Lecture 13a
Alias and points-to analysis

Motivation

We've seen a number of different analyses that are affected by ambiguity in variables accessed (e.g. in LVA we assume all address-taken variables are referenced).

Alongside this, in modern machines we would like to exploit parallelism where possible, either by running code in separate threads on a multi-core, or in separate lanes using short vector (SIMD) instructions.

Our ability to do this depends on us being able to tell whether memory-access instructions alias (i.e. access the same memory location).

Example

As a simple example, consider some MP3 player code:

```
for (channel = 0; channel < 2; channel++)
  process_audio(channel);
```

Or even

```
process_audio_left();
process_audio_right();
```

Can we run these two calls in parallel?
In other words, when is it safe to do so?

Memory accessed

In general we can parallelise if neither call writes to a memory location read or written by the other.

We therefore want to know, at compile time, what memory locations a procedure might read from and write to at run time.

Essentially, we're asking what locations the procedure's instructions access at run time.

Memory accessed

We can reduce this problem to finding locations accessed by each instruction, then combining for all instructions within a procedure.

So, given a pointer value, we are interested in finding a *finite description* of the locations it *might* point to.

If two such descriptions have an empty intersection then we can parallelise / reorder the instructions / …

Andersen's analysis

Andersen's points-to analysis is an $O(n^3)$ analysis—the underlying problem is the same as 0-CFA.

We'll only look at the intra-procedural case.

We won't consider pointer arithmetic or functions returning pointers.

All object fields are conflated, although a 'field-sensitive' analysis is possible too.

Andersen's analysis

Assume the program has been re-written so that all *pointer-typed* operations are of the form:

$x := \texttt{new}_\ell$	ℓ is a program point
$x := \texttt{null}$	optional, a variant of `new`
$x := \&y$	C-like languages, also like `new`
$x := y$	copy
$x := *y$	field access of an object
$*x := y$	field access of an object

Andersen's analysis

We first define a set of abstract values:

$$V = Var \cup \{\texttt{new}_\ell \mid \ell \in Prog\} \cup \{\texttt{null}\}$$

Note that all allocations at program point ℓ are conflated, which makes things finite but loses precision.

We create the *points-to* relation as a function:

$$pt : V \to \mathcal{P}(V)$$

Some analyses have a different pt at each program point (like LVA); Andersen keeps one per function.

Andersen's analysis

We could use type-like constraints (one per source-level assignment):

$$\frac{}{\vdash x := \&y : y \in pt(x)}$$

$$\frac{}{\vdash x := \texttt{null} : \texttt{null} \in pt(x)}$$

$$\frac{}{\vdash x := \texttt{new}_\ell : \texttt{new}_\ell \in pt(x)}$$

$$\frac{}{\vdash x := y : pt(y) \subseteq pt(x)}$$

$$\frac{z \in pt(y)}{\vdash x := *y : pt(z) \subseteq pt(x)}$$

$$\frac{z \in pt(x)}{\vdash *x := y : pt(y) \subseteq pt(z)}$$

Andersen's analysis

Or use the style of 0-CFA:

```
x  :=  &y
```

$$pt(x) \supseteq \{\ y\ \}$$

Andersen's analysis

Or use the style of 0-CFA:

$$\texttt{x} \;\; \texttt{:=} \;\; \texttt{y}$$

$$\downarrow \qquad\qquad \downarrow$$

$$pt(x) \supseteq pt(y)$$

Andersen's analysis

Or use the style of 0-CFA:

```
x := *y
```

$$pt(y) \supseteq \{ z \} \Longrightarrow pt(x) \supseteq pt(z)$$

Andersen's analysis

Or use the style of 0-CFA:

```
*x := y
```

$$pt(x) \supseteq \{ z \} \implies pt(z) \supseteq pt(y)$$

Note that this is just stylistic, it's the same constraint system but no obvious deep connections between 0-CFA and Andersen's points-to analysis.

Andersen's analysis

The algorithm is flow-insensitive—it only considers the statements and not the order in which they occur. This is faster but less precise.

Property inference rules are then essentially:

$$(\mathrm{ASS})\,\frac{}{\vdash x := e : \ldots} \qquad (\mathrm{SEQ})\,\frac{\vdash C : S \quad \vdash C' : S'}{\vdash C; C' : S \cup S'}$$

$$(\mathrm{COND})\,\frac{\vdash C : S \qquad \vdash C' : S'}{\vdash \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C' : S \cup S'}$$

$$(\mathrm{WHILE})\,\frac{\vdash C : S}{\vdash \texttt{while}\ e\ \texttt{do}\ C : S}$$

Andersen example

Consider the following code:

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

pt(a) = {}
pt(c) = {}
pt(b) = {}
pt(d) = {}
pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(a) \supseteq \{\ b\ \}$

pt(a) = { b }

pt(c) = {}

pt(b) = {}

pt(d) = {}

pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(c) \supseteq \{ d \}$

pt(a) = { b }

pt(b) = {}

pt(c) = { d }

pt(d) = {}

pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(d) \supseteq \{ a \}$

pt(a) = { b }

pt(b) = {}

pt(c) = { d }

pt(d) = { a }

pt(e) = {}

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(e) \supseteq pt(c)$

pt(a) = { b }

pt(c) = { d }

pt(b) = {}

pt(d) = { a }

pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$$pt(e) \supseteq \{ d \} \implies pt(c) \supseteq pt(d)$$

pt(a) = { b }

pt(b) = {}

pt(c) = { a, d }

pt(d) = { a }

pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$$pt(a) \supseteq \{ b \}$$
$$\implies pt(b) \supseteq pt(d)$$

pt(a) = { b }

pt(c) = { a, d }

pt(b) = { a }

pt(d) = { a }

pt(e) = { d }

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(e) \supseteq pt(c)$

pt(a) = { b }

pt(b) = { a }

pt(c) = { a, d }

pt(d) = { a }

pt(e) = { a, d }

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$$pt(e) \supseteq \{ a, d \}$$
$$\Longrightarrow pt(c) \supseteq pt(a)$$
$$\Longrightarrow pt(c) \supseteq pt(d)$$

pt(a) = { b }

pt(b) = { a }

pt(c) = { a, b, d }

pt(d) = { a }

pt(e) = { a, d }

Andersen example

```
a = &b;
c = &d;
d = &a;
e = c;
c = *e;
*a = d;
```

$\rightarrow$ $pt(e) \supseteq pt(c)$

pt(a) = { b }

pt(b) = { a }

pt(c) = { a, b, d }

pt(d) = { a }

pt(e) = { a, b, d }

Andersen example

$pt(a) = \{ b \}$ $pt(c) = \{ a, b, d \}$

$pt(b) = \{ a \}$ $pt(d) = \{ a \}$

$pt(e) = \{ a, b, d \}$

Note that a flow-sensitive algorithm would give
$pt(c) = \{ a, d \}$ and $pt(e) = \{ d \}$
assuming the statements appear in the given order in a single basic block that is not in a loop.

Other approaches

Steensgaard's algorithm merges `a` and `b` if any pointer can reference both. This is less accurate but runs in almost linear time.

Shape analysis (Sagiv, Wilhelm, Reps) models abstract heap nodes and edges between them representing *must* or *may* point-to. More accurate but the abstract heaps can get very large.

In general, coarse techniques give poor results whereas more sophisticated techniques are expensive.

Summary

- Points-to analysis identifies which memory locations variables (and other memory locations) point to
- We can use this information to improve data-dependence analysis
- This allows us to reorder loads and stores, or parallelise / vectorise parts of the code
- Andersen's analysis is a flow-insensitive algorithm that works in $O(n^3)$