^3)$.

New edges are inserted at most $O(n^2)$ times. Constant sets are included at most $O(n^2)$ times, once for each $\{t\} \subseteq x$ constraint.

Finally, to limit the cost of propagating bits along edges, we imagine that each pair of corresponding bits along an edge are connected by a tiny bitwire. Whenever the source bit is set to 1, that value is propagated along the bitwire which then is broken:



Since we have at most n^3 bitwires, the total cost for propagation is $O(n^3)$.

Pointer Constraints

id = malloc:	$\{\texttt{malloc-i}\} \subseteq [\![id]\!]$
id_1 = &id_2:	$\{\&id_2\} \subseteq [\![id_1]\!]$
id_1 = id_2:	$[\![id_2]\!] \subseteq [\![id_1]\!]$
id_1 = *id_2:	$\&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!]$
*id_1 = id_2:	$\&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!]$

These constraints match the requirements of the cubic algorithm. The resulting points-to function is defined as: *pt*(*p*) = [[*p*]].

The end result of a points-to analysis is a function *pt* that for each (pointer) variable *p* returns the set *pt*(*p*) of possible pointer targets to which it may evaluate.

Example

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

constraints

$$\texttt{malloc-1} \subseteq [\![\texttt{p}]\!]$$
$$[\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$$
$$[\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!]$$
$$\texttt{\&y} \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{y}]\!]$$
$$[\![\texttt{q}]\!] \subseteq [\![\texttt{p}]\!]$$
$$\{\texttt{\&y}\} \subseteq [\![\texttt{q}]\!]$$
$$\texttt{\&y} \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$$
$$\{\texttt{\&z}\} \subseteq [\![\texttt{p}]\!]$$

solution

$$pt(\texttt{p}) = [\![\texttt{p}]\!] = \{\texttt{malloc-1}, \texttt{\&y}, \texttt{\&z}\}$$
$$pt(\texttt{q}) = [\![\texttt{q}]\!] = \{\texttt{\&y}\}$$

This is so called Anderson's algorithm. Note that while this algorithm is flow insensitive, the directionality of the constraints implies that the dataflow is still modeled with some accuracy.

Steensgaard's Algorithm

- Performs a coarser analysis essentially by viewing assignments as being bidirectional.
- Uses a set consisting of the malloc-I tokens and two tokens of the form *id* and **id* for each variable named *id*.

Constraints

id = malloc:	$*id \sim$ malloc-i
$id_1 = \&id_2$:	$*id_1 \sim id_2$
$id_1 = id_2$:	$id_1 \sim id_2$
$id_1 = *id_2$:	$id_1 \sim *id_2$
$*id_1 = id_2$:	$*id_1 \sim id_2$

The generated constraints induce an equivalence relation on the tokens, which can be computed in almost linear time. The resulting points-to function is defined as:

$$pt(p) = \{\&id \mid *p \sim id\} \cup \{\texttt{malloc-i} \mid *p \sim \texttt{malloc-i}\}$$

Example

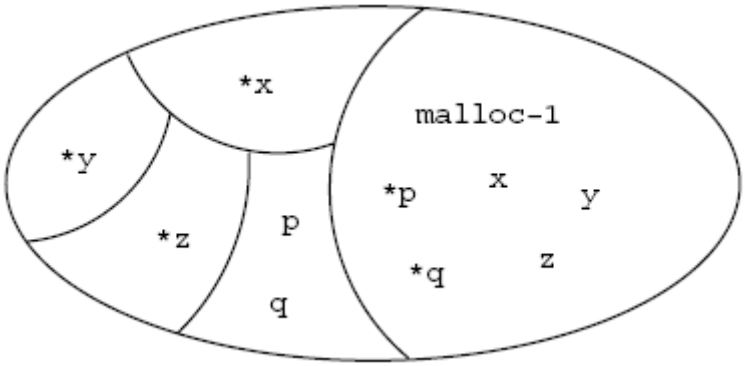
```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Steensgaard's constraints →

$*p \sim \texttt{malloc-1}$
$x \sim y$
$x \sim z$
$*p \sim z$

$p \sim q$
$\boxed{*p \sim y}$
$x \sim *p$
$*p \sim z$

↓ Equivalence classes



← solution

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \texttt{\&x}, \texttt{\&y}, \texttt{\&z}\}$$

↓ Restricting to the addresses that are actually taken in the program

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \texttt{\&y}, \texttt{\&z}\}$$

Null Pointer Analysis

For every CFG node *v* we introduce a constraint variable [[*v*]] denoting a symbol table giving the status for every variable at that program point.
For variable declarations:

$$[\![v]\!] = [id_1 \mapsto ?, \ldots, id_n \mapsto ?]$$

For the nodes corresponding to the various pointer manipulations we have the constraints:

$$\begin{array}{rll}
id \texttt{ = malloc}: & [\![v]\!] & = JOIN(v)[id \mapsto \texttt{NN}] \\
id_1 \texttt{ = \&}id_2: & [\![v]\!] & = JOIN(v)[id_1 \mapsto \texttt{NN}] \\
id_1 \texttt{ = } id_2: & [\![v]\!] & = JOIN(v)[id_1 \mapsto JOIN(v)(id_2)] \\
id_1 \texttt{ = *}id_2: & [\![v]\!] & = right(JOIN(v), id_1, id_2) \\
\texttt{*}id_1 \texttt{ = } id_2: & [\![v]\!] & = left(JOIN(v), id_1, id_2) \\
id \texttt{ = null}: & [\![v]\!] & = JOIN(v)[id \mapsto \texttt{IN}]
\end{array}$$

and for all other nodes the constraint:

$$[\![v]\!] = JOIN(v)$$

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

$$right(\sigma, x, y) = \sigma[x \mapsto \sigma(y) \sqcup \bigsqcup_{\&p \in pt(y)} \sigma(p)]$$

$$left(\sigma, x, y) = \sigma \quad [\, p \mapsto \sigma(p) \sqcup \sigma(y) \,]_{\&p \in pt(x)}$$

Note that allocation sites will always be mapped to $\perp$, which reflects that we are not tracking cardinality or connectivity of the heap. After the analysis, the evaluation of `*p` is guaranteed to be safe at program point v if $[\![v]\!](p) = \texttt{NN}$. The precision of this analysis depends of course on the quality of the underlying points-to analysis.

where `IN` means *initialized* and `NN` means *not* `null`.

Example

```
var p,q,r,n;
p = malloc;
q = &p;
n = null;
*q = n;
*p = r;
```

Anderson's algorithm →

$pt(\mathtt{p}) = \{\mathtt{malloc\text{-}1}\}$
$pt(\mathtt{q}) = \{\&\mathtt{p}\}$
$pt(\mathtt{r}) = \{\}$
$pt(\mathtt{n}) = \{\}$

↓ Null constraints

$[\![\mathtt{var\ p,q,r,n;}]\!] = [\mathtt{p} \mapsto ?, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{p=\&p}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{q=\&p}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{n=null}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*q=n}]\!] = [\mathtt{p} \mapsto \mathtt{IN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*p=r}]\!] = [\mathtt{p} \mapsto \mathtt{IN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$

← solution

$[\![\mathtt{var\ p,q,r,n;}]\!] = [\mathtt{p} \mapsto ?, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{p=malloc}]\!] = [\mathtt{var\ p,q,r,n;}][\mathtt{p} \mapsto \mathtt{NN}]$
$[\![\mathtt{q=\&p}]\!] = [\mathtt{p=malloc}][\mathtt{q} \mapsto \mathtt{NN}]$
$[\![\mathtt{n=null}]\!] = [\mathtt{q=\&p}][\mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*q=n}]\!] = [\mathtt{n=null}][\mathtt{p} \mapsto [\![\mathtt{n=null}]\!](\mathtt{p}) \sqcup [\![\mathtt{n=null}]\!](\mathtt{n})]$
$[\![\mathtt{*p=r}]\!] = [\mathtt{*q=n}]$

By inspecting this information, a compiler could statically detect that when `*p=r` is evaluated, the variable `p` may contain `null` and the variable `r` may be uninitialized.

