CS 473: COMPILER DESIGN
Alias Analysis

• So far, our analyses have been about the uses and definitions of variables/temporary registers
• What about memory locations/stack-allocated variables?

```c
int *x = a[1];
*x = 5;
```

We might never mention `a`, but still load from the memory that `a` points to!

• Conservatively, we might need to assume that any two pointers can point to the same memory location ("alias")
• But this will miss out on a lot of optimizations!

```c
p.x = 5;
a[1] = 8;
b = p.x;
p.x = 5;
a[1] = 8;
b = 5; //correct only if `p` and `a` don’t alias
```
Alias Analysis

• So far, our analyses have been about the uses and definitions of variables/temporary registers
• What about memory locations/stack-allocated variables?

\[ p.x = 5; \]
\[ a[1] = 8; \]
\[ b = p.x; \]

\[ p.x = 5; \]
\[ a[1] = 8; \]
\[ b = 5; \] //correct only if p and a don’t alias

• Goal: for every pair of uses of a stack-allocated variable/memory access, determine whether they might refer to the same location
• We have to be conservative to maintain correctness, so the two possible answers are “may alias” or “will never alias”
• The more precise the analysis (i.e., the more things we can tell will never alias), the more optimizations we can do
Type-Based Alias Analysis

- In some languages, references to data of different types will never alias
  - Including Tiger, Java, OCaml... definitely not C!

- This gives us an easy approximation for alias analysis (made a little more complicated by the fact that we don’t have types in IRs):
  1. At typechecking, associate every type with an alias class
  2. When we translate to the tree IR (turning array, record, etc. accesses into memory accesses), mark every load and store with its alias class
  3. Now two accesses may alias if they have the same class, and won’t if they have different classes

- Refinements: different fields of the same record don’t alias, references to different local variables don’t alias with each other or arrays/records, ... (depending on language)
Questions

Top
Flow-Based Alias Analysis

• In some languages, references to data of different types will sometimes alias

```c
int *x; char *y;
y = (char *) x;
```

• And local variables may alias each to other and arrays, records, etc.

```c
int x;
int *y = &x; // now *y = 0 will set x to 0
int a[10];
int *z = a[5]; // now *z = 0 will set a[5] to 0
```

• Even in languages that don’t do this, we can get better alias analysis by tracking pieces of memory (arrays, objects) instead of just types
Flow-Based Alias Analysis

• In some languages, references to data of different types will sometimes alias.

• Even in languages that don’t do this, we can get better alias analysis by tracking pieces of memory (arrays, objects) instead of just types.

• Define an alias class for each allocation (array/record creation, malloc, object creation, etc.).

• Use dataflow analysis to figure out which classes each pointer/reference might refer to at each point in the program.
Flow-Based Alias Analysis

- in and out sets contain triples (t, d, k) meaning “variable t may refer to the kth field/element/offset of allocation d”

- Define gen[n] and kill[n] as follows:

  - Quadruple forms n:  
    | Quadruple | gen[n] | kill[n] |
    |------------|--------|---------|
    | a = b op c | ?      | (a, d, k) for any d, k |
    | a = load b | (a, d, k) for any d, k | (a, d, k) for any d, k |
    | store b, a | Ø      | Ø       |
    | br L       | Ø      | Ø       |
    | br a L1 L2 | Ø      | Ø       |
    | a = f(b₁,…,bₙ) | (a, d, k) for any d, k | (a, d, k) for any d, k |
    | return a   | Ø      | Ø       |
    | a = initArray(...) [d] | (a, d, 0) | (a, d, k) for any d, k |
Flow-Based Alias Analysis

- in and out sets contain triples (t, d, k) meaning “variable t may refer to the kth field/element/offset of allocation d”

- What can a point to after a = b op c?

  a = b + c, b is a pointer, c is an int
  if b can point to (d, i), then a can point to (d, j) for some j

  a = b + c, b is an int, c is a pointer
  if c can point to (d, i), then a can point to (d, j) for some j

  a = b + 4
  if b can point to (d, i), then a can point to (d, i + 4)
Flow-Based Alias Analysis

• in and out sets contain triples (t, d, k) meaning “variable t may refer to the kth field/element/offset of allocation d”

• Quadruple forms n: gen[n]
  a = b + i      (a, d, j + i) for each existing (b, d, j)
  any other special cases we may know about
  a = b + c      (a, d, k) for each existing (b, d, j) or (c, d, j)

• Now we can do our usual dataflow analysis:

  in[n] := \( \bigcup_{n' \in \text{pred}[n]} \text{out}[n'] \)
  out[n] := gen[n] \( \cup \) (in[n] - kill[n])

• If (a, d, k) and (b, d, k) are both in in[n], then a and b may alias in n
Questions

Top
Using the Results of Alias Analysis

- For available expressions, we defined $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

  - Quadruple forms $n$:
    
    | Quadruple | gen[n] | kill[n] |
    |------------|--------|---------|
    | a = b op c | $\{n\} - \text{kill}[n]$ | uses[a] |
    | a = load b | $\{n\} - \text{kill}[n]$ | uses[a] |
    | store b, a | $\emptyset$ | all loads |
    | br L       | $\emptyset$ | $\emptyset$ |
    | br a L1 L2 | $\emptyset$ | $\emptyset$ |
    | a = f(b_1, ..., b_n) | $\emptyset$ | uses[a] $\cup$ all loads |
    | return a   | $\emptyset$ | $\emptyset$ |
Using the Results of Alias Analysis

• For available expressions, we defined $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

<table>
<thead>
<tr>
<th>Quadruple forms n</th>
<th>gen[n]</th>
<th>kill[n]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b \text{ op } c$</td>
<td>${n} - \text{kill}[n]$</td>
<td>uses[a]</td>
</tr>
<tr>
<td>$a = \text{load } b$</td>
<td>${n} - \text{kill}[n]$</td>
<td>uses[a]</td>
</tr>
<tr>
<td>store b, a</td>
<td>$\emptyset$</td>
<td>all loads that might alias with a</td>
</tr>
<tr>
<td>br L</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>br a L1 L2</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$a = f(b_1,\ldots,b_n)$</td>
<td>$\emptyset$</td>
<td>uses[a] $\cup$ all loads</td>
</tr>
<tr>
<td>return a</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

• The more things we know can’t alias, the more expressions don’t get killed by a store, so they’re available later and we can optimize more!

• Similar rules apply for most other dataflow analyses: if we know two pointers/references don’t alias, we can treat them as separate variables, but if they may alias, we have to treat a change to one as messing up the other
Alias Analysis: Summary

• Even in languages that hide pointers (Tiger, Java, etc.), we can end up with two references to the same memory location.

• In the worst case, we assume that every change to memory invalidates what we know about every memory location, but we can do better!

• Most languages use some combination of type-based and flow-based alias analysis
  – More type guarantees means a stronger analysis, which means more optimizations!

• The more precise our analysis, the more optimizations we enable (by improving the results of other analyses)
  – Lots of compiler development effort goes into improving alias analysis! There are always more tricks to add, and the payoff is good, since it improves every other analysis.