Static Program Analysis

Part 10 – pointer analysis

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University

Agenda

- **Introduction to pointer analysis**
- Andersen's analysis
- Steensgaard's analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis

Analyzing programs with pointers

How do we perform e.g. constant propagation analysis when the programming language has pointers?
(or object references?)

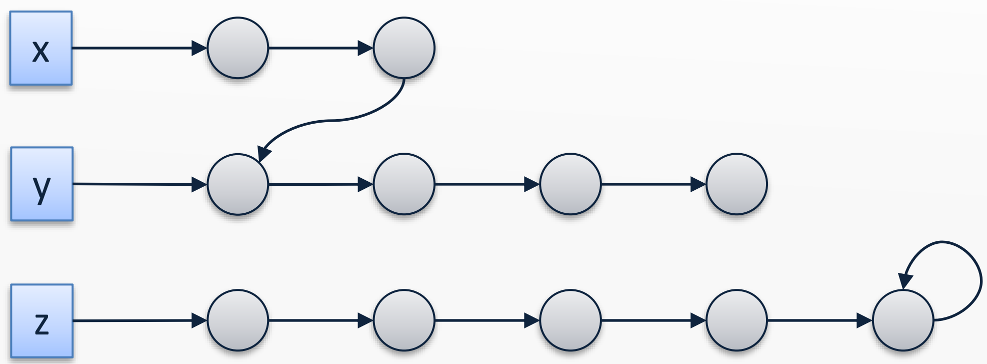
$Exp \rightarrow \ldots$
$\quad |$ `alloc` E
$\quad |$ $\&Id$
$\quad |$ $*Exp$
$\quad |$ `null`

$Stm \rightarrow \ldots$
$\quad |$ $*Id = Exp$ `;`

```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

Heap pointers

- For simplicity, we ignore records
  - `alloc` then only allocates a single cell
  - only linear structures can be built in the heap



- Let's at first also ignore functions as values
- We still have many interesting analysis challenges...

Pointer targets

- The fundamental question about pointers:
  What cells can they point to?

- We need a suitable abstraction
- The set of (abstract) cells, *Cells*, contains
  - `alloc-`*i* for each allocation site with index *i*
  - *X* for each program variable named *X*
- This is called ***allocation site abstraction***
- Each abstract cell may correspond to many concrete memory cells at runtime

Points-to analysis

- Determine for each pointer variable *X* the set $pt(X)$ of the cells *X* may point to

```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

- A *conservative* ("may points-to") analysis:
  - the set may be too large
  - can show absence of aliasing: $pt(X) \cap pt(Y) = \emptyset$
- We'll focus on *flow-insensitive* analyses:
  - take place on the AST
  - before or together with the control-flow analysis

Obtaining points-to information

- An almost-trivial analysis (called *address-taken*):
  - include all `alloc-`*i* cells
  - include the *X* cell if the expression &*X* occurs in the program

- Improvement for a typed language:
  - eliminate those cells whose types do not match

- This is sometimes good enough
  - and clearly very fast to compute

Pointer normalization

- Assume that all pointer usage is normalized:
  - $X = \texttt{alloc}\ P$ where P is `null` or an integer constant
  - $X = \&Y$
  - $X = Y$
  - $X = *Y$
  - $*X = Y$
  - $X = \texttt{null}$
- Simply introduce lots of temporary variables…
- All sub-expressions are now named
- We choose to ignore the fact that the cells created at variable declarations are uninitialized (otherwise it is impossible to get useful results from a flow-insensitive analysis)

Agenda

- Introduction to pointer analysis
- **Andersen's analysis**
- Steensgaard's analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis

Andersen's analysis (1/2)

- For every cell *c*, introduce a constraint variable $[\![c]\!]$ ranging over sets of cells, i.e. $[\![\cdot]\!]: \mathit{Cells} \rightarrow \mathcal{P}(\mathit{Cells})$

- Generate constraints:
  - $X = \texttt{alloc}\ P$: $\texttt{alloc-}i \in [\![X]\!]$
  - $X = \&Y$: $Y \in [\![X]\!]$
  - $X = Y$: $[\![Y]\!] \subseteq [\![X]\!]$
  - $X = {*}Y$: $c \in [\![Y]\!] \Rightarrow [\![c]\!] \subseteq [\![X]\!]$ for each $c \in \mathit{Cells}$
  - ${*}X = Y$: $c \in [\![X]\!] \Rightarrow [\![Y]\!] \subseteq [\![c]\!]$ for each $c \in \mathit{Cells}$
  - $X = \texttt{null}$: (no constraints)

(For the conditional constraints, there's no need to add a constraint for the cell x if &x does not occur in the program)

Andersen's analysis (2/2)

- The points-to map is defined as:

  $$pt(X) = [\![X]\!]$$

- The constraints fit into the cubic framework ☺
- Unique minimal solution in time $O(n^3)$
- In practice, for Java: $O(n^2)$

- The analysis is flow-insensitive but *directional*
  - models the direction of the flow of values in assignments

Example program

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

Cells = {p, q, x, y, z, `alloc-1`}

Applying Andersen

- Generated constraints:

$\texttt{alloc-1} \in [\![p]\!]$
$[\![y]\!] \subseteq [\![x]\!]$
$[\![z]\!] \subseteq [\![x]\!]$
$c \in [\![p]\!] \Rightarrow [\![z]\!] \subseteq [\![\alpha]\!]$ for each $c \in \mathit{Cells}$
$[\![q]\!] \subseteq [\![p]\!]$
$y \in [\![q]\!]$
$c \in [\![p]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![x]\!]$ for each $c \in \mathit{Cells}$
$z \in [\![p]\!]$

- Smallest solution:

$pt(\mathrm{p}) = \{ \texttt{alloc-1}, \mathrm{y}, \mathrm{z} \}$

$pt(\mathrm{q}) = \{ \mathrm{y} \}$

$pt(\mathrm{x}) = pt(\mathrm{y}) = pt(\mathrm{z}) = \emptyset$

A specialized cubic solver

- At each load/store instruction, instead of generating a conditional constraint for each cell, generate a single universally quantified constraint:
  - $t \in [\![x]\!]$
  - $[\![x]\!] \subseteq [\![y]\!]$
  - $\forall t \in [\![x]\!]\colon [\![t]\!] \subseteq [\![y]\!]$
  - $\forall t \in [\![x]\!]\colon [\![y]\!] \subseteq [\![t]\!]$
- Whenever a token is added to a set, lazily add new edges according to the universally quantified constraints
- Note that every token is also a constraint variable here
- Still cubic complexity, but faster in practice

A specialized cubic solver

- $x.sol \subseteq T$: the set of tokens for x (the bitvectors)
- $x.succ \subseteq V$: the successors of x (the edges)
- $x.from \subseteq V$: the first kind of quantified constraints for x
- $x.to \subseteq V$: the second kind of quantified constraints for x
- $W \subseteq T \times V$: a worklist (initially empty)

Implementation: `SpecialCubicSolver`

A specialized cubic solver

- $t \in [\![x]\!]$
  addToken(t, x)
  propagate()
- $[\![x]\!] \subseteq [\![y]\!]$
  addEdge(x, y)
  propagate()
- $\forall t \in [\![x]\!]: [\![t]\!] \subseteq [\![y]\!]$
  add y to x.from
  for each t in x.sol
    addEdge(t, y)
  propagate()
- $\forall t \in [\![x]\!]: [\![y]\!] \subseteq [\![t]\!]$
  add y to x.to
  for each t in x.sol
    addEdge(y, t)
  propagate()

```
addToken(t, x):
  if t ∉ x.sol
    add t to x.sol
    add (t, x) to W

addEdge(x, y):
  if x ≠ y ∧ y ∉ x.succ
    add y to x.succ
    for each t in x.sol
      addToken(t, y)

propagate():
  while W ≠ ∅
    pick and remove (t, x) from W
    for each y in x.from
      addEdge(t, y)
    for each y in x.to
      addEdge(y, t)
    for each y in x.succ
      addToken(t, y)
```

Agenda

- Introduction to pointer analysis
- Andersen's analysis
- **Steensgaard's analysis**
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis

Steensgaard's analysis

- View assignments as being bidirectional
- Generate constraints:
  - $X = \texttt{alloc}\ P$: $\texttt{alloc-}i \in [\![X]\!]$
  - $X = \&Y$: $Y \in [\![X]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {*}Y$: $c \in [\![Y]\!] \Rightarrow [\![c]\!] = [\![X]\!]$ for each $c \in \mathit{Cells}$
  - ${*}X = Y$: $c \in [\![X]\!] \Rightarrow [\![Y]\!] = [\![c]\!]$ for each $c \in \mathit{Cells}$
- Extra constraints:

  $c_1, c_2 \in [\![c]\!] \Rightarrow [\![c_1]\!] = [\![c_2]\!]$ and $[\![c_1]\!] \cap [\![c_2]\!] \neq \emptyset \Rightarrow [\![c_1]\!] = [\![c_2]\!]$

  (whenever a cell may point to two cells, they are essentially merged into one)

- Steensgaard's original formulation uses conditional unification for $X = Y$:
  $c \in [\![Y]\!] \Rightarrow [\![X]\!] = [\![Y]\!]$ for each $c \in \mathit{Cells}$ (avoids unifying if Y is never a pointer)

Steensgaard's analysis

- Reformulate as term unification
- Generate constraints:
  - $X = \texttt{alloc}\ P$: $[\![X]\!] = \Uparrow[\![\texttt{alloc-}i]\!]$
  - $X = \&Y$: $[\![X]\!] = \Uparrow[\![Y]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {*}Y$: $[\![Y]\!] = \Uparrow\alpha \ \wedge\ [\![X]\!] = \alpha$ where α is fresh
  - ${*}X = Y$: $[\![X]\!] = \Uparrow\alpha \ \wedge\ [\![Y]\!] = \alpha$ where α is fresh
- Terms:
  - term variables, e.g. $[\![X]\!]$, $[\![\texttt{alloc-}i]\!]$, α (each representing the possible values of a cell)
  - each a single (unary) term constructor $\Uparrow t$ (representing pointers)
  - each $[\![c]\!]$ is now a term variable, not a constraint variable holding a set of cells
- Fits with our unification solver! (union-find…)
- The points-to map is defined as pt(X) = $\{ c \in Cells \mid [\![X]\!] = \Uparrow[\![c]\!] \}$
- Note that there is only one kind of term constructor, so unification never fails

Applying Steensgaard

- Generated constraints (as sets or terms, respectively):

$\texttt{alloc-1} \in [\![p]\!]$
$[\![y]\!] = [\![x]\!]$
$[\![z]\!] = [\![x]\!]$
$c \in [\![p]\!] \Rightarrow [\![z]\!] = [\![c]\!]$ for each $c \in \mathit{Cells}$
$[\![q]\!] = [\![p]\!]$
$y \in [\![q]\!]$
$c \in [\![p]\!] \Rightarrow [\![c]\!] = [\![x]\!]$ for each $c \in \mathit{Cells}$
$z \in [\![p]\!]$
+ the extra constraints

$[\![p]\!] = \uparrow [\![\texttt{alloc-1}]\!]$
$[\![y]\!] = [\![x]\!]$
$[\![z]\!] = [\![x]\!]$
$[\![p]\!] = \uparrow \alpha_1$ $\quad [\![z]\!] = \alpha_1$
$[\![q]\!] = [\![p]\!]$
$[\![q]\!] = \uparrow [\![y]\!]$
$[\![p]\!] = \uparrow \alpha_2$ $\quad [\![x]\!] = \alpha_2$
$[\![p]\!] = \uparrow [\![z]\!]$

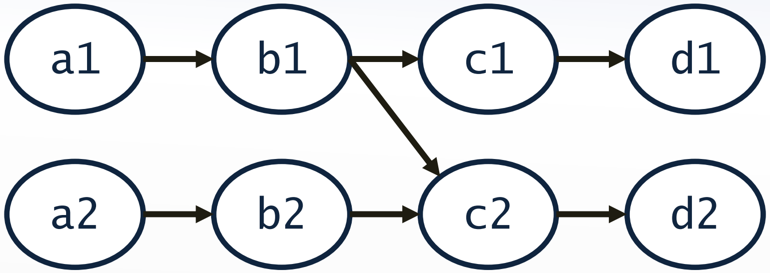
- Smallest solution:

$\mathit{pt}(p) = \{\, \texttt{alloc-1}, y, z \,\}$
$\mathit{pt}(q) = \{\, \texttt{alloc-1}, y, z \,\}$
...

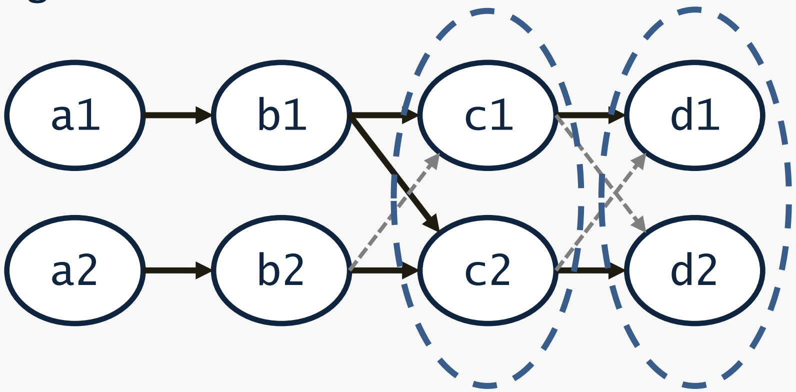
Another example

```
a1 = &b1;
b1 = &c1;
c1 = &d1;
a2 = &b2;
b2 = &c2;
c2 = &d2;
b1 = &c2;
```

Andersen:



Steensgaard:



Recall our type analysis…

- Focusing on pointers…
- Constraints:
  - $X = \texttt{alloc}\ P$: $[\![X]\!] = \uparrow[\![P]\!]$
  - $X = \&Y$: $[\![X]\!] = \uparrow[\![Y]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {*Y}$: $\uparrow[\![X]\!] = [\![Y]\!]$
  - ${*X} = Y$: $[\![X]\!] = \uparrow[\![Y]\!]$
- Implicit extra constraint for term equality:

$$\uparrow t_1 = \uparrow t_2 \Rightarrow t_1 = t_2$$

- Assuming the program type checks, is the solution for pointers the same as for Steensgaard's analysis?

Agenda

- Introduction to pointer analysis
- Andersen's analysis
- Steensgaard's analysis
- **Interprocedural pointer analysis**
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis

Interprocedural pointer analysis

- In TIP, function values and pointers may be mixed together:

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

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

- The idea: Treat function values as a kind of pointers

Function call normalization

- Assume that all function calls are of the form

$$X = X_0(X_1, \ldots, X_n)$$

- Assume that all return statements are of the form

$$\texttt{return}\ X';$$

- As usual, simply introduce lots of temporary variables…

- Include all function names in *Cells*

CFA with Andersen

- For the function call

  $$X = X_0(X_1, \ldots, X_n)$$

  and every occurrence of

  $$f(X'_1, \ldots, X'_n)\ \{ \ldots\ \mathtt{return}\ X'; \}$$

  add these constraints:

  $$f \in [\![f]\!]$$

  $$f \in [\![X_0]\!] \Rightarrow ([\![X_i]\!] \subseteq [\![X'_i]\!] \text{ for i=1,...,}n \wedge [\![X']\!] \subseteq [\![X]\!])$$

- (Similarly for simple function calls)
- Fits directly into the cubic framework!

Andersen's analysis is already closely connected to control-flow analysis!

CFA with Steensgaard

- For the function call

  $$X = X_0(X_1, \ldots, X_n)$$

  and every occurrence of

  $$f(X'_1, \ldots, X'_n)\ \{ \ldots \ \texttt{return}\ X'; \}$$

  add these constraints:

  $$f \in [\![f]\!]$$

  $$f \in [\![X_0]\!] \Rightarrow ([\![X_i]\!] = [\![X'_i]\!] \text{ for i=1,...,}n \wedge [\![X']\!] = [\![X]\!])$$

- (Similarly for simple function calls)
- Fits into the unification framework, but requires a generalization of the ordinary union-find solver

Context-sensitive pointer analysis

```
foo(a) {
  return *a;
}

bar() {
  ...
  x = alloc null; // alloc-1
  y = alloc null; // alloc-2
  *x = alloc null; // alloc-3
  *y = alloc null; // alloc-4
  ...
  q = foo(x);
  w = foo(y);
  ...
}
```

Are q and w aliases?

Context-sensitive pointer analysis

- Generalize the abstract domain $\mathit{Cells} \rightarrow \mathcal{P}(\mathit{Cells})$ to
  $$\mathit{Contexts} \rightarrow \mathit{Cells} \rightarrow \mathcal{P}(\mathit{Cells})$$
  (or equivalently: $\mathit{Cells} \times \mathit{Contexts} \rightarrow \mathcal{P}(\mathit{Cells})$)
  where *Contexts* is a (finite) set of call contexts
- As usual, many possible choices of *Contexts*
  - recall the call string approach and the functional approach
- We can also track the set of reachable contexts (like the use of lifted lattices earlier):
  $$\mathit{Contexts} \rightarrow \mathrm{lift}(\mathit{Cells} \rightarrow \mathcal{P}(\mathit{Cells}))$$

- Does this still fit into the cubic solver?

Context-sensitive pointer analysis

```
mk() {
  return alloc null; // alloc-1
}

baz() {
  var x,y;
  x = mk();
  y = mk();
  ...
}
```

Are x and y aliases?

$[\![x]\!]$ = {alloc-1}
$[\![y]\!]$ = {alloc-1}

Context-sensitive pointer analysis

- We can go one step further and introduce *context-sensitive heap* (a.k.a. *heap cloning*)
- Let each abstract cell be a pair of
  - `alloc-`*i* (the `alloc` with index *i*) or *X* (a program variable)
  - a heap context from a (finite) set *HeapContexts*
- This allows abstract cells to be named by the source code allocation site ***and (information from) the current context***
- One choice:
  - set *HeapContexts* = *Contexts*
  - at `alloc`, use the entire current call context as heap context

Context-sensitive pointer analysis with heap cloning

Assuming we use the call string approach with k=1, so *Contexts* = {ε, c1, c2}, and *HeapContexts* = *Contexts*

```
mk() {
  return alloc null; // alloc-1
}

baz() {
  var x,y;
  x = mk(); // c1
  y = mk(); // c2
  ...
}
```

Are x and y aliases?

$[\![x]\!] = \{\ (\mathtt{alloc\text{-}1}, c1)\ \}$

$[\![y]\!] = \{\ (\mathtt{alloc\text{-}1}, c2)\ \}$

Agenda

- Introduction to pointer analysis
- Andersen's analysis
- Steensgaard's analysis
- Interprocedural pointer analysis
- **Records and objects**
- Null pointer analysis
- Flow-sensitive pointer analysis

Records in TIP

$$
\begin{aligned}
Exp \rightarrow\ & \ldots \\
& \mid \texttt{\{}\ Id \texttt{:} Exp \texttt{,}\ \ldots \texttt{,}\ Id \texttt{:} Exp\ \texttt{\}} \\
& \mid\ Exp \texttt{.} Id
\end{aligned}
$$

- Field write operations: see SPA...
- Values of record fields cannot themselves be records
- After normalization:
  - $X = \texttt{\{}\ F_1 \texttt{:} X_1 \texttt{,}\ \ldots \texttt{,}\ F_k \texttt{:} X_k\ \texttt{\}}$
  - $X = \texttt{alloc}\ \texttt{\{}\ F_1 \texttt{:} X_1 \texttt{,}\ \ldots \texttt{,}\ F_k \texttt{:} X_k\ \texttt{\}}$
  - $X = Y.F$

Let us extend Andersen's analysis accordingly...

Constraint variables for record fields

- $[\![\cdot]\!]$: $(\mathit{Cells} \cup (\mathit{Cells} \times \mathit{Fields})) \rightarrow \mathcal{P}(\mathit{Cells})$
  where *Fields* is the set of field names in the program

- Notation: $[\![\mathtt{c.f}]\!]$ means $[\![(\mathtt{c}, \mathtt{f})]\!]$

Analysis constraints

- $X = \{\ F_1 \mathtt{:} X_1 \mathtt{,}\ \ldots \mathtt{,}\ F_k \mathtt{:} X_k\ \}$: $[\![X_1]\!] \subseteq [\![X.F_1]\!] \wedge \ldots \wedge [\![X_k]\!] \subseteq [\![X.F_k]\!]$
- $X = \mathtt{alloc}\ \{\ F_1 \mathtt{:} X_1 \mathtt{,}\ \ldots \mathtt{,}\ F_k \mathtt{:} X_k\ \}$: $\mathtt{alloc}\text{-}i \in [\![X]\!]\ \wedge$
  $[\![X_1]\!] \subseteq [\![\mathtt{alloc}\text{-}i.F_1]\!] \wedge \ldots \wedge [\![X_k]\!] \subseteq [\![\mathtt{alloc}\text{-}i.F_k]\!]$
- $X = Y.F$: $[\![Y.F]\!] \subseteq [\![X]\!]$

- $X = Y$: $[\![Y]\!] \subseteq [\![X]\!]\ \wedge\ [\![Y.F]\!] \subseteq [\![X.F]\!]$ for each $F \in \mathit{Fields}$
- $X = {*}Y$: $c \in [\![Y]\!] \Rightarrow ([\![c]\!] \subseteq [\![X]\!]\ \wedge\ [\![c.F]\!] \subseteq [\![X.F]\!])$
  for each $c \in \mathit{Cells}$ and $F \in \mathit{Fields}$
- ${*}X = Y$: $c \in [\![X]\!] \Rightarrow ([\![Y]\!] \subseteq [\![c]\!]\ \wedge\ [\![Y.F]\!] \subseteq [\![c.F]\!])$
  for each $c \in \mathit{Cells}$ and $F \in \mathit{Fields}$

See example in SPA

Objects as mutable heap records

$Exp \rightarrow \ldots$
  | Id
  | `alloc` { Id `:` Exp `,` … `,` Id `:` Exp }
  | `(*`Exp`).`Id
  | `null`

$Stm \rightarrow \ldots$
  | Id = Exp `;`
  | `(*`Exp`).`Id = Exp `;`

- `E.X` in Java corresponds to `(*E).X` in TIP (or C)
- Can only create pointers to heap-allocated records (=objects), not to variables or to cells containing non-record values

Agenda

- Introduction to pointer analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Interprocedural pointer analysis
- Records and objects
- **Null pointer analysis**
- Flow-sensitive pointer analysis

Null pointer analysis

- Decide for every dereference `*p`,
  is `p` different from `null`?

- (Why not just treat null as a special cell
  in an Andersen or Steensgaard-style analysis?)

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

- Let us consider an intraprocedural analysis
  (i.e. we ignore function calls)

A lattice for null analysis

- Define the simple lattice *Null*:

$$\begin{array}{c} ? \\ | \\ \text{NN} \end{array}$$

where NN represents "definitely **n**ot **n**ull"
and ? represents "maybe null"

- Use for every program point the map lattice:

$$\mathit{Cells} \rightarrow \mathit{Null}$$

(here for TIP without records)

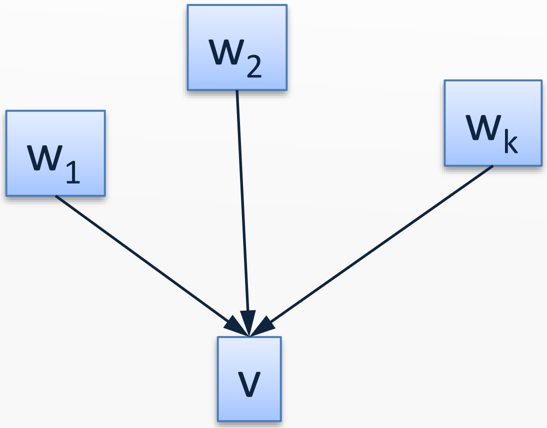
Setting up

- For every CFG node, v, we have a variable $[\![v]\!]$:
  - a map giving abstract values for all cells at the program point *after* v

- Auxiliary definition:

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

(i.e. we make a *forward* analysis)



Null analysis constraints

- For operations involving pointers:
  - $X = \texttt{alloc}\ P$: $[\![v]\!]$ = ???
  - $X = \&Y$: $[\![v]\!]$ = ???
  - $X = Y$: $[\![v]\!]$ = ???
  - $X = {*}Y$: $[\![v]\!]$ = ???
  - ${*}X = Y$: $[\![v]\!]$ = ???
  - $X = \texttt{null}$: $[\![v]\!]$ = ???

where *P* is null or an integer constant

- For all other CFG nodes:
  - $[\![v]\!] = \mathit{JOIN}(v)$

Null analysis constraints

- For a heap store operation $*X = Y$ we need to model the change of whatever X points to
- That may be *multiple* abstract cells (i.e. the cells $pt(X)$)
- With the present abstraction, each abstract heap cell `alloc-`i may describe *multiple* concrete cells
- So we settle for **weak** update:

$$*X = Y: \qquad [\![v]\!] = store(JOIN(v), X, Y)$$

$$\text{where } store(\sigma, X, Y) = \sigma[\underset{\alpha \in pt(X)}{\alpha \mapsto \sigma(\alpha) \sqcup \sigma(Y)}]$$

Null analysis constraints

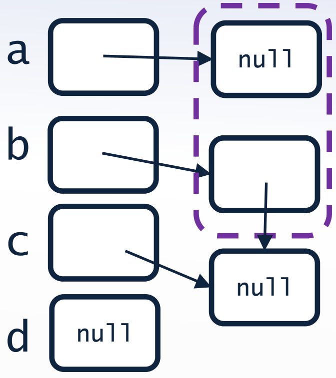
- For a heap load operation $X = {*}Y$ we need to model the change of the program variable X
- Our abstraction has a *single* abstract cell for X
- That abstract cell represents a *single* concrete cell
- So we can use **strong** update:

$$X = {*}Y: \qquad [\![v]\!] = load(JOIN(v), X, Y)$$

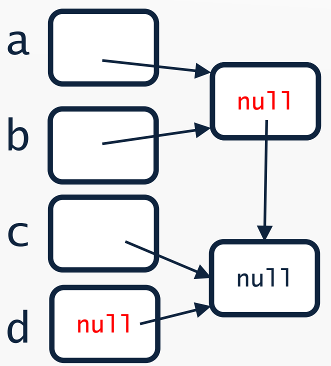
$$\text{where} \qquad load(\sigma, X, Y) = \sigma[X \mapsto \bigsqcup_{\alpha \in pt(Y)} \sigma(\alpha)]$$

Strong and weak updates

concrete execution:



abstract execution:




```
mk() {
  return alloc null; // alloc-1
}

...
a = mk();
b = mk();
c = alloc null; // alloc-2
*b = c; // strong update here would be unsound!
d = *a;
```

is d null here?

The abstract cell `alloc-1` corresponds to *multiple concrete cells*

Strong and weak updates

```
a = alloc null; // alloc-1
b = alloc null; // alloc-2
*a = alloc null; // alloc-3
*b = alloc null; // alloc-4
if (...) {
  x = a;
} else {
  x = b;
}
n = null;
*x = n; // strong update here would be unsound!
c = *x;
```

is c null here?

The points-to set for x contains *multiple abstract cells*

Null analysis constraints

- $X = \texttt{alloc}\ P$: $[\![v]\!] = \mathit{JOIN}(v)[X \mapsto \mathrm{NN}, \texttt{alloc-i} \mapsto ?]$
- $X = \&Y$: $[\![v]\!] = \mathit{JOIN}(v)[X \mapsto \mathrm{NN}]$
- $X = Y$: $[\![v]\!] = \mathit{JOIN}(v)[X \mapsto \mathit{JOIN}(v)(Y)]$
- $X = \texttt{null}$: $[\![v]\!] = \mathit{JOIN}(v)[X \mapsto ?]$

could be improved…

- In each case, the assignment modifies a program variable
- So we can use strong updates, as for heap load operations

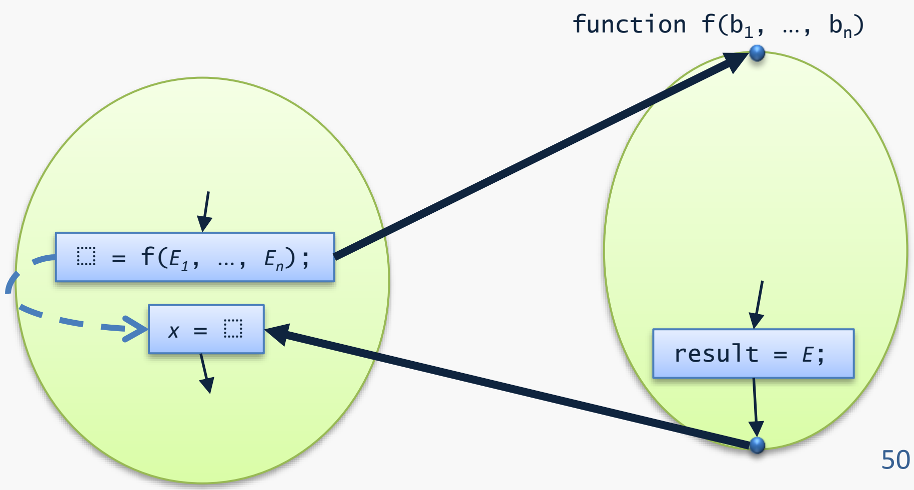
Strong and weak updates, revisited

- Strong update: $\sigma[c \mapsto \textit{new-value}]$
  - possible if *c* is known to refer to a single concrete cell
  - works for assignments to local variables
    (as long as TIP doesn't have e.g. nested functions)

- Weak update: $\sigma[c \mapsto \sigma(c) \sqcup \textit{new-value}]$
  - necessary if *c* may refer to multiple concrete cells
  - bad for precision, we lose some of the power of flow-sensitivity
  - required for assignments to heap cells
    (unless we extend the analysis abstraction!)

Interprocedural null analysis

- Context insensitive or context sensitive, as usual…
  - at the after-call node, use the heap from the callee
- But be careful!
  Pointers to local variables may escape to the callee
  - the abstract state at the after-call node cannot simply copy the abstract values for local variables from the abstract state at the call node



Using the null analysis

- The pointer dereference *p is "safe" at entry of v if

  JOIN(v)(p) = NN

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

Example program

```
p = alloc null;
q = &p;
n = null;
*q = n;
*p = n;
```

Andersen generates:

pt(p) = {`alloc-1`}

pt(q) = {p}

pt(n) = ∅

Generated constraints

$[\![$`p=alloc null`$]\!] = \bot[$p $\mapsto$ NN , `alloc-1` $\mapsto$?$]$

$[\![$q=&p$]\!] = [\![$`p=alloc null`$]\!][$q $\mapsto$ NN$]$

$[\![$`n=null`$]\!] = [\![$q=&p$]\!][$n $\mapsto$?$]$

$[\![$*q=n$]\!] = [\![$`n=null`$]\!][$p $\mapsto$ $[\![$`n=null`$]\!]$(p) $\sqcup$ $[\![$`n=null`$]\!]$(n)$]$

$[\![$*p=n$]\!] = [\![$*q=n$]\!][$`alloc-1` $\mapsto$ $[\![$*q=n$]\!]$(`alloc-1`) $\sqcup$ $[\![$*q=n$]\!]$(n)$]$

Solution

⟦`p=alloc null`⟧ = [p ↦ NN, q ↦ NN, n ↦ NN , `alloc-1` ↦ ?]
⟦q=&p⟧ = [p ↦ NN, q ↦ NN, n ↦ NN , `alloc-1` ↦ ?]
⟦`n=null`⟧ = [p ↦ NN, q ↦ NN, n ↦ ?, `alloc-1` ↦ ?]
⟦*q=n⟧ = [p ↦ ?, q ↦ NN, n ↦ ?, `alloc-1` ↦ ?]
⟦*p=n⟧ = [p ↦ ?, q ↦ NN, n ↦ ?, `alloc-1` ↦ ?]

- At the program point before the statement `*q=n` the analysis now knows that q is definitely non-null
- … and before `*p=n`, the pointer p is maybe null
- Due to the weak updates for all heap store operations, precision is bad for `alloc-i` cells

Agenda

- Introduction to pointer analysis
- Andersen's analysis
- Steensgaard's analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- **Flow-sensitive pointer analysis**

Points-to graphs

- Graphs that describe possible heaps:
  - nodes are abstract cells
  - edges are possible pointers between the cells
- The lattice of points-to graphs is $\mathcal{P}(Cells \times Cells)$ ordered under subset inclusion
  (or alternatively, $Cells \rightarrow \mathcal{P}(Cells)$)
- For every CFG node, v, we introduce a constraint variable $[\![v]\!]$ describing the state *after* v
- Intraprocedural analysis (i.e. ignore function calls)

Constraints

- For pointer operations:
  - $X = \texttt{alloc}\ P$: $[\![v]\!] = JOIN(v)\downarrow X \cup \{ (X, \texttt{alloc-}i) \}$
  - $X = \&Y$: $[\![v]\!] = JOIN(v)\downarrow X \cup \{ (X, Y) \}$
  - $X = Y$: $[\![v]\!] = JOIN(v)\downarrow X \cup \{ (X, t) \mid (Y, t) \in JOIN(v)\}$
  - $X = {*}Y$: $[\![v]\!] = JOIN(v)\downarrow X \cup \{ (X, t) \mid (Y, s) \in \sigma, (s, t) \in JOIN(v)\}$
  - ${*}X = Y$: $[\![v]\!] = JOIN(v) \cup \{ (s, t) \mid (X, s) \in JOIN(v), (Y, t) \in JOIN(v)\}$

    note: weak update!
  - $X = \texttt{null}$: $[\![v]\!] = JOIN(v)\downarrow X$

  where $\sigma\downarrow X = \{ (s,t) \in \sigma \mid s \neq X\}$

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

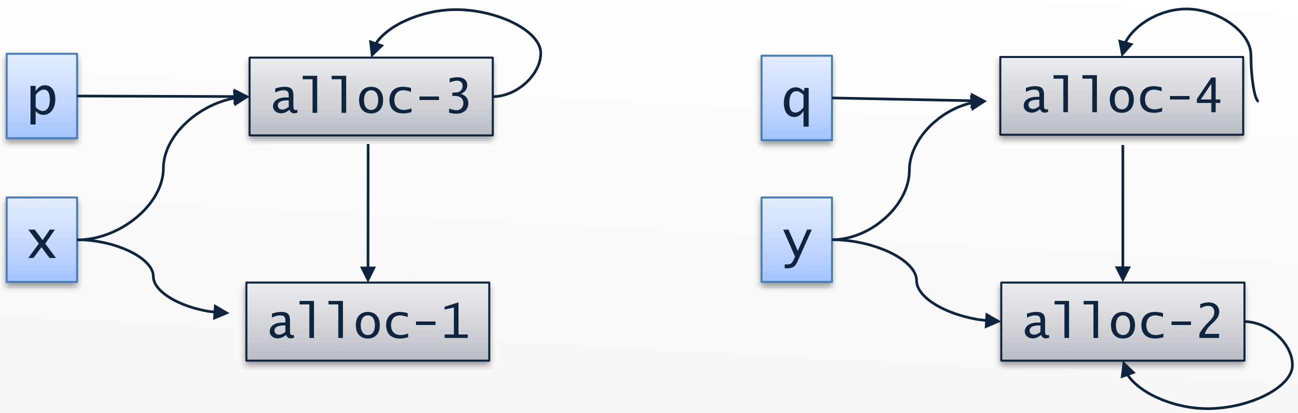
- For all other CFG nodes:
  - $[\![v]\!] = JOIN(v)$

Example program

```
var x,y,n,p,q;
x = alloc null; y = alloc null;
*x = null; *y = y;
n = input;
while (n>0) {
  p = alloc null; q = alloc null;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

Result of analysis

- After the loop we have this points-to graph:



- We conclude that x and y will always be disjoint

Points-to maps from points-to graphs

- A points-to map for each program point v:

  $$pt(X) = \{\, t \mid (X,t) \in [\![\mathrm{v}]\!] \,\}$$

- More expensive, but more precise:
  - Andersen: $pt(\mathrm{x}) = \{\, \mathrm{y}, \mathrm{z} \,\}$
  - flow-sensitive: $pt(\mathrm{x}) = \{\, \mathrm{z} \,\}$

```
x = &y;
x = &z;
```

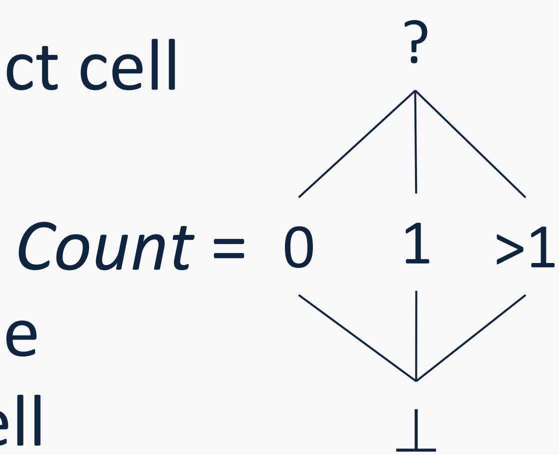
Improving precision with abstract counting

- The points-to graph is missing information:
  - `alloc-2` nodes always form a self-loop in the example

- We need a more detailed lattice:

$$\mathcal{P}(\mathit{Cells} \times \mathit{Cells}) \times (\mathit{Cell} \rightarrow \mathit{Count})$$

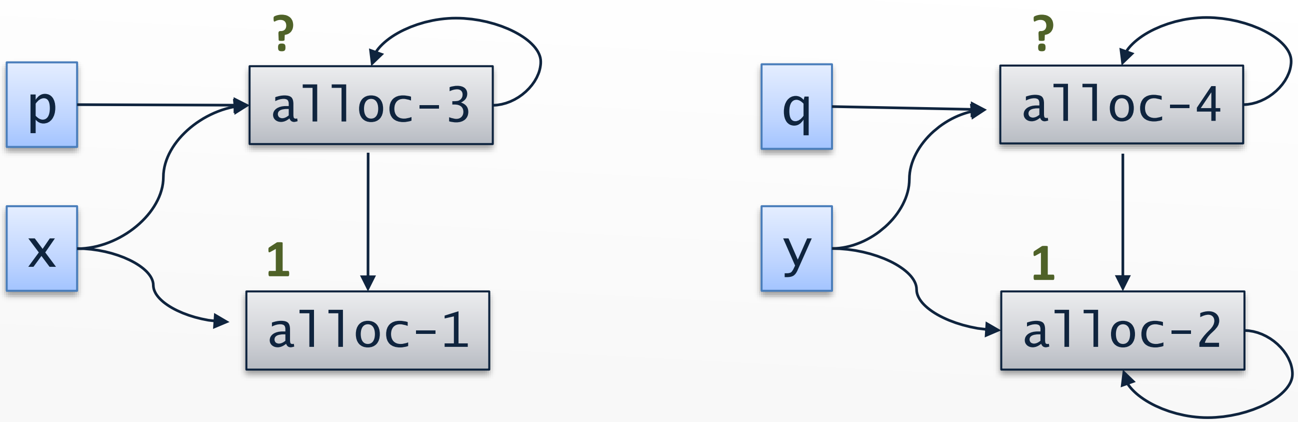
  where we for each cell keep track of how many concrete cells that abstract cell describes



- This permits **strong updates** on those that describe precisely 1 concrete cell

Better results

- After the loop we have this extended points-to graph:



- Thus, `alloc-2` cells form a self-loop
- Both `alloc-1` and `alloc-2` permit strong updates

Escape analysis

- Perform a points-to analysis
- Look at return expression
- Check reachability in the points-to graph to arguments or variables defined in the function itself

- None of those

  ⇓

  no escaping stack cells

```
baz()  {
  var x;
  return &x;
}

main() {
  var p;
  p=baz();
  *p=1;
  return *p;
}
```