Pointer Analysis

Yu Zhang

Most content comes from http://cs.au.dk/~amoeller/spa/
http://staff.ustc.edu.cn/~yuzhang/pldpa

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?)

$$
\begin{aligned}
E \rightarrow\ & \&X \\
\mid\ & \texttt{alloc}\ E \\
\mid\ & {*E} \\
\mid\ & \texttt{null} \\
\mid\ & \ldots
\end{aligned}
$$

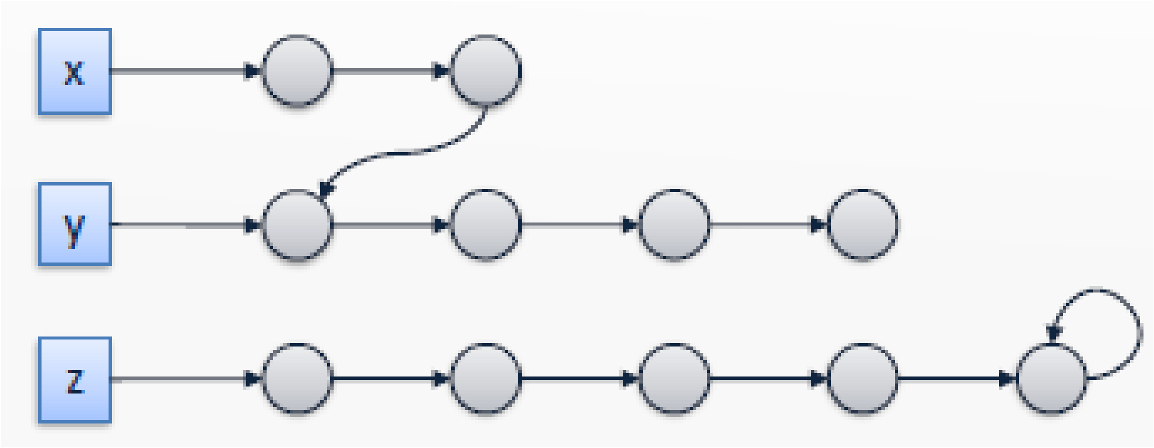
$$
\begin{aligned}
S \rightarrow\ & {*X} = E; \\
\mid\ & \ldots
\end{aligned}
$$

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

Depend on whether x and y point to the same location, if so, z is -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?

```
p =alloc null
*p = z
```
alloc-1

- 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 取址*):
  - include all alloc-*i* cells 注：为程序正文中的分配点
  - 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*=alloc *P*   where *P* is null or an integer constant
  - *X*=&*Y*
  - *X*=*Y*
  - *X*=**Y*
  - **X*=*Y*
  - *X*=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

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. $[\![\cdot]\!]: \mathit{Cells} \rightarrow \mathcal{P}(\mathit{Cells})$

基于集合的包含关系

- Generate constraints:
  - $X =$ `alloc` P: `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 $c \in \mathit{Cells}$
  - $X =$ `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;
```

$X =$ `alloc` P:	`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 $c \in Cells$
${*}X = Y$:	$c \in [\![X]\!] \Rightarrow [\![Y]\!] \subseteq [\![c]\!]$ for each $c \in Cells$
$X =$ `null`:	(no constraints)

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

Applying Andersen

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

$\texttt{alloc-1} \in [\![p]\!]$

$[\![y]\!] \subseteq [\![x]\!]$

$[\![z]\!] \subseteq [\![x]\!]$

$c \in [\![p]\!] \Rightarrow [\![z]\!] \subseteq [\![\alpha]\!]$ 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:

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

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

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

A Specialized Cubic Solver

- At each load/store instruction, instead of generating a conditional constraint for each cell, generate a single universally quantified constraint:
  - $t \in [\![x]\!]$
  - $[\![x]\!] \subseteq [\![y]\!]$
  - $\forall t \in [\![x]\!]: [\![t]\!] \subseteq [\![y]\!]$
  - $\forall t \in [\![x]\!]: [\![y]\!] \subseteq [\![t]\!]$

Original constraint forms

- $t \in x$
- $t \in x \Rightarrow y \subseteq z$

- 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 = `alloc` P: `alloc`-$i \in [\![X]\!]$
  - $X = \&Y$: $Y \in [\![X]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {*}Y$: $c \in [\![Y]\!] \Rightarrow [\![c]\!] = [\![X]\!]$ for each $c \in Cells$
  - ${*}X = Y$: $c \in [\![X]\!] \Rightarrow [\![Y]\!] = [\![c]\!]$ for each $c \in Cells$
- Extra constraints:

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

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

基于类型及其**等价**关系

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

Steensgaard's Analysis

- Reformulate as term unification
- Generate constraints:
  - X = `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 $\mathrm{pt}(X) = \{\, c \in \mathit{Cells} \mid [\![X]\!] = \uparrow[\![c]\!] \,\}$
- Note that there is only one kind of term constructor, so unification never fails

Applying Steensgaard

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

$[\![p]\!] = \uparrow[\![\texttt{alloc-1}]\!]$

$[\![y]\!] = [\![x]\!]$

$[\![z]\!] = [\![x]\!]$

$[\![p]\!] = \uparrow\alpha_1 \qquad [\![z]\!] = \alpha_1$

$[\![q]\!] = [\![p]\!]$

$[\![q]\!] = \uparrow[\![y]\!]$

$[\![p]\!] = \uparrow\alpha_2 \qquad [\![x]\!] = \alpha_2$

$[\![p]\!] = \uparrow[\![z]\!]$

Smallest solution:

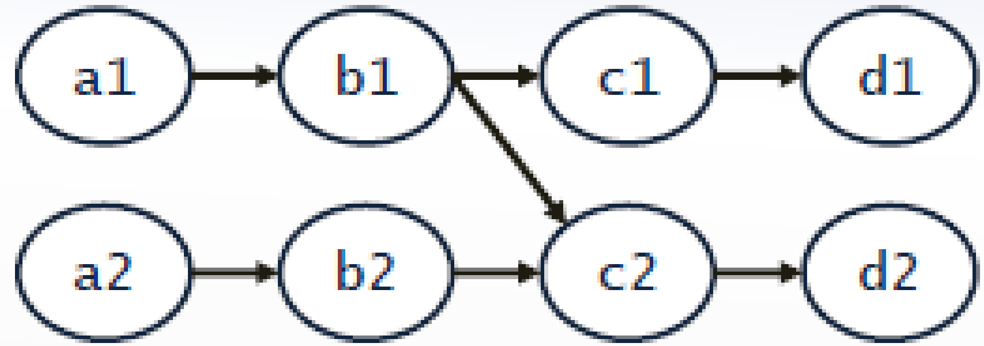
pt(p) = { alloc-1, y, z}

pt(q) = {alloc-1, y, z}

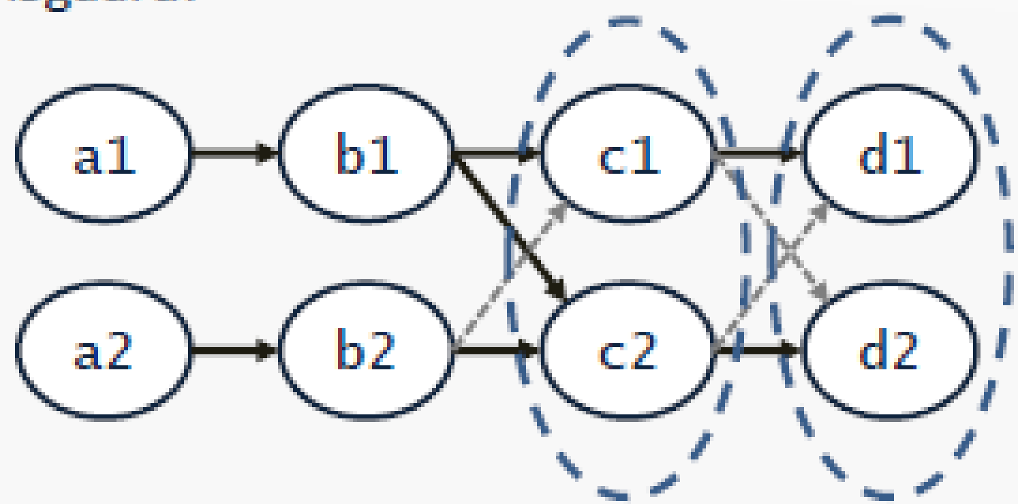
Another Example

```
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 Points-to 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

  $x = y(a_1,\ldots,a_n)$
- *y* may be a variable whose value is a function pointer
- Assume that all return statements are of the form

  return *z*;

- As usual, simply introduce lots of temporary variables…

- Include all function names in *Cells*

CFA with Andersen

- For the function call

  $x = y(a_1, \ldots, a_n)$

  and every occurrence of

  $f(x_1, \ldots, x_n)$ { ... `return` z; }

  add these constraints:

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

  $f \in [\![y]\!] \Rightarrow ([\![a_i]\!] \subseteq [\![x_i]\!]$ for i=1,...,$n \wedge [\![z]\!] \subseteq [\![x]\!])$

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

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

CFA with Steensgaard

- For the function call

  $$x = y(a_1, \ldots, a_n)$$

  and every occurrence of

  $$f(x_1, \ldots, x_n)\ \{ \ldots \texttt{return}\ z; \}$$

  add these constraints:

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

  $$f \in [\![y]\!] \Rightarrow ([\![a_i]\!] = [\![x_i]\!] \text{ for i=1,...,}n \wedge [\![z]\!] = [\![x]\!])$$

- (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 $Cells \rightarrow \mathcal{P}(Cells)$ to

  $$Contexts \rightarrow Cells \rightarrow \mathcal{P}(Cells)$$

  (or equivalently: $Cells \times Contexts \rightarrow \mathcal{P}(Cells)$)
  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):

  $$Contexts \rightarrow \text{lift}(Cells \rightarrow \mathcal{P}(Cells))$$

- 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?

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

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?

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

$[\![y]\!]$ = { (alloc-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

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

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

  Let us extend Andersen's analysis accordingly …

Constraint Variables for Record Fields

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

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

Analysis Constraints

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

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

See example in SPA

Objects as Mutable Heap Records

Exp → ...
  | *Id*
  | `alloc` { *Id* : *Exp* , ... , *Id* : *Exp* }
  | (* *Exp*) . *Id*
  | `null`

Stm → ...
  | *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*:

  ?
  |
  NN

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

- Use for every program point the map lattice:

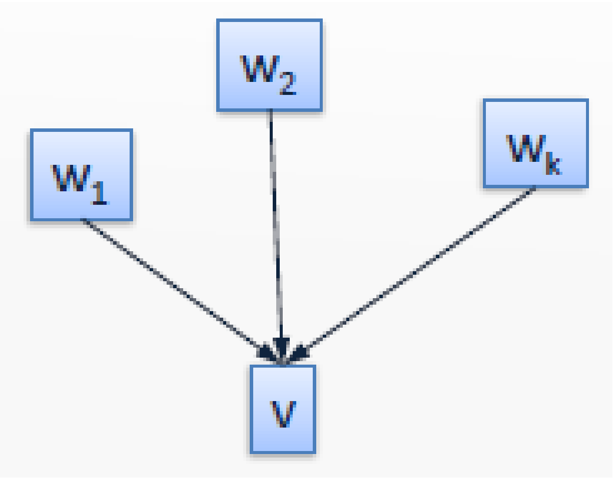
  $$\mathit{Cells} \rightarrow \mathit{Null}$$

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 =alloc P: $[\![v]\!] =$ *???*
  - X =&Y: $[\![v]\!] =$ *???*
  - X =Y: $[\![v]\!] =$ *???*
  - X =*Y: $[\![v]\!] =$ *???*
  - *X =Y: $[\![v]\!] =$ *???*
  - X =null: $[\![v]\!] =$ *???*

where P is null or an integer constant

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

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:

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

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

Null Analysis Constraints

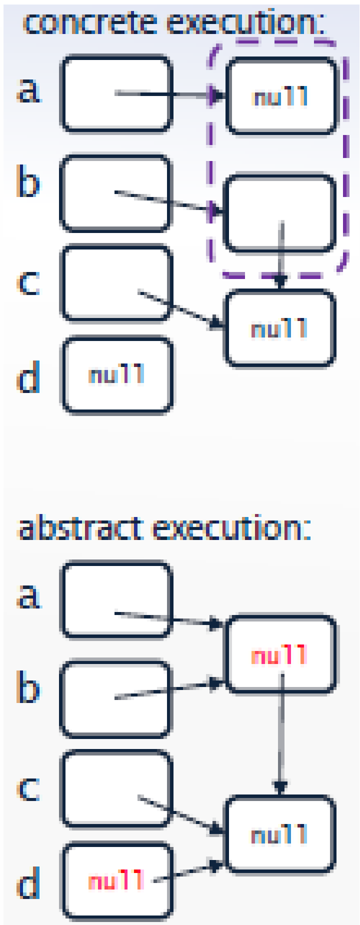
- For a heap load operation $X = *Y$ we need to model the change of the program variable X
- Our abstraction has a *single* abstract cell for X
- That abstract cell represents a *single* concrete cell
- So we can use **strong** update:

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

where

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

Strong and Weak Updates




```
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?

weak update

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

strong update

$store(\sigma, X, Y) = \sigma[\alpha \mapsto \cancel{\sigma(\alpha) \sqcup}\ \boxed{\sigma(Y)}]_{\alpha \in pt(X)}$

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

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

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!)

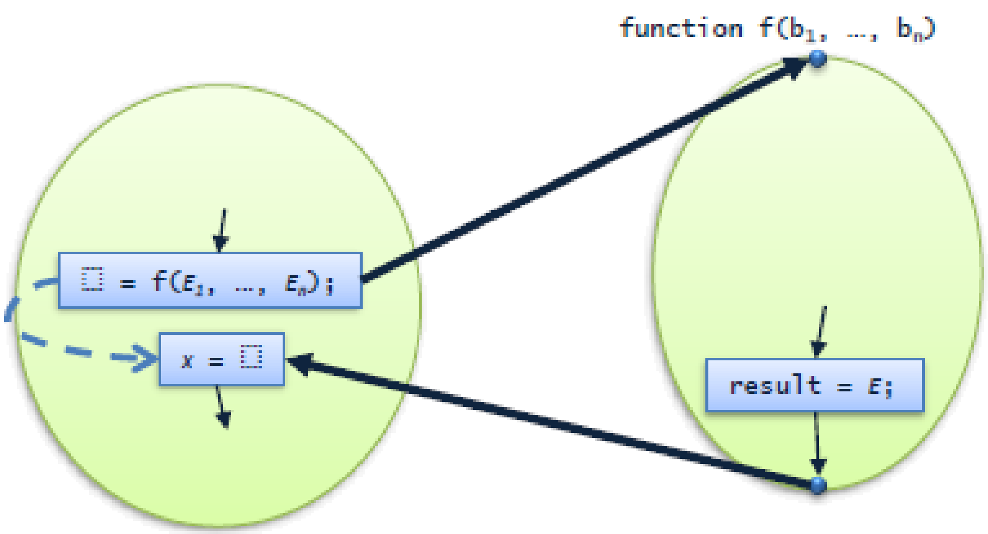
Interprocedural Null Analysis

- 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

Escape Analysis
逃逸分析：分析对象是否逃逸出一个函数



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 & Constraints

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

Andersen generates:

$pt(\mathrm{p}) = \{\text{alloc-1}\}$

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

$pt(\mathrm{n}) = \varnothing$

$[\![\texttt{p=alloc null}]\!] = \bot[\texttt{p} \mapsto \mathrm{NN}, \texttt{alloc-1} \mapsto ?]$

$[\![\texttt{q=\&p}]\!] = [\![\texttt{p=alloc null}]\!][\texttt{q} \mapsto \mathrm{NN}]$

$[\![\texttt{n=null}]\!] = [\![\texttt{q=\&p}]\!][\texttt{n} \mapsto ?]$

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

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

Solution

$[\![p=\text{alloc null}]\!]= [p\mapsto NN, q\mapsto NN, n\mapsto NN, \text{alloc-1}\mapsto ?]$

$[\![q=\&p]\!]= [p\mapsto NN, q\mapsto NN, n\mapsto NN, \text{alloc-1}\mapsto ?]$

$[\![n=\text{null}]\!]= [p\mapsto NN, q\mapsto NN, n\mapsto ?, \text{alloc-1}\mapsto ?]$

$[\![*q=n]\!]= [p\mapsto ?, q\mapsto NN, n\mapsto ?, \text{alloc-1}\mapsto ?]$

$[\![*p=n]\!]= [p\mapsto ?, q\mapsto NN, n\mapsto ?, \text{alloc-1}\mapsto ?]$

- 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 may be 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 $\mathcal{P}(\mathrm{Cells} \times \mathrm{Cells})$ ordered under subset inclusion (or alternatively, *Cells*$\rightarrow\mathcal{P}(\mathrm{Cells})$)
- For every CFG node, v, we introduce a constraint variable $[\![\mathrm{v}]\!]$ describing the state *after* v
- Intraprocedural analysis (i.e. ignore function calls)

Constraints

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

  note: weak update!

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

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

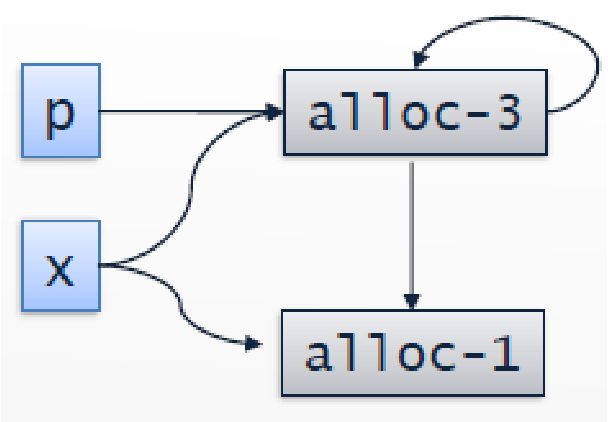
- For all other CFG nodes:
  - $[\![v]\!] = JOIN(v)$

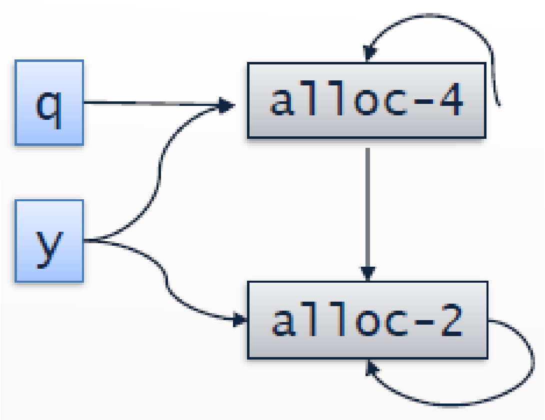
Example Program

```
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

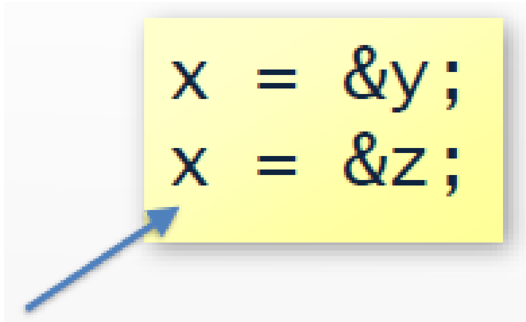
```
var x,y,n,p,q;
x = alloc null; y = alloc null;
*x = null; *y = y;
n = input;
while (n>0) {
  p = alloc null; q = alloc null;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

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: $pt(\mathrm{x}) = \{\, \mathrm{z} \}$



Improving Precision with Abstract Counting

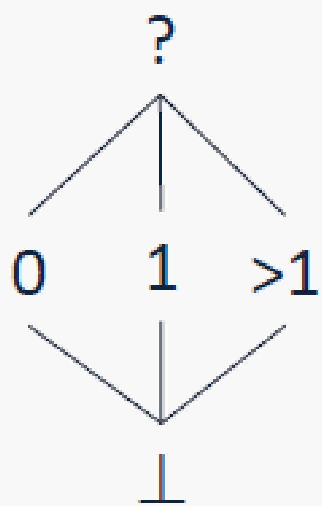
- The points-to graph is missing information:
  - `alloc-2` nodes always form a self-loop in the example

- We need a more detailed lattice:

$$2^{Cell \times Cell} \times (Cell \to Count)$$

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

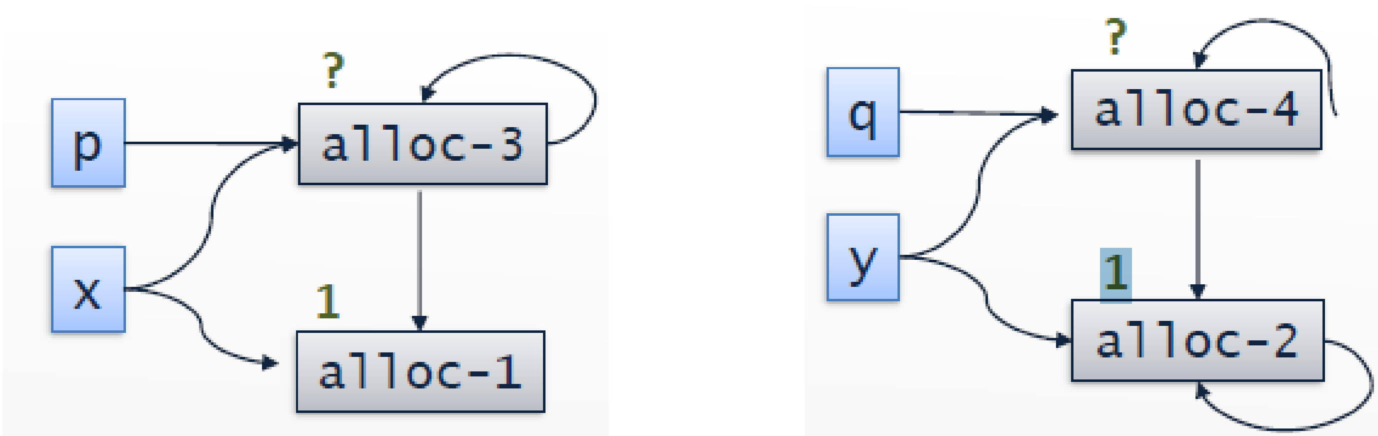
$Count =$



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

Constraints and Better Results

- *X* = alloc *P*: …
- **X* =*Y*: …
- …
- After the loop we have this extended points-to graph:



- Thus, alloc-2 nodes form a self-loop

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

$\Downarrow$

no escaping stack cells

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

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

THANKS