**Pointer Analysis**

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

**Pointer and Alias Analysis**

- 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.
  - Points to information
  - Alias pairs
  - Storage shape analysis

**Intraprocedural Points-to Analysis**

- Want to compute may-points-to information
  - Lattice:
Flow functions

1. \( x := a + b \) (in) = \( F_{x \rightarrow y}(\text{in}) = \)

2. \( x := k \) (in) = \( F_{x \rightarrow y}(\text{in}) = \)

3. \( x := \& y \) (in) = \( F_{x \rightarrow y}(\text{in}) = \)

4. \( *x := y \) (in) = \( F_{x \rightarrow y}(\text{in}) = \)

Intraprocedural Points-to Analysis

- Flow functions:
  
  \[
  \begin{align*}
  \text{killed(x)} &= S - \text{kill}(x) \\
  F_{x:=k}(S) &= S - \text{kill}(x) \\
  F_{x:=\&y}(S) &= S - \text{kill}(x) \cup \{(x, v) \mid (y, v) \in S\} \\
  F_{x:=y}(S) &= S - \text{kill}(x) \cup \{(x, v) \mid \forall t \in \text{vars} . ((y, t) \in S \land (t, v) \in S)\} \\
  F_{x:=k}(S) &= \text{let } V := \{v \mid (x, v) \in S\} \text{ in } S - \text{if } V = \{x\} \text{ then } \text{kill}(v) \text{ else } 0 \\
  &\cup \{(x, t) \mid t \in V \times (y, t) \in S\}
  \end{align*}
  \]

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:

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

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

- Summary nodes can use other criterion for merging.

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 (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.

Conservative approximation on entry

- Say we don’t have interprocedural pointer analysis.
- What should the information be at the input of the following procedure:

  ```c
  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

```
  S1: l := new Cons
  p := l

  S2: t := 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 insensitive vs. insensitive

Flow insensitive pointer analysis: fixed

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

This is Andersen’s algorithm ’94

S1: l := new Cons

\[ p := l \]

S2: t := new Cons

\[ *p := t \]

\[ p := t \]

Flow insensitive vs. insensitive, again

S1: l := new Cons

\[ p := l \]

S2: t := new Cons

\[ *p := t \]

\[ p := t \]

Flow sensitive vs. insensitive, again

Flow insensitive loss of precision

• Flow insensitive analysis leads to loss of precision!

```plaintext
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!

S1: p := new Cons

\[ *p = q \]

S2: q := new Cons

\[ r = &q \]

\[ *q = r \]

\[ s = p \]

\[ s = r \]

\[ *r = s \]

\[ *q = p \]

\[ *x = y \]

In Class Exercise! solved

S1: p := new Cons

S2: q := new Cons

\[ *p = q \]

\[ x := &y; \]

\[ *x := &z; \]

\[ x := &z; \]

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

Handling: $x = y$

Flow insensitive loss of precision

Another example

```c
bar() {
  i := &a;
  j := &b;
  foo(i); // 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!

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