Alias and Points-to Analysis

Alan Mycroft
Computer Laboratory, Cambridge University

http://www.cl.cam.ac.uk/teaching/current/OptComp

Lecture 13a
Points-to analysis, parallelisation etc.

Consider an MP3 player containing code:

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

or even

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

Can we run these two calls in parallel?
Points-to analysis, parallelisation etc. (2)

Multi-core CPU: *probably* want to run these two calls in parallel:

```c
#pragma omp parallel for // OpenMP
for (channel = 0; channel < 2; channel++)
    process_audio(channel);
```

or

```c
spawn process_audio_left(); // e.g. Cilk, X10
process_audio_right();
sync;
```

or

```c
par { process_audio_left() // language primitives
    ||| process_audio_right()
}
```

Question: when is this transformation *safe*?
Can we know what locations are read/written?

Basic parallelisation criterion: parallelise only if neither call writes to a memory location read or written by the other.

So, we want to know (at compile time) what locations a procedure might write to at run time. Sounds hard!
Can we know what locations are read/written?

Non-address-taken variables are easy, but consider:

\[
\text{for (i = 0; i < n; i++) } v[i] \rightarrow \text{field}++; \\
\]

Can this be parallelised? Depends on knowing that each cell of \( v[] \) points to a distinct object (i.e. there is no aliasing).

So, given a pointer value, we are interested in finding a finite description of what locations it might point to – or, given a procedure, a description of what locations it might read from or write to.

If two such descriptions have empty intersection then we can parallelise.
Can we know what locations are read/written?

For simple variables, even including address-taken variables, this is moderately easy (we have done similar things in “ambiguous ref” in LVA and “ambiguous kill” in Avail). Multi-level pointers, e.g.

```c
int a, *b, **c;
b=&a;
c=&b;
```

make the problem more complicated here.

What about `new`, especially in a loop?

Coarse solution: treat all allocations done at a single program point as being aliased (as if they all return a pointer to a single piece of memory).
Andersen’s points-to analysis

An $O(n^3)$ analysis – underlying problem same as 0-CFA. We’ll only look at the intra-procedural case.

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

- $x := \text{new}_l$  \( l \) is a program point (label)
- $x := \text{null}$ optional, can see as variant of \text{new}
- $x := &y$ only in C-like languages, also like \text{new} variant
- $x := y$ copy
- $x := *y$ field access of object
- $*x := y$ field access of object

Note: no pointer arithmetic (or pointer-returning functions here). Also fields conflated (but ‘field-sensitive’ is possible too).
Andersen’s points-to analysis (2)

Get set of abstract values $V = Var \cup \{\text{new}_\ell \mid \ell \in \text{Prog}\} \cup \{\text{null}\}$.

Note that this means that all new allocations at program point $\ell$ are conflated – makes things finite but loses precision.

The points-to relation is seen as a function $pt : V \to \mathcal{P}(V)$. While we might imagine having a different $pt$ at each program point (like liveness) Andersen keeps one per function.

Have type-like constraints (one per source-level assignment)

\[
\begin{align*}
\vdash x := &\ y : y \in pt(x) & \vdash x := &\ y : pt(y) \subseteq pt(x) \\
&\quad \quad z \in pt(y) & &\quad \quad z \in pt(x) \\
\vdash x := &\ \ast y : pt(z) \subseteq pt(x) & \vdash *x := &\ y : pt(y) \subseteq pt(z)
\end{align*}
\]

$x := \text{new}_\ell$ and $x := \text{null}$ are treated identically to $x := \& y$. 
Andersen’s points-to analysis (3)

Alternatively, the same formulae presented in the style of 0-CFA (this is only stylistic, it’s the same constraint system, but there are no obvious deep connections between 0-CFA and Andersen’s points-to):

- for command $x := \& y$ emit constraint $pt(x) \supseteq \{y\}$

- for command $x := y$ emit constraint $pt(x) \supseteq pt(y)$

- for command $x := * y$ emit constraint implication $pt(y) \supseteq \{z\} \implies pt(x) \supseteq pt(z)$

- for command $* x := y$ emit constraint implication $pt(x) \supseteq \{z\} \implies pt(z) \supseteq pt(y)$
Andersen’s points-to analysis (4)

Flow-insensitive – we only look at the assignments, not in which order they occur. Faster but less precise – syntax-directed rules all use the same set-like combination of constraints (∪ here).

Flow-insensitive means property inference rules are essentially of the form:

\[
\begin{align*}
\text{(ASS)} & \quad \vdash x := e : \ldots \\
\text{(SEQ)} & \quad \vdash C : S \implies \vdash C' : S' \\
\text{(COND)} & \quad \vdash C : S \implies \vdash C' : S' \\
\text{(WHILE)} & \quad \vdash C : S \implies \vdash \text{while } e \text{ do } C' : S
\end{align*}
\]
Andersen: example

[Example taken from notes by Michelle Mills Strout of Colorado State University]

<table>
<thead>
<tr>
<th>command</th>
<th>constraint</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = &amp; b$;</td>
<td>$pt(a) \supseteq {b}$</td>
<td>$pt(a) = {b, d}$</td>
</tr>
<tr>
<td>$c = a$;</td>
<td>$pt(c) \supseteq pt(a)$</td>
<td>$pt(c) = {b, d}$</td>
</tr>
<tr>
<td>$a = &amp; d$;</td>
<td>$pt(a) \supseteq {d}$</td>
<td>$pt(b) = pt(d) = {}$</td>
</tr>
<tr>
<td>$e = a$;</td>
<td>$pt(e) \supseteq pt(a)$</td>
<td>$pt(e) = {b, d}$</td>
</tr>
</tbody>
</table>

Note that a flow-sensitive algorithm would instead give $pt(c) = \{b\}$ and $pt(e) = \{d\}$ (assuming the statements appear in the above order in a single basic block).
### Andersen: example (2)

<table>
<thead>
<tr>
<th>command</th>
<th>constraint</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = &amp; b;\ $</td>
<td>$pt(a) \supseteq {b}$</td>
<td>$pt(a) = {b, d}$</td>
</tr>
<tr>
<td>$c = &amp; d;\ $</td>
<td>$pt(c) \supseteq {d}$</td>
<td>$pt(c) = {d}$</td>
</tr>
<tr>
<td>$e = &amp; a;\ $</td>
<td>$pt(e) \supseteq {a}$</td>
<td>$pt(e) = {a}$</td>
</tr>
<tr>
<td>$f = a;\ $</td>
<td>$pt(f) \supseteq pt(a)$</td>
<td>$pt(f) = {b, d}$</td>
</tr>
<tr>
<td>$* e = c;\ $</td>
<td>$pt(e) \supseteq {z}$ $\implies pt(z) \supseteq pt(c)$</td>
<td>(generates) $pt(a) \supseteq pt(c)$</td>
</tr>
</tbody>
</table>
Points-to analysis – some other approaches

• Steensgaard’s algorithm: treat $e := e'$ and $e' := e$ identically. Less accurate than Andersen’s algorithm but runs in almost-linear time.

• shape analysis (Sagiv, Wilhelm, Reps) – a program analysis with elements being abstract heap nodes (representing a family of real-world heap notes) and edges between them being *must* or *may* point-to. Nodes are labelled with variables and fields which may point to them. More accurate but abstract heaps can become very large.

Coarse techniques can give poor results (especially inter-procedurally), while more sophisticated techniques can become very expensive for large programs.
Points-to and alias analysis

“Alias analysis is undecidable in theory and intractable in practice.”

It’s also very discontinuous: small changes in program can produce global changes in analysis of aliasing. Potentially bad during program development.

So what can we do?

Possible answer: languages with type-like restrictions on where pointers can point to.

- Dijkstra said (effectively): spaghetti code is bad; so use structured programming.

- I argue elsewhere that spaghetti data is bad; so need language primitives to control aliasing (“structured data”).