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 $p \to x$ bindings,
  if $p$ points to $x$
• two variations:
  • may points-to: $p$ might point to $x$
  • must points-to: $p$ definitely points to $x$

Alias-pair analysis
• at each program point, calculate set of $(\text{expr}_1, \text{expr}_2)$
  pairs, if $\text{expr}_1$ and $\text{expr}_2$ reference the same memory
• may and must alias-pair versions
  + can handle aliasing of variables, unlike pts-to analysis
  - potentially infinite number of alias pairs,
    so want the "minimal" set

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 ...

A points-to analysis

At each program point, calculate set of $p \to x$ 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: $\text{Pow}(\text{Var} \times \text{Var})$
• each variable may point to any number of other variables
• may-point-to $PP(x) = \{ X | P \to X \in \text{Soln}(\text{MayPT}, PP) \}$

Forward flow functions:
\[
\text{MayPT}_P : = \delta_X \in \text{in} \cdot (P \to Y) \cup (P \to X)
\]
\[
\text{MayPT}_P : = \chi \in \text{in} \cdot (P \to Y) \cup (P \to X) \cup (P \to Y) \in \text{in}
\]
\[
\text{MayPT}_X : = \chi \in \text{in} \cdot (P \to Y) \quad \text{(assuming $P$ can't point to a ptr)}
\]
\[
\text{MayPT}_X : = \chi \in \text{in} \quad \text{(assuming $P$ can't point to a ptr)}
\]

Meet function: union

What about nil?
Example

```
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
```

Must-point-to

How to define must-point-to analysis?

Option 1: analogous to may-point-to, but as must problem

- meet function: intersection

Option 2: interpretation of may-point-to results

- if \( P \) may point only to \( X \), then \( P \) must point to \( X \), i.e.,

\[
\text{must-point-to}_{P,P}(\mathcal{F}) = \{ X | \{ X \} = \text{may-point-to}_{P,P}(\mathcal{F}) \}
\]

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

Using alias info

E.g. reaching definitions

At each program point, calculate set of \( X \to S \) bindings,
if \( X \) might get its definition from stmt \( S \)

Simple flow functions:

\[
RD_{S:X} := \ldots (in) = in \setminus (X \to *) \cup (X \to S)
\]

\[
RD_{S:*P} := \ldots (in) = in \setminus (X \to *) \setminus \{ X \in \text{must-point-to}(\mathcal{F}) \}
\]

\[
\cup (X \to S | X \in \text{may-point-to}(\mathcal{F}))
\]

Reaching “right hand sides”

A variation on reaching definitions
that skips through trivial copies

\( X \to S \) in set if \( X \) might get its definition from rhs of stmt \( S \),
skipping through trivial variable and pointer copies where possible

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

Additional flow functions:

\[
RD_{S:X} := \gamma (in) = in \setminus (X \to *) \cup (X \to S') \setminus \{ X \to S' \in in \}
\]

\[
RD_{S:*P} := \gamma (in) = in \setminus (X \to *)
\]

\[
\cup (X \to S' | Y \in \text{may-point-to}(\mathcal{F}) \land
\gamma \to S' \in in)
\]

\[
RD_{S:*P} := \gamma (in) = in \setminus (X \to *) \setminus \{ X \in \text{must-point-to}(\mathcal{F}) \}
\]

\[
\cup (X \to S' | X \in \text{may-point-to}(\mathcal{F}) \land
\gamma \to S' \in in)
\]
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

Example:

\[
\begin{align*}
a & := 5 \\
\ldots \\
w & := &a \\
\ldots \\
b & := *w
\end{align*}
\]

Adding pointers to pointers

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

Revised flow functions for loads and stores:

\[
MayPT_{P} := \phi (in) = \text{in} \cdot (P \rightarrow \mathbb{E}) \\
\cup (P \rightarrow X \cup Q \rightarrow R \in \text{in} \land R \rightarrow X \in \text{in})
\]

\[
MayPT_{X,P} := \phi (in) = \text{in} \cdot (X \rightarrow *) \\
\cup (X \rightarrow X \cup P \rightarrow R \in \text{in} \land Q \rightarrow X \in \text{in})
\]

Example

\[
\begin{align*}
\text{int } x, y, z; \\
\text{int } *p, *q; \\
\text{int } **m;
\end{align*}
\]

\[
\begin{align*}
x & := 5 \\
y & := 6 \\
p & := &x \\
q & := &y \\
m & := &p
\end{align*}
\]

\[
\begin{align*}
*p & := m \\
&*q &:= 7 \\
x & := 8
\end{align*}
\]

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: \(Loc = Var \cup Loc \times Field\)
- either a variable or a location followed by a field name

Old PT domain: sets of \(v_{1} \rightarrow v_{2}\) pairs = \(Pow(Var \times Var)\)

New PT domain: sets of \(l_{1} \rightarrow l_{2}\) pairs = \(Pow(Loc \times Loc)\)

Some new forward flow functions:

\[
MayPT_{P} := \phi (in) = \text{in} \cdot (P \rightarrow *) \\
\cup (P \rightarrow X \cup Q \rightarrow R \in \text{in} \land R \rightarrow X \in \text{in})
\]

\[
MayPT_{X,P} := \phi (in) = \text{in} \cdot (X \rightarrow *) \\
\cup (X \rightarrow X \cup P \rightarrow R \in \text{in} \land Q \rightarrow X \in \text{in})
\]

\[
MayPT_{X,F} := \phi (in) = \text{in} \cdot (X \rightarrow *) \\
\cup (X \rightarrow X \cup Q \rightarrow L \in \text{in})
\]

\[
MayPT_{P,F} := \phi (in) = \text{in} \cdot (P \rightarrow *) \\
\cup (P \rightarrow L \cup Q \rightarrow L \in \text{in})
\]

\[
MayPT_{X,F} := \phi (in) = \text{in} \cdot (X \rightarrow *) \\
\cup (X \rightarrow X \cup Q \rightarrow L \in \text{in})
\]

\[
MayPT_{X,F} := \phi (in) = \text{in} \cdot (X \rightarrow *) \\
\cup (X \rightarrow X \cup Q \rightarrow L \in \text{in})
\]
Adding pointers to dynamically-allocated memory

\[ P := \text{new } \tau \]
- \( \tau \) could be scalar, pointer, structure, ...

Issue: each execution of new creates a new location

Idea: introduce new set of possible memory locations: \( \text{Mem} \)

Extend \( \text{Loc} \) to also allow a location to be a \( \text{Mem} \):
\[ \text{Loc} = \text{Var} \cup \text{Mem} \cup \text{LockField} \]

Flow function:
- \( \text{MayPT}(P := \text{new } \tau, \text{in}) = \text{in} \cdot \{ \text{new } \tau \} \cup \{ \tau \text{newvar} \} \)
- \( \text{newvar} \): return next unallocated element of \( \text{Mem} \)

Example

```
1st := new Cons
p := 1st
```

```
t := new Cons
(*p).next := t
p := t
```

A monotonic, finite approximation

Can't allocate a new memory location
each time analyze new statement
- infinite \( \text{Mem} \) \( \Rightarrow \) infinite \( \text{Loc} \) \( \Rightarrow \) infinitely tall \( \text{Pow}(\text{Loc} \times \text{Loc}) \)
- not a monotonic flow function!

One solution:
- create a special summary node for each new stmt
- \( \text{Loc} = \text{Var} \cup \text{Stmt} \cup \text{LockField} \)

Fixed flow function:
- \( \text{MayPT}(S; P := \text{new } \tau, \text{in}) = \text{in} \cdot \{ \text{new } \tau \} \cup \{ \tau \text{newvar} \} \)

Summary nodes represent a set of possible locations
\( \Rightarrow \) cannot strongly update a summary node
- \( \text{MayPT}(S; P := \text{new } \tau, \text{in}) = \text{in} \cdot \{ \text{new } \tau \} \cup \{ \tau \text{newvar} \} \)

Alternative summarization strategies:
- summary node for each type \( \tau \)
- \( k \)-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 \( b[j] \) if:
- \( a \) 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
\( \Rightarrow \) array dependence analysis

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