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 $\langle a, b \rangle$ 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 $\langle a, b \rangle$ such that on all execution paths to $p$, $a$ and $b$ refer to the same memory location.

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.

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 $q$. 
Where Does Aliasing Occur?

### Formal–Formal Aliasing

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

#### Generated Code

```
F: load R1, c^ # R1 holds c
   add R2, R1, 6
   store b^, R2
   PRINT R1 # PRINT c
main: storec a, 5 # a := 5
       pusha a
       pusha a
       call F # F(&a,&a)
```

### Formal–Global Aliasing

```pascal
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.
```

#### Generated Code

```
F: load R1, a # R1 holds a
   store x, R1
   store b^, 6
   PRINT R1 # PRINT a
main: storec a, 5 # a := 5
       pusha a
       pusha a
       call F # F(&a)
```

### Pointer–Pointer Aliasing

```pascal
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.
```

#### Generated Code

```
main: storec X, 7 # X := 7
       new a, 8 # a := NEW Ptr
       copy b, a # b := a
       load R1, X # R1 holds X
       store b^+4, R1 # b^.V := X
       storec a^+4, 5 # a^.V := 5
       pusha a^+4, 5
       PRINT R1 # PRINT b^.X;
```
VAR A : ARRAY [0..100] OF INTEGER;
VAR i, j, X : INTEGER;
BEGIN
  i:=5; j:=2; X:=9; · · ·; j:=j+3;
END.

Classifying Aliasing

Flow-Sensitive vs. Flow-Insensitive

Flow-Sensitive Flow-Insensitive

S_1 : p=&r; \{<*p,r>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
  if (...) 
  \{<*p,r>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
S_2 : q=p \{<*p,r>,<*q,s>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
  else 
  \{<*p,r>,<*q,s>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
S_3 : q=&s \{<*p,r>,<*q,s>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
S_4 : \ldots \{<*p,r>,<*q,s>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}
S_5 : q=&t \{<*p,r>,<*q,t>\} \{<*p,r>,<*q,s>,<*q,r>,<*q,t>\}

Let z and v be pointers in the following program fragment:

(1) x := y + z
(2) v := 5
(3) PRINT y + z

If we were performing an Available Expressions data flow analysis in order to find common sub-expressions, we would have to assume that the value computed for \( y + z \) on line (1) was killed by the assignment on line (2).

However, if alias analysis could determine that \( \text{may-alias}(z,v) = \text{false} \) then we could be sure that replacing \( y + z \) by \( x \) on line (3) would be safe.

Using May-Alias Analysis

<p,q> is a common notation for p may-alias q.

Flow-insensitive algorithms are cheaper. Flow-sensitive algorithms are more precise.
A Type-Based Algorithm

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.

```plaintext
TYPE T1 : POINTER TO CHAR;
TYPE T2 : POINTER TO REAL;
VAR p,r : T1; VAR q : T2;
BEGIN
  p := NEW T1; r := NEW T1; q := NEW T2;
END;
```

A Flow-Sensitive Algorithm

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

- \( p := \text{new } T \) create a new object of type \( T \).
- \( p := \&a \) \( p \) now points only to \( a \).
- \( p := q \) \( p \) now points only to what \( q \) points to.
- \( p := \text{nil} \) \( p \) now points to nothing.

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 \(<p,q>\). \(p\) 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 \(<p,q>-pairs.\)
- \(<p,q> \in \text{in}[B]\) if \(p\) and \(q\) could refer to the same memory location at the beginning of \(B\).

\[
\text{out}[B] = \text{trans}_B(\text{in}[B])
\]

\[
\text{in}[B] = \bigcup_{\text{predecessors } P \text{ of } B} \text{out}[P]
\]

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

\[
\begin{array}{c|c}
\text{trans}_B(S) &  \\
\hline
d: p := \textbf{new } T & \{S - \{<p,b> | \text{any } b\}\} \cup \{<p,d>\} \\
p := &a & \{S - \{<p,b> | \text{any } b\}\} \cup \{<p,a>\} \\
p := q & \{S - \{<p,b> | \text{any } b\}\} \cup \{<p,b> <q,b> \in S\} \\
p := \text{nil} & S - \{<p,b> | \text{any } b\} \\
\end{array}
\]

Example I/A – Initial State

```
\text{i=\{} \\
\text{q := &c} \\
\text{p := &a} \\
\text{q := &a} \\
\text{p := q}
```

Example I/B – After First Iteration

```
\text{i=\{} \\
\text{q := &c} \\
\text{p := q} \\
\text{repeat} \\
\text{q := &c; } \\
\text{if (...) \text{ then}} \\
\text{\quad p := &a} \\
\text{\text{else}} \\
\text{\quad q := &a} \\
\text{\quad p := q} \\
\text{\text{until ...;}}
```

```
\text{i=\{} \\
\text{q := &c} \\
\text{o=\{}<q,c>\} \\
\text{p := &a} \\
\text{q := &a} \\
\text{o=\{}<q,a>,<p,a>\} \\
\text{p := q} \\
\text{repeat} \\
\text{p := q; } \\
\text{if (...) \text{ then}} \\
\text{\quad p := &a} \\
\text{\text{else}} \\
\text{\quad q := &a} \\
\text{\quad p := q} \\
\text{\text{until ...;}}
```

```
\text{i=\{} \\
\text{p := q} \\
\text{o=\{}<q,a>,<p,a>\} \\
\text{q := &a} \\
\text{o=\{}<q,a>,<q,a>,<p,a>,<p,a>\} \\
\text{p := q} \\
```

```
\text{i=\{} \\
\text{p := q} \\
\text{o=\{}<q,a>,<q,a>,<p,a>,<p,a>\} \\
```

```
\text{i=\{} \\
\text{p := nil} \\
\text{o=\{}<q,a>,<q,a>,<p,a>,<p,a>\} \\
```

```
\text{i=\{} \\
\text{p := nil} \\
\text{o=\{}<q,a>,<q,a>,<p,a>,<p,a>\} \\
```
Example I/C – After Second Iteration

\[ \begin{align*}
p &:= q \\
o &:= \{<q,c>, <p,c>, <p,a>\}
\end{align*} \]

\[ \begin{align*}
i &:= \{<q,c>, <p,c>, <p,a>\}
\end{align*} \]

\[ \begin{align*}
o &:= \{<q,c>, <p,a>\}
\end{align*} \]

Example II/A

\[ \begin{align*}
\text{TYPE T} &= \text{REF RECORD[head:INTEGER;tail:T;]} \\
\text{VAR p,q : T;} \\
\text{BEGIN} \\
S_1 &:= p := \text{NEW T;} \\
S_2 &:= p^.head := 0; \\
S_3 &:= p^.tail := \text{NIL;} \\
S_4 &:= q := \text{NEW T;} \\
S_5 &:= q^.head := 6; \\
S_6 &:= q^.tail := p; \\
\text{IF a=0 THEN} \\
S_7 &:= p := q; \\
\text{ENDIF;} \\
S_8 &:= p^.head := 4; \\
\text{END;} \\
\end{align*} \]

Example II/B

\[ \begin{align*}
S_1 &:= p := \text{new T} \\
in[S_1] &= \{\} \\
\text{out}[S_1] &= \{<p,S_1>\} \\
S_2 &:= p^.head := 0 \\
in[S_2] &= \text{out}[S_1] = \{<p,S_1>\} \\
\text{out}[S_2] &= \{<p,S_1>\} \\
S_3 &:= p^.tail := \text{nil} \\
in[S_3] &= \text{out}[S_2] = \{<p,S_1>\} \\
\text{out}[S_3] &= \{<p,S_1>\} \\
S_4 &:= q := \text{new T} \\
in[S_4] &= \text{out}[S_3] = \{<p,S_1>\} \\
\text{out}[S_4] &= (\text{in}[S_4] - \{\}) \cup \{<q,S_4>\} \\
&= \{<p,S_1>,<q,S_4>\} \\
\end{align*} \]

Example II/C

\[ \begin{align*}
S_5 &:= q^.head := 6 \\
in[S_5] &= \text{out}[S_4] = \{<p,S_1>,<q,S_4>\} \\
\text{out}[S_5] &= \{<p,S_1>,<q,S_4>\} \\
S_6 &:= q^.tail := p \\
in[S_6] &= \text{out}[S_5] = \{<p,S_1>,<q,S_4>\} \\
\text{out}[S_6] &= (\text{in}[S_6] - \{\}) \cup \{<q.tail,S_1>\} \\
&= \{<p,S_1>,<q,S_4>,<q.tail,S_1>\} \\
S_7 &:= p := q \\
in[S_7] &= \text{out}[S_6] = \{<p,S_1>,<q,S_4>,<q.tail,S_1>\} \\
\text{out}[S_7] &= (\text{in}[S_6] - \{<p,S_1>\}) \cup \{<p,S_4>\} \\
&= \{<p,S_4>,<q,S_4>,<q.tail,S_1>\} \\
\end{align*} \]
Example II/D

\[ S_8: p^.\text{head} := 4 \quad \text{in}[S_8] = \text{out}[S_6] \cup \text{out}[S_7] = \{< p, S_1>, < p, S_4>, < q, S_4>, < q.\text{tail}, S_1>\} \]

\[ \text{out}[S_8] = \text{in}[S_8] = \{< p, S_1>, < p, S_4>, < q, S_4>, < q.\text{tail}, S_1>\} \]

Summary

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) \( Q(p^.\text{left}) \) and \( Q(p^.\text{right}) \) on different processors. If \( p \) instead turns out to point to a general graph structure, then this parallelization will not work.
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).

**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!

**Readings and References**

- Further readings: