**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 \( x \)

Alias-pair analysis
- at each program point, calculate set of \( \langle \text{expr}_1, \text{expr}_2 \rangle \) pairs, if \( \text{expr}_1 \) and \( \text{expr}_2 \) reference the same memory
- **may** and **must** alias-pair versions

Storage shape analysis
- at each program point, calculate an abstract description of the structure of pointers etc., e.g. list-like, or tree-like, or DAG-like, or ...

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

**A 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 vars of scalar type
  - add pointers to pointers
  - add pointers to and from structures
  - add pointers to dynamically-allocated storage
  - add pointers to array elements

**May-point-to scalars**

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

Forward flow functions:

\[
\begin{align*}
\text{PT}_p := \lambda x. (\text{in} - \{ p \rightarrow \star \} \cup \{ p \rightarrow x \}) \\
\text{PT}_p := q.(\text{in} - \{ p \rightarrow \star \} \cup \{ p \rightarrow v \mid q \rightarrow v \in \text{in} \})
\end{align*}
\]

Meet function: union

What about \( p := \text{nil} \)?
**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 \( x \):
  
  \[
  \text{must-point-to}(p) = \{ x | \{ x \} = \text{may-point-to}(p) \}
  \]

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

**Example**

\[
\begin{align*}
1 & \ x := 3 \\
2 & \ p := \&x \\
3 & \ y := 5 \\
4 & \ q := \&y \\
5 & \ q := \&x \\
6 & \ *p := 7 \\
7 & \ z := *q \\
8 & \ *q := 4 \\
9 & \ w := *p
\end{align*}
\]

**Using alias info**

E.g. reaching definitions

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

Simple flow functions:

\[
\text{RD}_{x \mapsto s}(\ldots) = \text{in} - (x\mapsto*) \cup \{x\mapsto s\}
\]

\[
\text{RD}_{*p \mapsto \ldots}(\text{in}) = \text{in} - (x\mapsto*) \cup \{x\mapsto s| \forall x \in \text{must-point-to}(p)\} \\
\cup \{x\mapsto s| \forall x \in \text{may-point-to}(p)\}
\]

**Reaching “right hand sides”**

A variation on reaching definitions  
that passes definitions through copies  
\( x\mapsto s \) 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/uea chains  
that skip through copies, e.g. for better constant propagation

Additional flow functions:

\[
\text{RD}_{x \mapsto y}(\text{in}) = \text{in} - (x\mapsto*) \cup (x\mapsto s'| y\mapsto s' \in \text{in})
\]

\[
\text{RD}_{*p \mapsto \text{in}}(\text{in}) = \text{in} - (x\mapsto*) \cup (x\mapsto s'| p\mapsto y \in \text{may-point-to}(p) \wedge y\mapsto s' \in \text{in})
\]
Another use: "scalar replacement"

If we know that a pointer expression $*p$ aliases a variable $x$ ($p$ must point to $x$) at some point, then can replace $*p$ with $x$
- both for load & store

Load part also known as "redundant load elimination"

Adding pointers to pointers

Now allow a pointer to point to a pointer
- loads may return pointers, stores may store pointers

New flow functions:
\[
\text{PT}_p : \ L = \text{pt}(\text{in} - \{p \to *\} \cup \{p \to v \mid v \in \text{in} \land z \to v \in \text{in})}
\]

\[
\text{PT}_p : \ L = \text{pt}(\text{in} - \{z \to *\} \cup \{z \in \text{in})}
\]

Adding pointers to structs/records/objects/...

A variable can be a structure with a collection of named fields
- a pointer can point to a field of a structure variable
- a field can hold a pointer

Introduce location domain: $\text{Loc} = \text{Var} + \text{Loc} \times \text{Field}$
- either a variable or a location followed by a field name

Old PT domain: sets of $v_1 \to v_2$ pairs = $\text{Pow} \times \text{Var} \times \text{Var}$
New PT domain: sets of $l_1 \to l_2$ pairs = $\text{Pow} \times \text{Loc} \times \text{Loc}$

Some new forward flow functions:
\[
\text{PT}_p : \ y \times \text{f} (\text{in}) = \text{pt} (\{p \to *\} \cup \{p \to x \times \text{f}\})
\]
\[
\text{PT}_p : \ y \times \text{f} (\text{in}) = \text{pt} (\{p \to *\} \cup \{p \to x \times \text{f}\})
\]
\[
\text{PT}_p : \ y \times \text{f} (\text{in}) = \text{pt} (\{p \to *\} \cup \{p \to x \times \text{f}\})
\]
\[
\text{PT}_p : \ y \times \text{f} (\text{in}) = \text{pt} (\{p \to *\} \cup \{p \to x \times \text{f}\})
\]
Adding pointers to dynamically-allocated memory

\[ p := \text{new } T \]

- \( T \) could be scalar, pointer, structure, ...

Issue: each execution creates a new location

Idea: generate new var of type \( T \) to stand for new location

- make Var domain unbounded
- \textit{newvar}: return next unused element of \textit{Var}

Flow function:
\[ P_{p} := \text{new } T \in = \in - \{ p \rightarrow * \} \cup \{ p \rightarrow \text{newvar} \} \]

Example

- \( \text{lst} := \text{new Cons} \)
- \( p := \text{lst} \)

- \( t := \text{new Cons} \)
- \( (*p).\text{next} := t \)
- \( p := t \)

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 \textit{summary} node for each \textit{new stmt}

- \( \text{Loc} = \text{Var} + \text{Stmt} + \text{Loc} \times \text{Field} \)

Fixed flow function:
\[ P_{s,p} := \text{new } T \in = \in - \{ p \rightarrow * \} \cup \{ p \rightarrow s \} \]

Summary nodes abstract a set of possible locations

- cannot strongly update a summary node

- \[ P_{s,p} := q \in = \in - \{ p \rightarrow * \} \cup \{ p \rightarrow q \} \]

Alternative summarization strategies:

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

Adding pointers to array elements

Array index expressions can generate aliases:
- \( a[i] \) aliases \( b[j] \):
  - \( a \) aliases \( b \) and \( i \) equals \( j \)
  - more generally, \( a \) and \( b \) overlap, and \( \&a[i] = \&b[j] \)

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?

Option 1: reason about array index expressions

- array dependence analysis

Option 2: use a summary node to stand for all elements of the array