Pointer Analysis

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

Useful for what?

- Improve the precision of analyses that require knowing what is modified or referenced (eg const prop, CSE ...)
- Eliminate redundant loads/stores and dead stores.
  \[ x := p; \quad \ldots \quad y := p; \quad \text{// is } x \text{ dead?} \]
- 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
  \[ x.\text{lock}(); \quad \ldots \quad y.\text{unlock}(); \quad \text{// same object as } x? \]

Kinds of alias information

- Points-to information (must or may versions)
  - at program point, compute a set of pairs of the form \( p \rightarrow x \), 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 \( (e_1, e_2) \) where \( e_1 \) and \( e_2 \) must/may reference the same memory.

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

Intraprocedural Points-to Analysis

- Want to compute may-points-to information

- Lattice:
  \[ D = 2 = \{ \text{\text{v}} \text{\text{v}} \} \]
  \[ L = \{ \} \]
  \[ T = \{ \text{\text{v}} \text{\text{v}} \} \]
Flow functions

\[
\begin{align*}
F_{x := \texttt{a + b}}(\text{in}) &= \quad \text{in} \\
\quad x := \texttt{a + b} &\quad \text{out} \\
F_{x := \texttt{a + b}}(\text{out}) &= \\
\end{align*}
\]

Flow functions

\[
\begin{align*}
F_{x := \texttt{k}}(\text{in}) &= \quad \text{in} \\
\quad x := \texttt{k} &\quad \text{out} \\
F_{x := \texttt{k}}(\text{out}) &= \\
\end{align*}
\]

Flow functions

\[
\begin{align*}
F_{x := \texttt{&y}}(\text{in}) &= \quad \text{in} \\
\quad x := \texttt{&y} &\quad \text{out} \\
F_{x := \texttt{&y}}(\text{out}) &= \\
\end{align*}
\]

Flow functions

\[
\begin{align*}
F_{*x := \texttt{y}}(\text{in}) &= \quad \text{in} \\
\quad *x := \texttt{y} &\quad \text{out} \\
F_{*x := \texttt{y}}(\text{out}) &= \\
\end{align*}
\]

Intraprocedural Points-to Analysis

- Flow functions:
  \[
  \begin{align*}
  \text{kill}(x) &= \bigcup_{v \in \text{vars}(x)} \{(c, v)\} \\
  F_{x := \texttt{y}}(S) &= S \setminus \text{kill}(x) \\
  F_{x := \texttt{&y}}(S) &= S \setminus \text{kill}(x) \cup \{(x, v) \mid (y, v) \in S\} \\
  F_{x := \texttt{y}}(S) &= S \setminus \text{kill}(x) \cup \{(x, y)\} \\
  F_{x := \texttt{&y}}(S) &= S \setminus \text{kill}(x) \cup \{(x, y)\} \cup \{(y, t) \mid (y, t) \in S \land (t, v) \in S\} \\
  F_{x := \texttt{x}}(S) &= \text{let } V := \{v \mid (x, v) \in S\} \text{ in} \\
  &\quad \text{if } V = \emptyset \text{ then } \text{kill}(v) \text{ else } 0 \text{ end} \\
  &\quad \cup \{(v, t) \mid v \in V \land (y, t) \in S\}
  \end{align*}
  \]

Points to dynamically-allocated memory

- Handle statements of the form: \(x := \texttt{new } T\)
- One idea: generate a new variable each time the new statement is analyzed to stand for the new location:

\[
F_{x := \texttt{new } T}(S) = S \setminus \text{kill}(x) \cup \{(x, \text{newvar}())\}
\]

Example

\[
\begin{align*}
&1 := \texttt{new Cons} \\
p := 1 \\
&t := \texttt{new Cons} \\
*\texttt{p} := t \\
p := t
\end{align*}
\]
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.

---

**Example revisited**

Example solved

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

Summary nodes can use other criterion for merging.

---

**Example revisited & 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.
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.

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 (e.g., 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.

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

```
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 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
Flow insensitive pointer analysis: fixed

This is Andersen’s algorithm ’94

Final result

Iter 1

Iter 2

Iter 3

Flow sensitive vs. insensitive, again

Final result

Flow-sensitive Soln

Flow-insensitive Soln

Flow insensitive loss of precision

• Flow insensitive analysis leads to loss of precision!

```c
main() {
  x := &y;
  ...
  x := &z;
}
```

• Flow insensitive analysis tells us that x may point to z here!

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

In Class Exercise!

```c
S1: p := new Cons
S2: q := new Cons

*p = q

x = &q

*q = x

*z = 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 $y = x$?)
Our favorite example, once more!

1. \( l := \text{new Cons} \)
2. \( p := l \)
3. \( t := \text{new Cons} \)
4. \( *p := t \)
5. \( p := t \)

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

Handling: \( x = y \)

Our favorite example, once more!

Flow insensitive loss of precision

Another example

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

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

Another example

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

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

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

- 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!

In Class Exercise! solved

Advanced Pointer Analysis

- Combine flow-sensitive/flow-insensitive

- Clever data-structure design

- Context-sensitivity