**Aliasing – Definitions I**

- Aliasing occurs when two variables refer to the same memory location.
- Aliasing occurs in languages with reference parameters, pointers, or arrays.
- There are two alias analysis problems. Let $a$ and $b$ be references to memory locations. At a program point $p$
  - $\text{may-alias}(p)$ is the set of pairs $(a, b)$ such that there exists at least one execution path to $p$, where $a$ and $b$ refer to the same memory location.
  - $\text{must-alias}(p)$ is a set of pairs $(a, b)$ such that on all execution paths to $p$, $a$ and $b$ refer to the same memory location.

**Slide 8–1**

**Aliasing – Definitions II**

- An alias analysis algorithm can be
  - **flow-sensitive** i.e. it takes the flow of control into account when computing aliases, or
  - **flow-insensitive** i.e. it ignores if-statements, loops, etc.
- There are intra-procedural and inter-procedural alias analysis algorithms.
- In the general case alias analysis is undecidable. However, there exist many conservative algorithms that perform well for actual programs written by humans.

**Slide 8–2**

**Aliasing – Definitions III**

- A conservative may-alias analysis algorithm may sometimes report that two variables $p$ and $q$ might refer to the same memory location, while, in fact, this could never happen. Equivalently, $p$ may-alias $q$ if we cannot prove that $p$ is never an alias for $q$.

**Slide 8–3**
**Formal–Formal Aliasing**

VAR a : INTEGER;
PROCEDURE F (VAR b, c : INTEGER);
BEGIN
  b := c + 6; PRINT c;
END F;
BEGIN a := 5; F(a, a); END.

---

**Formal–Global Aliasing**

VAR a : INTEGER;
PROCEDURE F (VAR b: INTEGER);
VAR x : INTEGER;
BEGIN
  x := a; b := 6; PRINT a;
END F;
BEGIN a := 5; F(a); END.

---

**Pointer–Pointer Aliasing**

TYPE Ptr = REF RECORD [N:Ptr; V:INTEGER];
VAR a,b : Ptr;
VAR X : INTEGER := 7;
BEGIN
  b := a := NEW Ptr;
  b^.V := X; a^.V := 5;
  PRINT b^.V;
END.

---

**Array Element Aliasing**

VAR A : ARRAY [0..100] OF INTEGER;
VAR i, j, X : INTEGER;
BEGIN
  i:=5; j:=2; X:=9; ...; j:=j+3;
END.

---

**Generated Code**

<table>
<thead>
<tr>
<th>Formal–Formal Aliasing</th>
<th>Formal–Global Aliasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>F: load R1, c^ # R1 holds c</td>
<td>F: load R1, a # R1 holds a</td>
</tr>
<tr>
<td>add R2, R1, 6</td>
<td>store x, R1</td>
</tr>
<tr>
<td>store b^, R2</td>
<td>storex b^</td>
</tr>
<tr>
<td>PRINT R1 # PRINT c</td>
<td>PRINT R1 # PRINT a</td>
</tr>
<tr>
<td>main: storec a, 5 # a := 5</td>
<td>main: storec a, 5 # a := 5</td>
</tr>
<tr>
<td>pusha a</td>
<td>pusha a</td>
</tr>
<tr>
<td>pusha a</td>
<td>call F # F(&amp;a,&amp;a)</td>
</tr>
</tbody>
</table>

---

**Generated Code**

<table>
<thead>
<tr>
<th>Pointer–Pointer Aliasing</th>
<th>Array Element Aliasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>main: storec X, 7 # X := 7</td>
<td>main: storec i, 5 # i := 5</td>
</tr>
<tr>
<td>new a, 8 # a := NEW Ptr</td>
<td>storec j, 2 # j := 2</td>
</tr>
<tr>
<td>copy b, a # b := a</td>
<td>storec X, 9 # X := 9</td>
</tr>
<tr>
<td>load R1, X # R1 holds X</td>
<td>... ...</td>
</tr>
<tr>
<td>store b^+4, R1 # b^.V := X</td>
<td>add j, 3 # j := j + 3</td>
</tr>
<tr>
<td>storec a^+4, 5 # a^.V := 5</td>
<td>load R1, X # R1 holds X</td>
</tr>
<tr>
<td>PRINT R1 # PRINT b^.X;</td>
<td>store A[i], R1 # A[i] := X</td>
</tr>
<tr>
<td></td>
<td>store A[j], 8 # A[j] := 8</td>
</tr>
<tr>
<td></td>
<td>PRINT R1 # PRINT A[i]</td>
</tr>
</tbody>
</table>
Type-Based Algorithms

- In strongly typed languages (Java, Modula-3) we can use a type-based alias analysis algorithm.
- Idea: if $p$ and $q$ are pointers that point to different types of objects, then they cannot possibly be aliases.
- Below, $p$ may-alias $r$; but $p$ and $q$ cannot possibly be aliases.
- This is an example of a flow-insensitive algorithm; we don’t detect that $p$ and $r$ actually point to different objects.

\[
\begin{align*}
\text{TYPE } T1 & : \text{ POINTER TO CHAR}; \\
\text{TYPE } T2 & : \text{ POINTER TO REAL}; \\
\text{VAR } p, r & : T1; \text{ VAR } q : T2; \\
\text{BEGIN} & \quad p := \text{NEW } T1; \quad r := \text{NEW } T1; \quad q := \text{NEW } T2; \\
\text{END}; \\
\end{align*}
\]

A Flow-Sensitive Algorithm I

- Assume the following language ($p$ and $q$ are pointers):

\[
\begin{align*}
p & := \text{new } T \quad \text{create a new object of type } T. \\
p & := \&a \quad p \text{ now points only to } a. \\
p & := q \quad p \text{ now points only to what } q \text{ points to.} \\
p & := \text{nil} \quad p \text{ now points to nothing.} \\
\end{align*}
\]

- The language also has the standard control structures.
- May-alias analysis is a forward-flow data-flow analysis problem.

A Flow-Sensitive Algorithm II

- We’ll be manipulating sets of alias pairs $\langle p, q \rangle$: $p$ and $q$ are access paths, either:
  1. l-value’d expressions (such as $a[i].v[k].w$) or
  2. program locations $S_1, S_2, \ldots$. Program locations are used when new dynamic data is created using `new`.
- $\text{in}[B]$ and $\text{out}[B]$ are sets of $\langle p, q \rangle$-pairs.
- $\langle p, q \rangle \in \text{in}[B]$ if $p$ and $q$ could refer to the same memory location at the beginning of $B$.

\[
\begin{align*}
\text{out}[B] & = \text{trans}_B(\text{in}[B]) \\
\text{in}[B] & = \bigcup_{P \in \text{predecessors of } B} \text{out}[P] \\
\end{align*}
\]
A Flow-Sensitive Algorithm III

- $\text{trans}_B(S)$ is a transfer function. If $S$ is the alias pairs defined at the beginning of $B$, then $\text{trans}_B(S)$ is the set of pairs defined at the exit of $B$.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$\text{trans}_B(S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$: $p := \text{new } T$</td>
<td>$(S - {&lt;p,b&gt;</td>
</tr>
<tr>
<td>$p := &amp;a$</td>
<td>$(S - {&lt;p,b&gt;</td>
</tr>
<tr>
<td>$p := q$</td>
<td>$(S - {&lt;p,b&gt;</td>
</tr>
<tr>
<td>$p := \text{nil}$</td>
<td>$S - {&lt;p,b&gt;</td>
</tr>
</tbody>
</table>

Example I/B – After First Iteration

- $q := &c$
- $i = \{<q,c>\}$
- $o = \{<q,c>, <p,a>\}$

- $p := &a$
- $i = \{<q,c>, <p,a>\}$
- $o = \{<q,c>, <p,a>, <p,c>, <p,a>\}$

Example I/C – After Second Iteration

- $q := &c$
- $i = \{<q,c>, <q,a>, <p,c>, <p,a>\}$
- $o = \{<q,c>, <p,c>, <p,a>\}$

- $p := &a$
- $i = \{<q,c>, <p,c>, <p,a>\}$
- $o = \{<q,c>, <p,a>, <p,c>\}$

- $p := q$
- $i = \{<q,c>, <q,a>, <p,c>\}$
- $o = \{<q,c>, <q,a>, <p,c>, <p,a>\}$

Example I/A – Initial State

- $i = \{\}$

- $q := &c$
- $p := &a$
- $q := &a$

- $p := q$

repeat
  $q := &c$;
  if (...) then
  else
    $q := &a$
    $p := q$
until ...;
Example II/A

TYPE T = REF RECORD[head:INTEGER; tail:T];
VAR p,q : T;
BEGIN
  S1:  p := NEW T;
  S2:  p^.head := 0;
  S3:  p^.tail := NIL;
  S4:  q := NEW T;
  S5:  q^.head := 6;
  S6:  q^.tail := p;
  IF a=0 THEN S7:  p := q; ENDIF;
  S8:  p^.head := 4;
END;

Example II/B

<table>
<thead>
<tr>
<th>in[S1]</th>
<th>out[S1] = {&lt;p,S1&gt;,&lt;q,S1&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>in[S2]</td>
<td>out[S2] = {&lt;p,S2&gt;,&lt;q,S2&gt;}</td>
</tr>
<tr>
<td>in[S3]</td>
<td>out[S3] = {&lt;p,S3&gt;,&lt;q,S3&gt;}</td>
</tr>
<tr>
<td>in[S4]</td>
<td>out[S4] = {&lt;p,S4&gt;,&lt;q,S4&gt;}</td>
</tr>
</tbody>
</table>

Example II/C

<table>
<thead>
<tr>
<th>in[S5]</th>
<th>out[S5] = {&lt;p,S5&gt;,&lt;q,S5&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>in[S6]</td>
<td>out[S6] = {&lt;p,S6&gt;,&lt;q,S6&gt;}</td>
</tr>
<tr>
<td>in[S7]</td>
<td>out[S7] = {&lt;p,S7&gt;,&lt;q,S7&gt;}</td>
</tr>
<tr>
<td>in[S8]</td>
<td>out[S8] = {&lt;p,S8&gt;,&lt;q,S8&gt;}</td>
</tr>
</tbody>
</table>

Example II/D

<table>
<thead>
<tr>
<th>in[S9]</th>
<th>out[S9] = {&lt;p,S9&gt;,&lt;q,S9&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>in[S10]</td>
<td>out[S10] = {&lt;p,S10&gt;,&lt;q,S10&gt;}</td>
</tr>
<tr>
<td>in[S12]</td>
<td>out[S12] = {&lt;p,S12&gt;,&lt;q,S12&gt;}</td>
</tr>
</tbody>
</table>
Complexity Results

- Inter-procedural case is no more difficult than intra-procedural (wrt \( P \) vs. \( NP \)).
- 1-level of indirection \( \Rightarrow P \); \( \geq 2 \)-levels of indirection \( \Rightarrow NP \).

**Banning’79** Reference formals, no pointers, no structures \( \Rightarrow P \).

**Horwitz’97** Flow-insensitive, may-alias, arbitrary levels of pointers, arbitrary pointer dereferencing \( \Rightarrow NP \) — hard.

**Landi&Ryder’91** Flow-sensitive, may-alias, multi-level pointers, intra-procedural \( \Rightarrow NP \) — hard.

**Landi’92** Flow-sensitive, must-alias, multi-level pointers, intra-procedural, dynamic memory allocation \( \Rightarrow \) Undecidable.

Shape Analysis I

- It is often useful to determine what kinds of dynamic structures a program constructs.
- For example, we might want to find out what a pointer \( p \) points to at a particular point in the program. Is it a linked list? A tree structure? A DAG?
- If we know that
  1. \( p \) points to a (binary) tree structure, and
  2. the program contains a call \( Q(p) \), and
  3. \( Q \) doesn’t alter \( p \)
then we can parallelize the call to \( Q \), running (say) \( Q(p\.left) \) and \( Q(p\.right) \) on different processors. If \( p \) instead turns out to point to a general graph structure, then this parallelization will not work.

Shape Analysis II

- Shape analysis requires alias analysis. Hence, all algorithms are approximate.

**Ghiya’96a** Accurate for programs that build simple data structures (trees, arrays of trees). Cannot handle major structural changes to the data structure.

**Chase’90** Problems with destructive updates. Handles list append, but not in-place list reversal.

**Hendren’90** Cannot handle cyclic structures.

**various** Only handle recursive structures no more than \( k \) levels deep.

**Deutsch’94** Powerful, but large (8000 lines of ML) and slow (30 seconds to analyze a 50 line program).

References

- Further readings:
Summary

- We should track aliases across procedure calls. This is *inter-procedural alias analysis*. See the Dragon book, pp. 655–660.

- Why is aliasing difficult? A program that has recursive data structures can have an infinite number of objects which can alias each other. Any aliasing algorithm must use a finite representation of all possible objects.

- Many (all?) static analysis techniques require alias analysis. Much use in software engineering, e.g. in the analysis of legacy programs.

- Pure functional languages don’t need alias analysis!