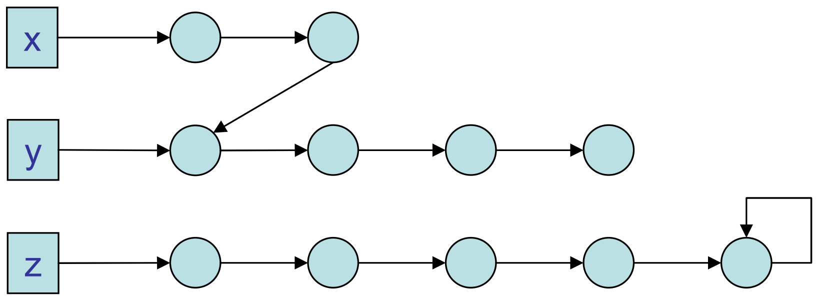
Pointer Analysis

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

Heap Pointers

- Pointers in the TIP language are limited:
  - `malloc` only allocates a single cell
  - only linear structures can be built in the heap



- But we still have all relevant analysis challenges...

Pointer Targets

- The fundamental question about pointers: What are their possible targets?
- We need a suitable abstraction:
  - `&`*id* for a program variable named *id*
  - `malloc-`*i* for an allocation site with index *i*
- The set of all these is denoted *Targets*
- Each target may correspond to many actual memory cells at runtime

Points-To Analysis

- Determine for each pointer variable `p` the set, *pt*(`p`), of its possible targets
- A *conservative* analysis:
  - the set may be too large
  - the trivial answer is *pt*(`p`) = *Targets*
  - can e.g. eliminate aliases: $pt(\mathtt{p}) \cap pt(\mathtt{q}) = \emptyset$
- A *flow-insensitive* analysis:
  - takes place on the AST
  - before or together with the control-flow analysis

Obtaining Points-To Information

- The simplest non-trivial analysis:
  - include all `malloc-`*i* targets
  - include `&`*id* if that expression occurs in the program
  - this is called *address-taken*
- Improvement for a typed language:
  - eliminate those targets whose types do not match
- Amazingly, this is sometimes good enough
  - and clearly very fast to compute

Pointer Normalization

- Assume that all pointer usage is normalized:
  - id = `malloc`
  - id_1 = &id_2
  - id_1 = id_2
  - id_1 = *id_2
  - *id_1 = id_2
  - id = `null`

- Simply introduce lots of temporary variables
- All subexpressions are now named

Andersen's Analysis (1/2)

- For every program variable, v, introduce a variable [[v]] ranging over *Targets*

- Generate constraints:
  - id = `malloc`: { `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]]

Andersen's Analysis (2/2)

- The points-to map is defined as: $pt(\mathtt{p}) = [[\mathtt{p}]]$

- The constraints fit into the cubic framework
- Unique minimal solution in time $O(n^3)$

- The analysis is flow-insensitive but *directional*
  - we know which way values flow in assignments

Example Program

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

Applying Andersen

- Generated constraints:

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

- Smallest solution:

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

Steensgaard's Analysis (1/2)

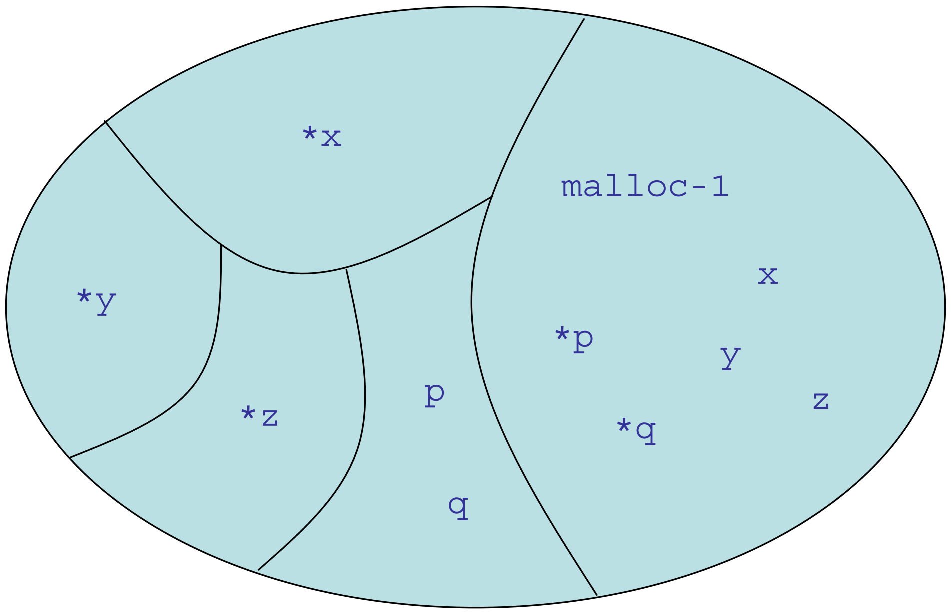
- View assignments as being bidirectional
- Introduce tokens:
  - `malloc-`*i*
  - *id* and `*`*id* for each variable *id*
- Define a relation on these tokens
- Compute smallest enclosing equivalence, $\sim$
  - this can be done in time $O(n\alpha(n))$

Steensgaard's Analysis (2/2)

- Generate constraints:
  - id = `malloc`: $*id \sim$ `malloc`-i
  - $id_1 = \&id_2$: $*id_1 = id_2$
  - $id_1 = id_2$: $id_1 \sim id_2$
  - $id_1 = *id_2$: $id_1 \sim *id_2$
  - $*id_1 = id_2$: $*id_1 \sim id_2$
- The points-to map is defined as:
  - $pt(\mathtt{p}) = \{\&id \mid *\mathtt{p} \sim id\} \cup \{\, \mathtt{malloc}\text{-}i \mid *\mathtt{p} \sim \mathtt{malloc}\text{-}i \,\}$

Applying Steensgaard

```
*p ~ malloc-1
p ~ q
x ~ y
*p ~ y
x ~ z
x ~ *p
*p ~ z
*p ~ z
```



- *pt*(`p`) = *pt*(`q`) = { `malloc-1`, `&x`, `&y`, `&z` }
- Intersecting with address-taken eliminates `&x`

Interprocedural Points-To Analysis

- If function pointers are distinct from heap pointers:
  - first run a CFA
  - then run Andersen or Steensgaard

- But both pointers may be mixed together:

```
(***x)(1,2,3)
```

- In this case the CFA and the points-to analysis must happen simultaneously

Function Call Normalization

- Assume that all function calls are of the form:

  $id_1 = id_2(a_1, \ldots, a_n)$

- Assume that all return statements are of the form:

  `return` *id*`;`

- Simply introduce lots of temporary variables

CFA with Andersen

- For the function call and every occurrence of:

  $f\,(x_1,\ \ldots,\ x_n)\ \{\ \ldots\ \mathtt{return}\ id\,;\ \}$

  add the constraints:

  $\{\ \&f\ \} \subseteq [[f]]$
  $\&f \in [[id_2]] \Rightarrow [[a_i]] \subseteq [[x_i]] \wedge [[id]] \subseteq [[id_1]]$

- Solve the constraints using the cubic framework

CFA with Steensgaard

- Always add the constraints:

$$a_i \sim x_i \wedge id \sim id_1$$

- Very imprecise, since any *n*-argument function is assumed to be a possible target for the call

NULL Pointer Analysis

- Decide for every dereference `*p`:
  - has `p` been initialized?
  - is `p` different from `null`?

- Use the monotone framework
  - assuming that a points-to map has been computed

A Lattice for NULL Analysis

- Define the simple lattice *Null*:

```
 ?
 |
IN
 |
NN
 |
 ⊥
```

  where `IN` is *initialized* and `NN` is *not null*

- Use for every program point the map lattice:

  $$\mathit{Vars} \rightarrow \mathit{Null}$$

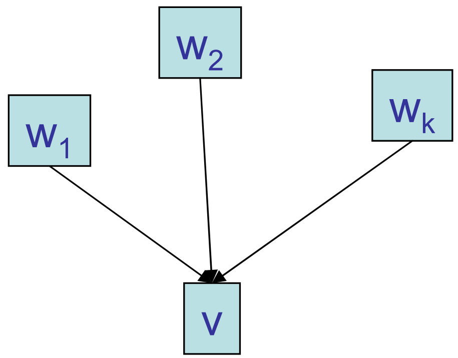
  where *Vars* are the declared variables

Setting Up

- For every CFG node, v, we have a variable [[v]]:
  - a map giving the NULL status for all program variables at the program point *after* v

- Auxiliary definition:

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



NULL Constraints

- For variable declarations:
  - $[[v]] = [id_1 \rightarrow ?, \ldots, id_n \rightarrow ?]$
- For pointer operations:
  - id = `malloc`: $[[v]] = JOIN(v)[id \rightarrow \texttt{NN}]$
  - id_1 = &id_2: $[[v]] = JOIN(v)[id_1 \rightarrow \texttt{NN}]$
  - $id_1 = id_2$: $[[v]] = JOIN(v)[id_1 \rightarrow JOIN(v)(id_2)]$
  - $id_1 = {*}id_2$: $[[v]] = right(JOIN(v), id_1, id_2)$
  - ${*}id_1 = id_2$: $[[v]] = left(JOIN(v), id_1, id_2)$
  - id = `null`: $[[v]] = JOIN(v)[id \rightarrow \texttt{IN}]$
- For all other CFG nodes:
  - $[[v]] = JOIN(v)$

Auxiliary Functions

- $x = *y$:

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

- $*x = y$

  $$left(\sigma,x,y) = \sigma_{\&p \in pt(x)}[p \to \sigma(p) \sqcup \sigma(y)]$$

- *Strong* update: $\sigma[x \to change]$
- *Weak* update: $\sigma[x \to \sigma(x) \sqcup change]$

Using the NULL Analysis

- The pointer dereference `*p` is safe at v if:

$$\left(\bigsqcup_{w \in \mathit{pred}(v)} [[w]] \right)(\mathtt{p}) = \mathtt{NN}$$

- The quality of the NULL analysis depends on the quality of the underlying points-to analysis

Example Program

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

- Andersen generates:

  $pt(\mathtt{q}) = \{\mathtt{malloc\text{-}1}\}$

  $pt(\mathtt{p}) = \{\mathtt{\&q}\}$

  $pt(\mathtt{r}) = pt(\mathtt{n}) = \{\}$

Generated Constraints

$[\![\texttt{var p,q,r,n}]\!] = [\texttt{p} \to ?, \texttt{q} \to ?, \texttt{r} \to ?, \texttt{n} \to ?]$

$[\![\texttt{p=malloc}]\!] = [\![\texttt{var p,q,r,n}]\!][\texttt{p} \to \mathrm{NN}]$

$[\![\texttt{q=\&p}]\!] = [\![\texttt{p=malloc}]\!][\texttt{q} \to \mathrm{NN}]$

$[\![\texttt{n=null}]\!] = [\![\texttt{q=\&p}]\!][\texttt{n} \to \mathrm{IN}]$

$[\![\texttt{*q=n}]\!] = [\![\texttt{n=null}]\!][\texttt{p} \to [\![\texttt{n=null}]\!](\texttt{p}) \sqcup [\![\texttt{n=null}]\!](\texttt{n})]$

$[\![\texttt{*p=r}]\!] = [\![\texttt{*q=n}]\!]$

Solution

$[\![\texttt{var p,q,r,n}]\!] = [\texttt{p} \to \texttt{?}, \texttt{q} \to \texttt{?}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{?}]$

$[\![\texttt{p=malloc}]\!] = [\texttt{p} \to \texttt{NN}, \texttt{q} \to \texttt{?}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{?}]$

$[\![\texttt{q=\&p}]\!] = [\texttt{p} \to \texttt{NN}, \texttt{q} \to \texttt{NN}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{?}]$

$[\![\texttt{n=null}]\!] = [\texttt{p} \to \texttt{NN}, \texttt{q} \to \texttt{NN}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{IN}]$

$[\![\texttt{*q=n}]\!] = [\texttt{p} \to \texttt{IN}, \texttt{q} \to \texttt{NN}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{IN}]$

$[\![\texttt{*p=r}]\!] = [\texttt{p} \to \texttt{IN}, \texttt{q} \to \texttt{NN}, \texttt{r} \to \texttt{?}, \texttt{n} \to \texttt{IN}]$

- For the statement `*p=r` the compiler now knows:
  - `p` may contain NULL
  - `r` may be uninitialized