**Pointer and Alias Analysis**

**Aliases:**
- two expressions that denote same mutable memory location

Introduced through
- pointers
- call-by-reference
- array indexing
- C unions, Fortran common, equivalence

Applications of alias analysis:
- improved side-effect analysis:
  - if assign to one expression, what other expressions are modified?
  - if certain modified or not modified, not a problem
- if uncertain, things can get ugly
- eliminate redundant loads/stores & dead stores (CSE & dead assign elim, for pointer ops)
- automatic parallelization of code
- manipulating data structures
- ...

**Kinds of alias info**

**Points-to analysis**
- at each program point, calculate set of \( p \rightarrow x \) bindings, if \( p \) points to \( x \)
- two related problems:
  - **may** points-to: \( p \) may point to \( x \)
  - **must** points-to: \( p \) must point to \( x \)

**Storage shape analysis**
- at each program point, calculate an abstract description of the structure of pointers etc.

**Alias-pair analysis**
- at each program point, calculate set of \((expr_1, expr_2)\) pairs, if \( expr_1 \) and \( expr_2 \) reference the same memory
- **may** and **must** alias-pair versions

Points-to analysis is simple
Storage shape analysis more abstract
Alias-pairs analysis more general than points-to analysis, but more complicated

**An intraprocedural points-to analysis**

At each program point, calculate set of \( p \rightarrow x \) bindings, if \( p \) points to \( x \)

Outline:
- define **may** version first, then consider **must** version
- develop algorithm in increasing stages of complexity
- pointers only to scalars
- add pointers to pointers
- add pointers to dynamically-allocated storage
- add pointers to array elements

**May-point-to scalars**

Domain: \( \text{Pow}(\text{Var} \times \text{Var}) \)

Flow functions:

\[
\begin{align*}
p & := s \times \mathbf{x} \\
\text{MAY-PT}_{\text{succ}} & = \text{MAY-PT}_{\text{pred}} - \{ p \rightarrow \} \cup \{ p \rightarrow x \} \\
p & := q \\
\text{MAY-PT}_{\text{succ}} & = \text{MAY-PT}_{\text{pred}} - \{ p \rightarrow \} \cup \{ p \rightarrow t | q \rightarrow t \in \text{MAY-PT}_{\text{pred}} \}
\end{align*}
\]

Meet function: union
**Must-point-to**

How to define must-point-to analysis?

Option 1: analogous to may-point-to, but as must problem
  - e.g. intersection is meet operation

Option 2: interpretation of may-point-to results
  - if \( p \) may point to only \( x \), then \( p \) must point to \( x \):

  \[
  \text{MUST-PT} = \{ p \rightarrow x \mid p \rightarrow x \in \text{MAY-PT} \text{ and } \exists y \text{ s.t. } p \rightarrow y \in \text{MAY-PT} \Rightarrow y = x \}
  \]

  - what if \( p \) points to nil? \( p \) assigned an integer?

**Using alias info**

E.g. reaching definitions

At each program point, calculate set of \( s:x \) bindings, if \( x \) might get its definition from stmt \( s \)

Simple flow functions:

\[
s: *p := x
\]

\[
\text{RD}_{\text{succ}} = \text{RD}_{\text{pred}} - \{ *: z \mid p \rightarrow z \in \text{MUST-PT}_{\text{pred}} \}
\]

\[
\bigcup
\{ *: x \mid p \rightarrow z \in \text{MAY-PT}_{\text{pred}} \}
\]

\[
s: x := *p
\]

\[
\text{RD}_{\text{succ}} = \text{RD}_{\text{pred}} - \{ *: x \}
\]

\[
\bigcup
\{ s: x \}
\]

**Reaching “right hand sides”**

A variation on reaching definitions
  that passes definitions through copies

\( s:x \) in set if \( x \) might get its definition from rhs of stmt \( s \),
  skipping through uninteresting copies and pointer loads
  where possible

Can use reaching right-hand sides to construct def/use chains
  that skip through copies, e.g. for better constant propagation

Flow functions:

\[
s: x := y
\]

\[
\text{RD}_{\text{succ}} = \text{RD}_{\text{pred}} - \{ *: x \}
\]

\[
\bigcup
\{ s': x \mid s': y \in \text{RD}_{\text{pred}} \}
\]

\[
s: x := *p
\]

\[
\text{RD}_{\text{succ}} = \text{RD}_{\text{pred}} - \{ *: x \}
\]

\[
\bigcup
\{ s': x \mid p \rightarrow z \in \text{MAY-PT}_{\text{pred}} \land
   \exists s'' \text{ s.t. } p \rightarrow z \in \text{MAY-PT}_{\text{pred}} \land
   s'' \rightarrow z \in \text{RD}_{\text{pred}} \}
\]

**Example**

1. \( x := 3 \)
2. \( p := 4x \)
3. \( y := 5 \)
4. \( q := 4y \)
5. \( q := 4x \)
6. \( *p := 7 \)
7. \( z := *q \)
8. \( *q := 4 \)
9. \( w := *p \)
Adding pointers to pointers

Flow functions:

\[
p := *q
\]
\[
\text{MAY-PT}_{\text{succ}} = \text{MAY-PT}_{\text{pred}} - \{ p \rightarrow * \} \cup \\
\{ p \rightarrow t \mid q \rightarrow r \in \text{MAY-PT}_{\text{pred}} \land \\
r \rightarrow t \in \text{MAY-PT}_{\text{pred}} \}
\]
\[
*p := q
\]
\[
\text{MAY-PT}_{\text{succ}} = \text{MAY-PT}_{\text{pred}} - \{ r \rightarrow * \mid p \rightarrow r \in \text{MUST-PT}_{\text{pred}} \} \\
\cup \{ r \rightarrow t \mid p \rightarrow r \in \text{MAY-PT}_{\text{pred}} \land \\
q \rightarrow t \in \text{MAY-PT}_{\text{pred}} \}
\]

Example

```
int x, y, z;
int *p, *q;
int **R;
```

```
x := 5
y := 6
p := &x
q := &y
R := &p
```

```
*p := q
```

```
*x := 8
```

```
*y := 6
```

```
*z := *p
```

Adding pointers to dynamically-allocated memory

```
p := new T
```

Issue: each execution creates a new location

Idea: generate new var to stand for new location
- make Var domain unbounded
- newvar: return next unused element of Var

Flow function:

\[
s: p := new T
\]
\[
\text{MAY-PT}_{\text{succ}} = \text{MAY-PT}_{\text{pred}} - \{ p \rightarrow * \} \cup \{ p \rightarrow \text{newvar} \}
\]

Example

```
① lst := new Cons
② p := lst
③ t := new Cons
④ *p := new Cons
⑤ p := t
⑥ p := t
```

```
```
A monotonic, finite approximation

Can’t create a new variable each time analyze statement
• lattice is infinitely tall if Var domain is infinite!
• not a monotonic flow function!

One solution:
   create a special summary node for each new stmt

Domain = Pow((Var+Stmt) × (Var+Stmt))

\[
\begin{align*}
    s: p &:= \text{new } T \\
    \text{MAY-PT}_{\text{succ}} &\Rightarrow \text{MAY-PT}_{\text{pred}} - \{p\rightarrow *\} \cup \{p\rightarrow \text{loc}_s\}
\end{align*}
\]

Alternatives:
• summary node for each type \( T \)
• \( k \)-limited summary
  • maintain distinct nodes up to \( k \) links removed from root vars, then merge together
• ...

Adding pointers to array elements

Array index expressions can generate aliases
\( a[i] \) aliases \( b[j] \) if:
• \( a \) aliases \( b \) and \( i \) equals \( j \)
• \( a \) and \( b \) overlap, and ...

Can have pointers to array elements:
\( p := &a[i] \)

Can have pointer arithmetic, for array addressing:
\( p := &a[0]; \ldots; p++ \)

How to model arrays?
• could treat whole array as big monolithic location
• could try to reason about array index expressions
  ⇒ array dependence analysis (later)