Pointer analysis

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

Aliases:
- 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 (eg const prop, CSE …)
- Eliminate redundant loads/stores and dead stores.
  \[ x := *p; \quad *x := \ldots \quad \text{is \( x \) dead?} \]
  \[ y := *p; \quad \text{// 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.\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 \( p \mapsto x \), 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.

Intraprocedural Points-to Analysis
- Want to compute may-points-to information

- Lattice:
  \[ \mathcal{D} = \mathbb{2} \{ (x, y) | x \in \mathcal{V}_a, y \in \mathcal{V}_e \} \]
  \[ \cup = \cup \]
  \[ \subseteq = \subseteq \]
  \[ \cap = \emptyset \]
  \[ T = \{ (x, y) | x \in \mathcal{V}_a, y \in \mathcal{V}_e \} \]
Flow functions

\[ F_{x \leftarrow k}(in) = \{ x \rightarrow k \} \]

\[ F_{x \leftarrow a + b}(in) = \]

Intraprocedural Points-to Analysis

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

Example

```
\begin{figure}
\begin{center}
\begin{tikzpicture}
\node (l) [label=left:5] {\texttt{5} := \text{new Cons}};
\node (p) [label=below:1, right of=l] {\texttt{l} := \texttt{p}};
\node (t) [label=below:1, right of=p] {\texttt{t} := \texttt{p}};
\node (x) [label=below:1, right of=t] {\texttt{p} := \texttt{s}};
\node (y) [label=below:1, right of=x] {\texttt{s} := \texttt{t}};
\node (z) [label=below:1, right of=y] {\texttt{t} := \text{null}};
\node (z') [label=below:1, right of=z] {\texttt{t} := \text{null}};
\node (z'') [label=below:1, right of=z'] {\texttt{t} := \text{null}};
\node (z''') [label=below:1, right of=z''] {\texttt{t} := \text{null}};
\end{tikzpicture}
\end{center}
\end{figure}
```
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 solved

1. `l := new Cons`
2. `p := l`
3. `t := new Cons`
4. `*p := t`
5. `p := t`

---

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

1. `l := new Cons`
2. `p := l`
3. `t := new Cons`
4. `*p := t`
5. `p := t`

---

Example revisited & solved

1. `l := new Cons`
2. `p := l`
3. `t := new Cons`
4. `*p := t`
5. `p := t`

---

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:

```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

- Initially:
  - S1: new Cons
  - S2: new Cons

- Iter 1:
  - p = l

- Iter 2:
  - *p = t

- Iter 3:
  - p = t

- In the third iteration, we are ready to compute the summary of `p`.
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: 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

S1: \( l := \text{new Cons} \)
\[ p := l \]
S2: \( t := \text{new Cons} \)
\[ *p := t \]
\[ p := t \]

This is Andersen’s algorithm ’94

Flow sensitive vs. insensitive, again

S1: \( l := \text{new Cons} \)
\[ p := l \]
S2: \( t := \text{new Cons} \)
\[ *p := t \]
\[ p := t \]

Flow insensitive loss of precision

- Flow insensitive analysis leads to loss of precision!

\[
\text{main()} \{ \\
\quad x := &y; \\
\quad \ldots \\
\quad x := &z; \\
\}
\]

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

In Class Exercise!

S1: \( p := \text{new Cons} \)
\[ *p = q \]
S2: \( q := \text{new Cons} \)
\[ *q = r \]
\[ s = p \]
\[ s = r \]
\[ *r = s \]

In Class Exercise! solved

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 $^*x = y$

More general case for $^x = _y$

Handling: $x = _y$

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

$\quad x = y \quad \Rightarrow \quad x = y$

Handling: $x = k y$

$\quad x = k y \quad \Rightarrow \quad x = k y$

Our favorite example, once more!

$S1: l := \text{new Cons}$

$p := l$

$S2: t := \text{new Cons}$

$p := t$

Flow insensitive loss of precision $\forall \alpha \in X$

Flow-sensitive loss of precision

Flow-insensitive based

Flow-insensitive based

Flow-insensitive based

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: $O(N \alpha(N, N))$

- 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!

Advanced Pointer Analysis

- Combine flow-sensitive/flow-insensitive

- Clever data-structure design

- Context-sensitivity

In Class Exercise!

S1: $p := \text{new Cons}$

S2: $q := \text{new Cons}$

*p = q

z = &q

*q = z

s = z

*r = s

*q = p

s = r

*r = s

*p = q

In Class Exercise! solved