Pointer Analysis

Yu Zhang

Most content comes from http://cs.au.dk/~amoeller/spa/

Agenda

- **Introduction to points-to analysis**
- Andersen's analysis
- Steensgaards's analysis
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

Analyzing Programs with Pointers

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

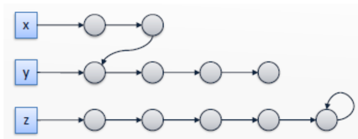
```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

```
E → &X
  | alloc E
  | *E
  | null
  | ...
```

```
S → *X = E;
  | ...
```


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
- A *conservative* ("may points-to") analysis:
  - the set may be too large
  - can show absence of aliasing: $pt(X) \cap pt(Y) = \varnothing$

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

- 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 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

7

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

8

Agenda

- Introduction to points-to analysis
- **Andersen's analysis**
- Steensgaards's analysis
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

9

Andersen's Analysis (1/2)

- For every cell *c*, introduce a constraint variable $[\![c]\!]$ ranging over sets of cells, i.e. $[\![\cdot]\!]$: $Cells \to 2^{Cells}$
- Generate constraints: 基于集合的包含关系
  - *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∈*Cells*
  - **X* = *Y*: $c \in [\![X]\!] \Rightarrow [\![Y]\!] \subseteq [\![c]\!]$ for each c∈*Cells*
  - *X* = `null`: (no constraints)

10

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

11

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`}

12

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;
```

- alloc-1 ∈ ⟦p⟧
- ⟦y⟧ ⊆ ⟦x⟧
- ⟦z⟧ ⊆ ⟦x⟧
- c ∈ ⟦p⟧ ⇒ ⟦z⟧ ⊆ ⟦α⟧ for each c∈*Cells*
- ⟦q⟧ ⊆ ⟦p⟧
- y ∈ ⟦q⟧
- c ∈ ⟦p⟧ ⇒ ⟦α⟧ ⊆ ⟦x⟧ for each c∈*Cells*
- z ∈ ⟦p⟧

Smallest solution:

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

pt(q) = { y}

Agenda

- Introduction to points-to analysis
- Andersen's analysis
- **Steensgaards's analysis**
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

Steensgaard's Analysis

- View assignments as being bidirectional
- Generate constraints: 基于类型及其等价关系
  - *X* = alloc *P*: alloc-*i* ∈ ⟦*X*⟧
  - *X* = &*Y*: *Y* ∈ ⟦*X*⟧
  - *X* = *Y*: ⟦*X*⟧ = ⟦*Y*⟧
  - *X* = **Y*: c ∈ ⟦*Y*⟧ ⇒ ⟦c⟧ = ⟦*X*⟧ for each c∈*Cells*
  - **X* = *Y*: c ∈ ⟦*X*⟧ ⇒ ⟦*Y*⟧ = ⟦c⟧ for each c∈*Cells*
- Extra constraints:

  $c_1, c_2 \in$ ⟦c⟧ ⇒ ⟦c_1⟧ = ⟦c_2⟧ and ⟦c_1⟧ ∩ ⟦c_2⟧ ≠ ∅ ⇒ ⟦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 ∈ ⟦*Y*⟧ ⇒ ⟦*X*⟧ = ⟦*Y*⟧ for each c∈*Cells* (avoids unifying if *Y* is never a pointer)

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;
```

- alloc-1 ∈ ⟦p⟧
- ⟦y⟧ = ⟦x⟧
- ⟦z⟧ = ⟦x⟧
- α ∈ ⟦p⟧ ⇒ ⟦z⟧ = ⟦α⟧
- ⟦q⟧ = ⟦p⟧
- y ∈ ⟦q⟧
- α ∈ ⟦p⟧ ⇒ ⟦α⟧ = ⟦x⟧
- z ∈ ⟦p⟧
- \+ the extra constraints

Smallest solution:

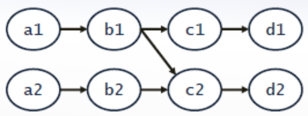
pt(p) = { alloc-1, y, z}

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

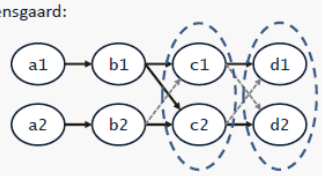
Another Example

```
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* = alloc *P*: ⟦*X*⟧ = &⟦*P*⟧
  - *X* = &*Y*: ⟦*X*⟧ = &⟦*Y*⟧
  - *X* = *Y*: ⟦*X*⟧ = ⟦*Y*⟧
  - *X* = **Y*: &⟦*X*⟧ = ⟦*Y*⟧
  - **X* = *Y*: ⟦*X*⟧ = &⟦*Y*⟧
- Implicit extra constraint for term equality:

  $\&t_1 = \&t_2 \Rightarrow t_1 = t_2$

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

Agenda

- Introduction to points-to analysis
- Andersen's analysis
- Steensgaards's analysis
- **Interprocedural points-to analysis**
- Null pointer analysis
- Flow-sensitive points-to 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

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

- 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,…,$n \wedge [\![z]\!] \subseteq [\![x]\!])$
- (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
  $f(x_1, \ldots, x_n)$ { … return *z*; }
  add these constraints:

  $f \in [\![f]\!]$
  $f \in [\![y]\!] \Rightarrow ([\![a_i]\!] = [\![x_i]\!]$ 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

- Generalize the abstract domain $Cells \rightarrow 2^{Cells}$ to
  $Contexts \rightarrow Cells \rightarrow 2^{Cells}$
  (or equivalently: $Cells \times Contexts \rightarrow 2^{Cells}$)
  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 for each function (like the use of lifted lattices earlier):
  $Contexts \rightarrow \text{lift}(Cells \rightarrow 2^{Cells})$
- Does this still fit into the cubic 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?

25

Context-sensitive Pointer Analysis

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

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

Are x and y aliases?

26

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

27

Agenda

- Introduction to points-to analysis
- Andersen's analysis
- Steensgaards's analysis
- Interprocedural points-to analysis
- **Null pointer analysis**
- Flow-sensitive points-to analysis

28

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)

29

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:
  $Cells \rightarrow Null$

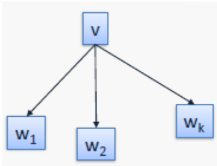
30

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⟧= *???*
- For all other CFG nodes:
  - ⟦v⟧= *JOIN*(v)

where *P* is null or an integer constant

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*: ⟦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*: ⟦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();
*a = alloc null; // alloc-2
n = null;
*b = n; // strong update here would be unsound!
c = *a;
```

is c 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 = `alloc` P: ⟦v⟧ = *JOIN*(v)[X ↦ NN, `alloc-i` ↦ ?]
- X = &Y: ⟦v⟧ = *JOIN*(v)[X ↦ NN]
- X = Y: ⟦v⟧ = *JOIN*(v)[X ↦ *JOIN*(v)(Y)]
- X = `null`: ⟦v⟧ = *JOIN*(v)[X ↦ ?]

could be improved...

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

37

Strong and Weak Updates, Revisited

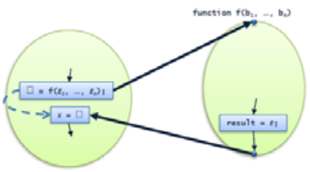
- Strong update: σ[*c*↦*new-value*]
  - possible if *c* is known to refer to a single concrete cell
  - works for assignments to local variables (as long as TIP doesn't have e.g. nested functions)
- Weak update: σ[*c*↦ σ(*c*) ⊔*new-value*]
  - necessary if *c* may refer to multiple concrete cells
  - bad for precision, we lose some of the power of flow-sensitivity
  - required for assignments to heap cells (unless we extend the analysis abstraction!)

38

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



39

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

40

Example Program & Constraints

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

Andersen generates:

pt(p) = {alloc-1}

pt(q) = {p}

pt(n) = Ø

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

41

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 may be null
- Due to the weak updates for all heap store operations, precision is bad for alloc-i cells

42

Agenda

- Introduction to points-to analysis
- Andersen's analysis
- Steensgaards's analysis
- Interprocedural points-to analysis
- Null pointer analysis
- **Flow-sensitive points-to analysis**

43

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 $2^{Cells \times Cells}$ ordered under subset inclusion (or alternatively, $Cells \rightarrow 2^{Cells}$)
- For every CFG node, v, we introduce a constraint variable ⟦v⟧ describing the state *after* v
- Intraprocedural analysis (i.e. ignore function calls)

44

Constraints

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

45

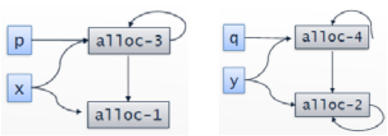
Example Program

```
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;
}
```

46

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:

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

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

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

48

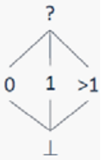
Improving Precision with Abstract Counting

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

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

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

Count =

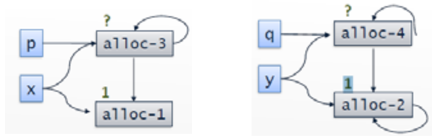


- This permits **strong updates** on those that describe precisely 1 concrete cell

49

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

50

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;
}
```

51