Pointer analysis

Outline:
- What is pointer analysis
- Intraprocedural pointer analysis
- Interprocedural pointer analysis
  - Andersen and Steensgaard

Aliases: two expressions that denote the same memory location.

- Aliases are introduced by:
  - pointers
  - call-by-reference
  - array indexing
  - C unions

Useful for what?

- Improve the precision of analyses that require knowing what is modified or referenced (e.g., const prop, CSE ...)
- Eliminate redundant loads/stores and dead stores.
  
  ```
  x := *p;
  ...
  y := *p; // replace with y := x?
  ```

- Parallelization of code
  - can recursive calls to quick_sort be run in parallel? Yes, provided that they reference distinct regions of the array.
- Identify objects to be tracked in error detection tools
  ```
  x.lock();
  ...
  y.unlock(); // same object as x?
  ```

Kinds of alias information

- Points-to information (must or may versions)
  - at program point, compute a set of pairs of the form p ! x, where p points to x.
  - can represent this information in a points-to graph

- Alias pairs
  - at each program point, compute the set of all pairs (e₁, e₂) where e₁ and e₂ must/may reference the same memory.

- Storage shape analysis
  - at each program point, compute an abstract description of the pointer structure.
Flow functions

\[ F_{x := k}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

\[ F_{x := a + b}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

Flow functions

\[ F_{x := k}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

\[ F_{x := a + b}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

Flow functions

\[ F_{x := &y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

\[ F_{x := \ast y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

Flow functions

\[ F_{x := \ast y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

\[ F_{x := y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

Flow functions

\[ F_{x := \ast y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

\[ F_{x := y}^{x \rightarrow y}(\text{in}) = \text{in} - \{ x \rightarrow y \} \]

Intraprocedural Points-to Analysis

- Flow functions:
  - \( kill(x) = \bigcup_{v \in \text{vars}} \{ (x, v) \} \)
  - \( F_{x := \text{new}(S)} = S - \text{kill}(x) \)
  - \( F_{x := \text{new}}(S) = S - \text{kill}(x) \)
  - \( F_{x := \text{new}}(S) = S - \text{kill}(x) \)
  - \( F_{x := \text{new}}(S) = S - \text{kill}(x) \)
  - \( F_{x := \text{new}}(S) = S - \text{kill}(x) \)
  - \( F_{x := \text{new}}(S) = S - \text{kill}(x) \)

Pointers to dynamically-allocated memory

- Handle statements of the form: \( x := \text{new} T \)
- One idea: generate a new variable each time the
new statement is analyzed to stand for the new
location:
  \[ F_{x := \text{new}} T(S) = S - \text{kill}(x) \cup \{(x, \text{newvar}())\} \]

Example

\[ l := \text{new Cons} \]
\[ p := l \]
\[ t := \text{new Cons} \]
\[ *p := t \]
\[ p := t \]
Example solved

What went wrong?

- Lattice infinitely tall!
- We were essentially running the program
- Instead, we need to summarize the infinitely many allocated objects in a finite way
- New Idea: introduce summary nodes, which will stand for a whole class of allocated objects.

What went wrong?

- Example: For each new statement with label $L$, introduce a summary node $loc_L$, which stands for the memory allocated by statement $L$.

$$F_L: x:=\text{new } T(S) = S - \text{kill}(x) \cup \{(x, loc_L)\}$$

- Summary nodes can use other criterion for merging.

Example revisited

Example revisited & solved

Example revisited & solved
Array aliasing, and pointers to arrays

- Array indexing can cause aliasing:
  - \( a[i] \) aliases \( b[j] \) if:
    - \( a \) aliases \( b \) and \( i = j \)
    - \( a \) and \( b \) overlap, and \( i = j + k \), where \( k \) is the amount of overlap.

- Can have pointers to elements of an array
  - \( p := &a[i]; \ldots; p++ \);

- How can arrays be modeled?
  - Could treat the whole array as one location.
  - Could try to reason about the array index expressions: array dependence analysis.

Summary

- We just saw:
  - intraprocedural points-to analysis
  - handling dynamically allocated memory
  - handling pointers to arrays

- But, intraprocedural pointer analysis is not enough.
  - Sharing data structures across multiple procedures is one the big benefits of pointers: instead of passing the whole data structures around, just pass pointers to them (eg C pass by reference).
  - So pointers end up pointing to structures shared across procedures.
  - If you don’t do an interproc analysis, you’ll have to make conservative assumptions functions entries and function calls.

Fields

- Can summarize fields using per-field summary
  - for each field \( F \), keep a points-to node called \( F \) that summarizes all possible values that can ever be stored in \( F \)

- Can also use allocation sites
  - for each field \( F \), and each allocation site \( S \), keep a points-to node called \( (F, S) \) that summarizes all possible values that can ever be stored in the field \( F \) of objects allocated at site \( S \).

Conservative approximation on entry

- Say we don’t have interprocedural pointer analysis.
- What should the information be at the input of the following procedure:
  ```
  global g;
  void p(x,y) {
    ...
  }
  ```

- They are all very conservative!
- We can try to do better.

Interprocedural pointer analysis

- Main difficulty in performing interprocedural pointer analysis is scaling

- One can use a top-down summary based approach (Wilson & Lam 95), but even these are hard to scale.
Example revisited

- Cost:
  - space: store one fact at each prog point
  - time: iteration

Iter 1
Iter 2
Iter 3

S1: \( l := \text{new Cons} \)
\( p := l \)

S2: \( t := \text{new Cons} \)
\( *p := t \)
\( p := t \)

New idea: store one dataflow fact

- Store one dataflow fact for the whole program
- Each statement updates this one dataflow fact
  - use the previous flow functions, but now they take the whole program dataflow fact, and return an updated version of it.
- Process each statement once, ignoring the order of the statements
- This is called a flow-insensitive analysis.

Flow insensitive pointer analysis

Flow sensitive vs. insensitive

What went wrong?

- What happened to the link between \( p \) and \( S1 \)?
  - Can’t do strong updates anymore!
  - Need to remove all the kill sets from the flow functions.
- What happened to the self loop on \( S2 \)?
  - We still have to iterate!
Flow insensitive pointer analysis: fixed

S1: \( l := \text{new Cons} \)
\( p := l \)

S2: \( t := \text{new Cons} \)
\( \ast p := t \)
\( p := t \)

This is Andersen’s algorithm ’94

Flow insensitive loss of precision

- Flow insensitive analysis leads to loss of precision!

```java
main()
    \( x := \& y; \)
    ...
    \( x := \& z; \)
```

- However:
  - uses less memory (memory can be a big bottleneck to running on large programs)
  - runs faster

Flow sensitive vs. insensitive, again

<table>
<thead>
<tr>
<th>Flow-sensitive Soln</th>
<th>Flow-insensitive Soln</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p := l )</td>
<td>( p := l )</td>
</tr>
<tr>
<td>( t := \text{new Cons} )</td>
<td>( t := \text{new Cons} )</td>
</tr>
<tr>
<td>( \ast p := t )</td>
<td>( \ast p := t )</td>
</tr>
<tr>
<td>( p := t )</td>
<td>( p := t )</td>
</tr>
</tbody>
</table>

In Class Exercise!

| S1: \( p := \text{new Cons} \) |
| S2: \( q := \text{new Cons} \) |
| \( \ast p = q \) |
| \( x = q \) |
| \( \ast q = p \) |
| \( s = q \) |
| \( \ast x = s \) |

In Class Exercise! solved

| S1: \( p := \text{new Cons} \) |
| S2: \( q := \text{new Cons} \) |
| \( \ast p = q \) |
| \( x = q \) |
| \( \ast q = p \) |
| \( s = q \) |
| \( \ast x = s \) |
Worst case complexity of Andersen

Worst case: \(N^2\) per statement, so at least \(N^3\) for the whole program. Andersen is in fact \(O(N^3)\)

New idea: one successor per node

• Make each node have only one successor.
• This is an invariant that we want to maintain.

More general case for \(*x = y\)

Handling: \(x = *y\)
Handling: $x = y$ (what about $y = x$?)

Handling: $x = \&y$

Handling: $x = \&y$

Handling: $x = y$ (what about $y = x$?)

get the same for $y = x$

Handling: $x = \&y$

Handling: $x = \&y$

Handling: $x = y$ (what about $y = x$?)

Our favorite example, once more!

Flow insensitive loss of precision

Another example

```c
bar() {
    i := &a;
    j := &b;
    foo(&i);
    foo(&j);
    // i points to what?
    *i := ...;
}

void foo(int* p) {
    printf("%d",*p);
}
```
Another example

```c
bar() {
    i := &a;
    j := &b;
    foo(i);
    foo(j);
    // i points to what?
    *i := ...
}

void foo(int* p) {
    printf("%d", *p);
}
```

Almost linear time

- Time complexity: $O(\alpha(N, N))$

- So slow-growing, it is basically linear in practice
- For the curious: node merging implemented using UNION-FIND structure, which allows set union with amortized cost of $O(\alpha(N, N))$ per op. Take CSE 202 to learn more!

In Class Exercise!

```
S1: p := new Cons
*S1 = q
S2: q := new Cons
*S2 = r
S = p
*S1 = r
*S2 = p
```

Advanced Pointer Analysis

- Combine flow-sensitive/flow-insensitive
- Clever data-structure design
- Context-sensitivity