

Alias and Points-to Analysis

Alan Mycroft Computer Laboratory, Cambridge University

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

Lecture 13a[may be updated for 2011]



Points-to analysis, parallelisation etc.

```
Consider an MP3 player containing code:
```

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

or even

process_audio_left();
process_audio_right();</pre>
```

Can we run these two calls in parallel?

2



Points-to analysis, parallelisation etc. (2)

Multi-core CPU: probably want to run these two calls in paraller MBRIDGE

```
#pragma omp parallel for
                            // OpenMP
  for (channel = 0; channel < 2; channel++)</pre>
    process_audio(channel);
or
  spawn process_audio_left(); // e.g. Cilk, X10
  process_audio_right();
  sync;
or
  ||| 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!





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

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.



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.

```
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).



Lecture 13a

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

 $x := \mathtt{new}_{\ell}$ ℓ is a program point (label) $x := \mathtt{null}$ optional, can see as variant of \mathtt{new} x := & y only in C-like languages, also like \mathtt{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).





Get set of abstract values $V = Var \cup \{\text{new}_{\ell} \mid \ell \in Prog\} \cup \{\text{null}\}^{\text{CAMBRIDGE}}$. Note that this means that all new allocations at program point ℓ 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)

 $x := \text{new}_{\ell} \text{ and } x := \text{null are treated identically to } x := \& y.$



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



Andersen's points-to analysis (4)

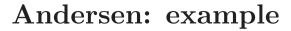
Flow-insensitive – we only look at the assignments, not in which order they occur. Faster but less precise – syntax-directed rules all use the same set-like combination of constraints (\cup here).

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

$$(ASS) \frac{\vdash C : S \vdash C' : S'}{\vdash C : C' : S \cup S'}$$

$$(COND) \frac{\vdash C : S \vdash C' : S'}{\vdash \text{ if } e \text{ then } C \text{ else } C' : S \cup S'}$$

$$(WHILE) \frac{\vdash C : S}{\vdash \text{ while } e \text{ do } C : S}$$





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

command	constraint	solution
a = &b	$pt(a) \supseteq \{b\}$	$pt(a) = \{b, d\}$
c = a;	$pt(c) \supseteq pt(a)$	$pt(c) = \{b, d\}$
a = &d	$pt(a) \supseteq \{d\}$	$pt(b) = pt(d) = \{\}$
e = a;	$pt(e) \supseteq pt(a)$	$pt(e) = \{b, d\}$

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)

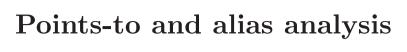
command	constraint	solution
a = &b	$pt(a) \supseteq \{b\}$	$pt(a) = \{b, \frac{\mathbf{d}}{\mathbf{d}}\}$
c = &d	$pt(c) \supseteq \{d\}$	$pt(c) = \{d\}$
e = &a	$pt(e) \supseteq \{a\}$	$pt(e) = \{a\}$
f = a;	$pt(f) \supseteq pt(a)$	$pt(f) = \{ b, d \}$
*e = c;	$pt(e) \supseteq \{z\} \Longrightarrow pt(z) \supseteq pt(c)$	
(generates)	$pt(a) \supseteq pt(c)$	



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.





"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").