CSC 6580
Spring 2020

Instructor: Stacy Prowell

Midterm

Changes

Adjusted scores from *best ten answers* to *best eight answers.*

Average change of +10 points

New high: 100
New average: 84

New grades are in iLearn

Steensgaard’s Algorithm

Points-to Analysis in Almost Linear Time

Bjarne Steensgaard

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

rusa@research.microsoft.com

We present a

- **flow insensitive**,
- **interprocedural**
- **points-to analysis** algorithm

that has a desirable

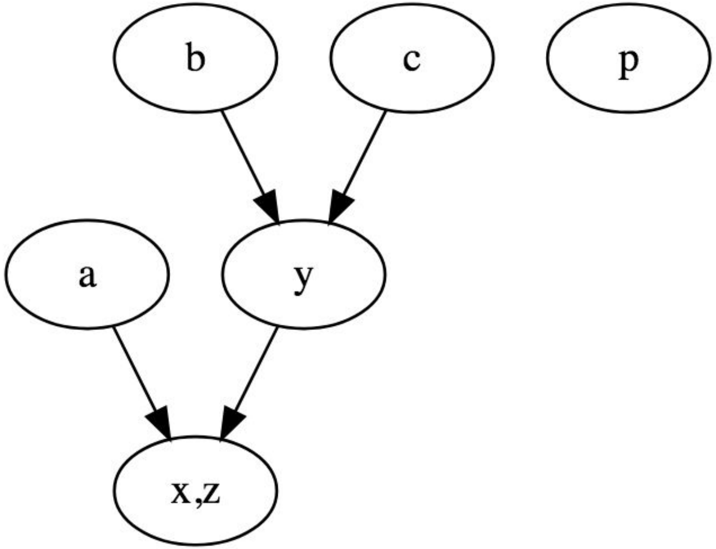
- **linear space** and
- almost **linear time** complexity and

is also **very fast in practice**.

Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

$$\begin{aligned}
\textbf{a}&: \tau_1 = \textbf{ref}(\tau_4 \times \bot)\\
\textbf{b}&: \tau_2 = \textbf{ref}(\tau_5 \times \bot)\\
\textbf{c}&: \tau_3 = \textbf{ref}(\tau_5 \times \bot)\\
\textbf{x}&: \tau_4 = \textbf{ref}(\bot \times \bot)\\
\textbf{y}&: \tau_5 = \textbf{ref}(\tau_4 \times \bot)\\
\textbf{z}&: \tau_4 = \textbf{ref}(\bot \times \bot)\\
\textbf{p}&: \tau_7 = \textbf{ref}(\bot \times \bot)
\end{aligned}$$



Recall

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

The "bottom" of the type lattice; think of it as not specified, or not determined

Basically anything not known to be a location or a pointer to a location

Recall

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

The type of some *thing* in memory

Recall

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

The type of of a *function*

Recall

$$
\begin{array}{lcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

Either one or the other (but probably not both)

Recall

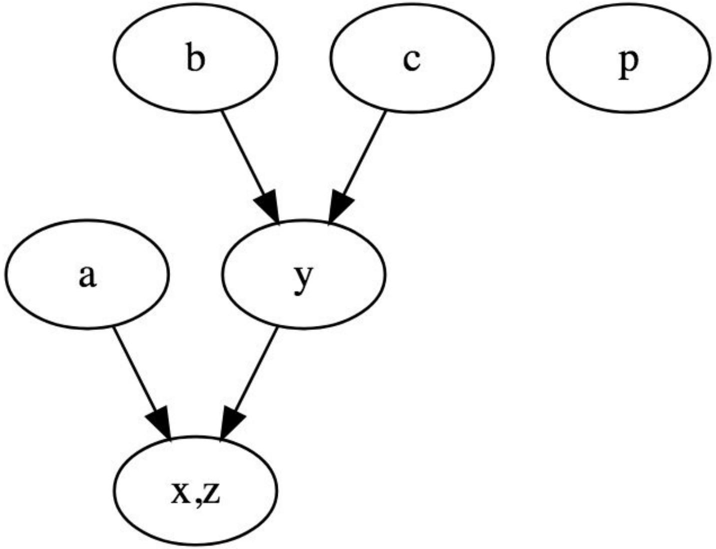
$$\begin{array}{lcl} \alpha & ::= & \tau \times \lambda \\ \tau & ::= & \bot \mid \mathbf{ref}(\alpha) \\ \lambda & ::= & \bot \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) \end{array}$$

Note the recursion

Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

$$
\begin{aligned}
&\mathbf{a}:\ \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{b}:\ \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{c}:\ \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{x}:\ \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{y}:\ \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{z}:\ \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{p}:\ \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$

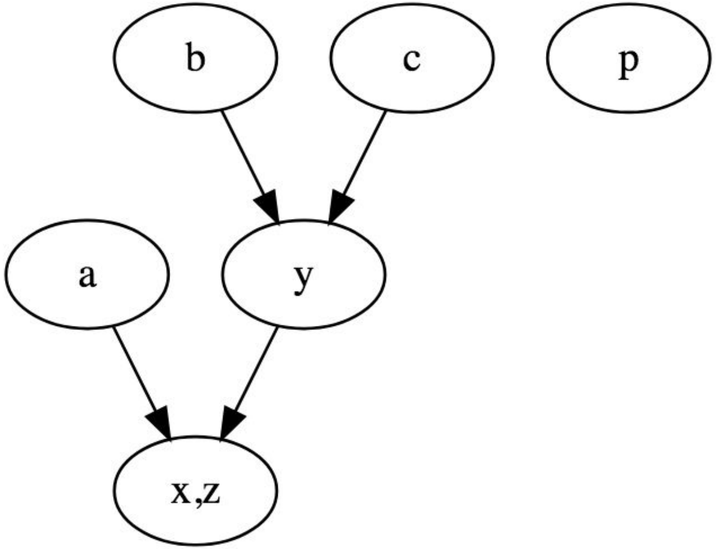


Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

variable

$$\begin{aligned}
&\mathbf{a}:\ \tau_1 = \mathbf{ref}(\tau_4 \times \perp)\\
&\mathbf{b}:\ \tau_2 = \mathbf{ref}(\tau_5 \times \perp)\\
&\mathbf{c}:\ \tau_3 = \mathbf{ref}(\tau_5 \times \perp)\\
&\mathbf{x}:\ \tau_4 = \mathbf{ref}(\perp \times \perp)\\
&\mathbf{y}:\ \tau_5 = \mathbf{ref}(\tau_4 \times \perp)\\
&\mathbf{z}:\ \tau_4 = \mathbf{ref}(\perp \times \perp)\\
&\mathbf{p}:\ \tau_7 = \mathbf{ref}(\perp \times \perp)
\end{aligned}$$

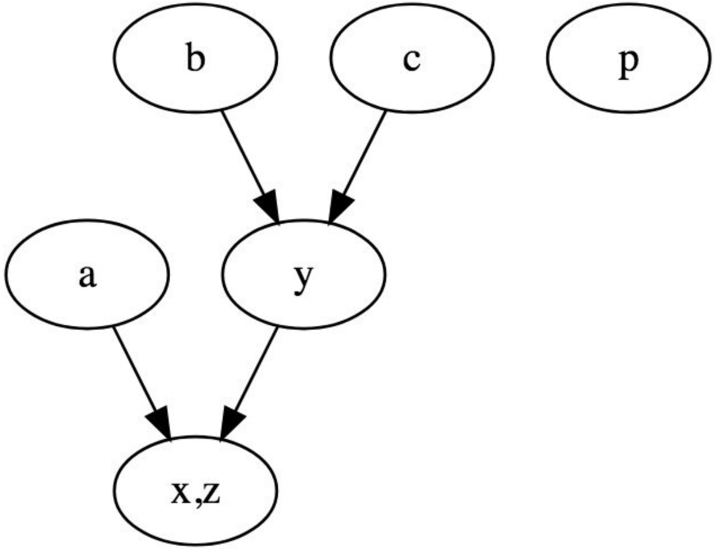


Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

type variable

$$
\begin{aligned}
&\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{c}\colon \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{z}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$

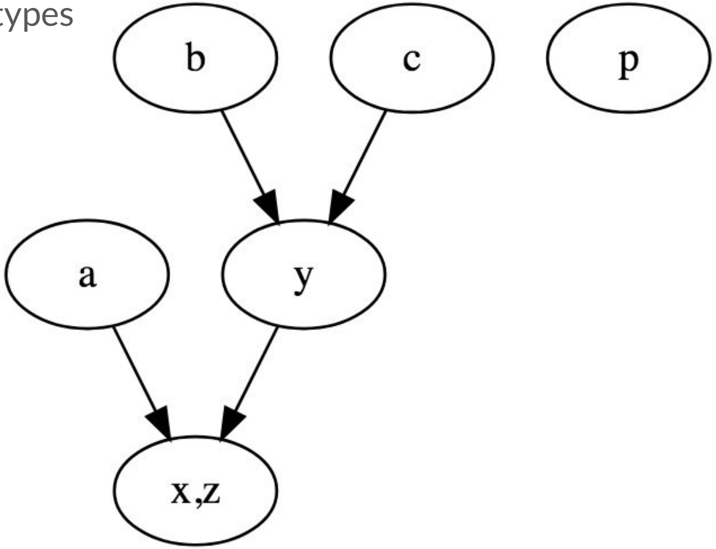


Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

simple types

$$
\begin{aligned}
&\mathbf{a}\colon \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{b}\colon \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{c}\colon \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
&\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
&\mathbf{z}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)\\
&\mathbf{p}\colon \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$

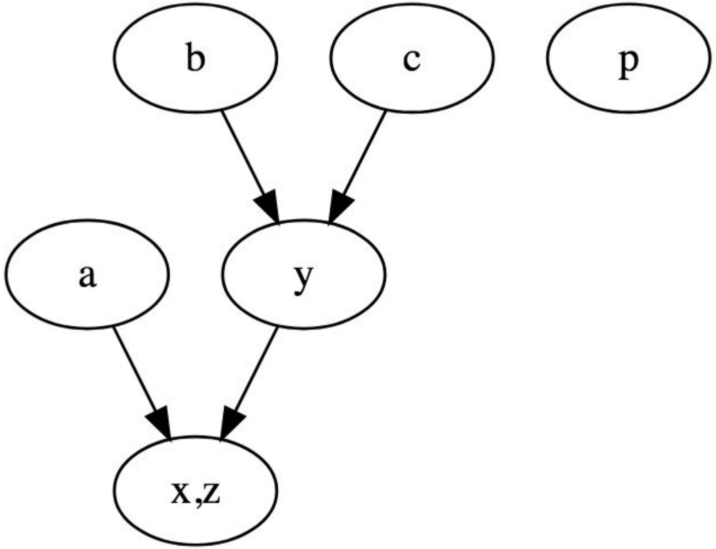


Points To Analysis

Figure out the *type* of thing pointed to by each variable (that is a pointer).

function pointer types

$$
\begin{aligned}
\mathbf{a}&: \tau_1 = \mathbf{ref}(\tau_4 \times \bot)\\
\mathbf{b}&: \tau_2 = \mathbf{ref}(\tau_5 \times \bot)\\
\mathbf{c}&: \tau_3 = \mathbf{ref}(\tau_5 \times \bot)\\
\mathbf{x}&: \tau_4 = \mathbf{ref}(\bot \times \bot)\\
\mathbf{y}&: \tau_5 = \mathbf{ref}(\tau_4 \times \bot)\\
\mathbf{z}&: \tau_4 = \mathbf{ref}(\bot \times \bot)\\
\mathbf{p}&: \tau_7 = \mathbf{ref}(\bot \times \bot)
\end{aligned}
$$



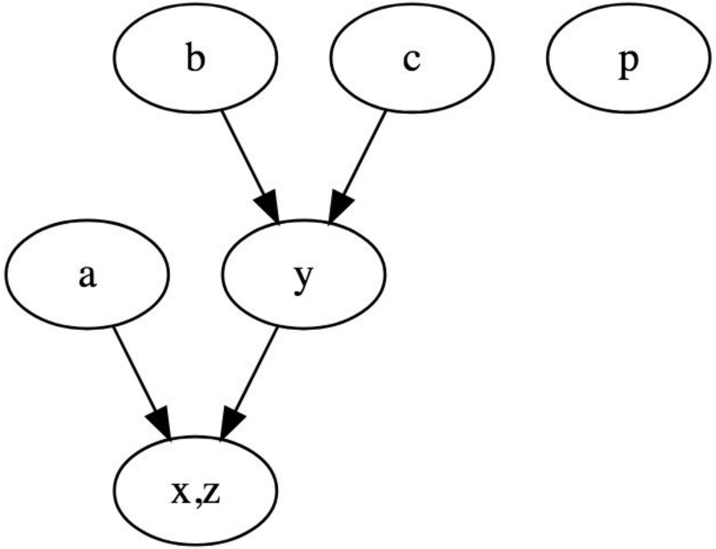
Points To Analysis

The algorithm works by:

- assuming every variable has a unique type
- *merging* type variables as it walks through the program

(See prior class notes.)

$$\boxed{\mathbf{a}\!: \tau_1 = \mathbf{ref}(\tau_4 \times \bot)}$$
$$\mathbf{b}\!: \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{c}\!: \tau_3 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{x}\!: \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\boxed{\mathbf{y}\!: \tau_5 = \mathbf{ref}(\tau_4 \times \bot)}$$
$$\mathbf{z}\!: \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{p}\!: \tau_7 = \mathbf{ref}(\bot \times \bot)$$

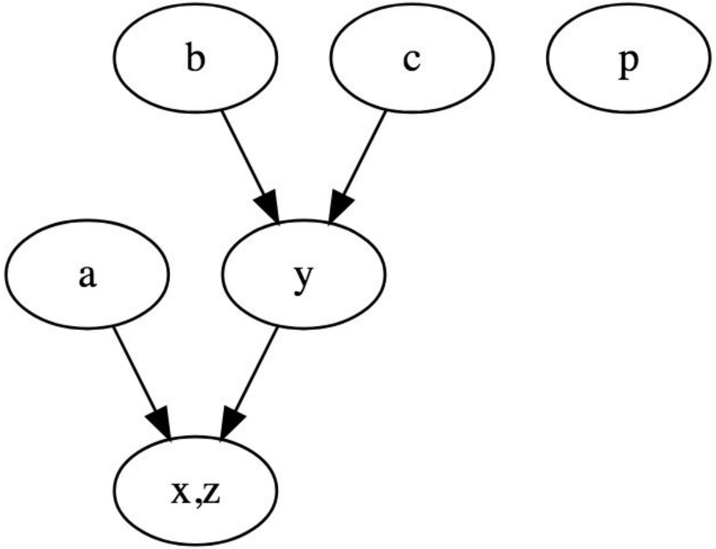


Points To Analysis

We need to be specific about when we can merge. We introduce a rule.

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

$$\boxed{\mathbf{a}\text{: } \tau_1 = \mathbf{ref}(\tau_4 \times \bot)}$$
$$\mathbf{b}\text{: } \tau_2 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{c}\text{: } \tau_3 = \mathbf{ref}(\tau_5 \times \bot)$$
$$\mathbf{x}\text{: } \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\boxed{\mathbf{y}\text{: } \tau_5 = \mathbf{ref}(\tau_4 \times \bot)}$$
$$\mathbf{z}\text{: } \tau_4 = \mathbf{ref}(\bot \times \bot)$$
$$\mathbf{p}\text{: } \tau_7 = \mathbf{ref}(\bot \times \bot)$$



Typing Rules

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

Typing Rules

$$\frac{\left\{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}\right\}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

If the things above the line are true…

Typing Rules

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{\left\{ A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y}) \right\}}$$

...then we can conclude that the thing below the line is true.

Typing Rules

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

A program statement

Typing Rules

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{\{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})\}}$$

We want to conclude that our rule produces *well-typed* programs

Typing Rules

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \quad A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \quad \alpha_2 \trianglelefteq \alpha_1}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

If our typing rule tells us (*entails*) that

- if **x** is a reference to some type α_1
- and **y** is a reference to some type α_2
- and if α_2 "fits in" α_1

then our typing rule correctly types **x = y**.

Partial Order

We defined a partial order among types as follows.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Partial Order

$$\boxed{t_1 \trianglelefteq t_2} \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

Defines a **partial order** among types.
Type t_1 is "less than" t_2 iff t_1 is bottom *or* the types are equal.

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Partial Order

$$\boxed{t_1 \trianglelefteq t_2} \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

Defines a **partial order** among types.
Type t_1 is "less than" t_2 iff t_1 is bottom *or* the types are equal.

$$\boxed{(t_1 \times t_2) \trianglelefteq (t_3 \times t_4)} \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

We extend the partial order to pairs by comparing the components.
This allows us to compare more complex data types, like structures.

Partial Order

Consider what this means if we can conclude that the left hand side is true, from our typing rule.

$$\boxed{t_1 \trianglelefteq t_2} \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$$

Partial Order

If we can conclude that the lhs is true from our typing rule, then the rhs must *also* be true.

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

So either:

- t_1 is bottom *or*
- $t_1 = t_2$

If the types are the same, we should *merge* them!

Partial Order

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

So if the lhs is ⊥, we don't need to merge, but if the lhs is *not* ⊥, then we do.

We call this a *conditional join*, and represent it by **cjoin**. This is different from the *always join* represented by **join**.

Partial Order

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \perp) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

So if the lhs is ⊥, we don't need to merge, but if the lhs is *not* ⊥, then we do.

But it is possible that we later change the type of a variable (join), and it is no longer ⊥. (We did this several times in the prior example.) Then the above relation might no longer hold for the prior lines.

Partial Order

We don’t want to go back (backtracking is expensive and messes up our time complexity), so instead:

- for every ⊥, keep a list of type variables to join with if we should ever change ⊥ to something else (let's call this set "pending") and...
- if we do change ⊥ to something else, merge all the type variables then.

Joins

$$\begin{aligned}
&\textbf{cjoin}(e_1, e_2):\\
&\quad \text{if } \textbf{type}(e_2) = \perp \text{ then}\\
&\quad\quad \textbf{pending}(e_2) \leftarrow \{e_1\} \cup \textbf{pending}(e_2)\\
&\quad \text{else}\\
&\quad\quad \textbf{join}(e_1, e_2)
\end{aligned}$$

This defines the conditional join of two types, exactly as we defined it earlier... but it is a bit hard to read. Either we add to the pending set, or we just join.

- for every ⊥, keep a list of type variables to join with if we should ever change ⊥ to something else (let's call this set "pending") and...
- if we do change ⊥ to something else, merge all the type variables then.

Joins

$\textbf{cjoin}(e_1, e_2)$:
  if $\textbf{type}(e_2) = \perp$ then
    $\textbf{pending}(e_2) \leftarrow \{e_1\} \cup \textbf{pending}(e_2)$
  else
    $\textbf{join}(e_1, e_2)$

Perform the join, watching for cases where we can merge the pending sets and then join all those variables.

This is where we pick up a bit more time complexity.

$\textbf{join}(e_1, e_2)$:
  let $t_1 = \textbf{type}(e_1)$
    $t_2 = \textbf{type}(e_2)$
    $e = \text{ecr-union}(e_1, e_2)$ in
  if $t_1 = \perp$ then
    $\textbf{type}(e) \leftarrow t_2$
    if $t_2 = \perp$ then
      $\textbf{pending}(e) \leftarrow \textbf{pending}(e_1) \cup \textbf{pending}(e_2)$
    else
      for $x \in \textbf{pending}(e_1)$ do $\textbf{join}(e, x)$
  else
    $\textbf{type}(e) \leftarrow t_1$
    if $t_2 = \perp$ then
      for $x \in \textbf{pending}(e_2)$ do $\textbf{join}(e, x)$
    else
      $\textbf{unify}(t_1, t_2)$

Joins

$$
\begin{array}{l}
\mathbf{cjoin}(e_1, e_2): \\
\quad \text{if } \mathbf{type}(e_2) = \bot \text{ then} \\
\quad\quad \mathbf{pending}(e_2) \leftarrow \{e_1\} \cup \mathbf{pending}(e_2) \\
\quad \text{else} \\
\quad\quad \mathbf{join}(e_1, e_2)
\end{array}
$$

$$
\boxed{
\begin{array}{l}
\mathbf{unify}(\mathbf{ref}(\tau_1 \times \lambda_1), \mathbf{ref}(\tau_2 \times \lambda_2)): \\
\quad \text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2) \\
\quad \text{if } \lambda_1 \neq \lambda_2 \text{ then } \mathbf{join}(\lambda_1, \lambda_2)
\end{array}
}
$$

Now unify the types by recursively joining their components.

$$
\begin{array}{l}
\mathbf{join}(e_1, e_2): \\
\quad \text{let } t_1 = \mathbf{type}(e_1) \\
\quad\quad t_2 = \mathbf{type}(e_2) \\
\quad\quad e = \text{ecr-union}(e_1, e_2) \text{ in} \\
\quad \text{if } t_1 = \bot \text{ then} \\
\quad\quad \mathbf{type}(e) \leftarrow t_2 \\
\quad\quad \text{if } t_2 = \bot \text{ then} \\
\quad\quad\quad \mathbf{pending}(e) \leftarrow \mathbf{pending}(e_1) \cup \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{pending}(e_2) \\
\quad\quad \text{else} \\
\quad\quad\quad \text{for } x \in \mathbf{pending}(e_1) \text{ do } \mathbf{join}(e, x) \\
\quad \text{else} \\
\quad\quad \mathbf{type}(e) \leftarrow t_1 \\
\quad\quad \text{if } t_2 = \bot \text{ then} \\
\quad\quad\quad \text{for } x \in \mathbf{pending}(e_2) \text{ do } \mathbf{join}(e, x) \\
\quad\quad \text{else} \\
\quad\quad\quad \mathbf{unify}(t_1, t_2)
\end{array}
$$

Rules

The paper builds rules for all the statements. The one we used in our prior example was this.

Don’t care

$$
\begin{array}{l}
\mathsf{x = \&y:} \\
\quad \text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{x})) \\
\qquad \tau_2 = \mathbf{ecr}(\mathsf{y}) \text{ in} \\
\qquad \text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2)
\end{array}
$$

Equivalence class representative

Rules

Consider: `y = &x;` $\mathbf{x}\colon \tau_4 = \mathbf{ref}(\bot \times \bot)$

$\mathbf{y}\colon \tau_5 = \mathbf{ref}(\tau_6 \times \bot)$

Substitute into the rule:

$\mathbf{x} = \&\mathbf{y}$:

$$\begin{aligned} &\text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathbf{x})) \\ &\qquad \tau_2 = \mathbf{ecr}(\mathbf{y}) \text{ in} \\ &\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2) \end{aligned}$$

$\Rightarrow$

$\mathbf{y} = \&\mathbf{x}$:

$$\begin{aligned} &\text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathbf{y})) \\ &\qquad \tau_2 = \mathbf{ecr}(\mathbf{x}) \text{ in} \\ &\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2) \end{aligned}$$

Rules

$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{ecr}(\mathbf{x}) = \tau_4$

$\mathbf{ecr}(\mathbf{y}) = \tau_5$

$\mathbf{type}(\mathbf{ecr}(\mathbf{y})) = \mathbf{type}(\tau_5) = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{y} = \&\mathbf{x}$:

$$\begin{aligned}
&\text{let } \mathbf{ref}(\tau_1 \times _) = \mathbf{type}(\mathbf{ecr}(\mathbf{y})) \\
&\qquad \tau_2 = \mathbf{ecr}(\mathbf{x}) \text{ in} \\
&\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2)
\end{aligned}$$

Rules

$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{ecr}(\mathbf{x}) = \tau_4$

$\mathbf{ecr}(\mathbf{y}) = \tau_5$

$\mathbf{type}(\mathbf{ecr}(\mathbf{y})) = \mathbf{type}(\tau_5) = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{y} = \&\mathbf{x}$:

$$\begin{aligned}&\text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathbf{y}))\\ &\qquad \tau_2 = \mathbf{ecr}(\mathbf{x}) \text{ in}\\ &\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2)\end{aligned}$$

$\Rightarrow$

$\mathbf{y} = \&\mathbf{x}$:

$$\begin{aligned}&\text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{ref}(\tau_6 \times \perp)\\ &\qquad \tau_2 = \tau_4 \text{ in}\\ &\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2)\end{aligned}$$

Rules

$\mathbf{x}$: $\tau_4 = \mathbf{ref}(\perp \times \perp)$
$\mathbf{y}$: $\tau_5 = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{ecr}(\mathbf{x}) = \tau_4$

$\mathbf{ecr}(\mathbf{y}) = \tau_5$

$\mathbf{type}(\mathbf{ecr}(\mathbf{y})) = \mathbf{type}(\tau_5) = \mathbf{ref}(\tau_6 \times \perp)$

$\mathbf{y} = \&\mathbf{x}$:

$$\begin{aligned} &\text{let } \mathbf{ref}(\tau_1 \times \_) = \mathbf{ref}(\tau_6 \times \perp) \\ &\qquad \tau_2 = \tau_4 \text{ in} \\ &\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{join}(\tau_1, \tau_2) \end{aligned}$$

$\Rightarrow$

$\mathbf{y} = \&\mathbf{x}$:

$$\text{if } \tau_6 \neq \tau_4 \text{ then } \mathbf{join}(\tau_6, \tau_4)$$

Rules

The other rules are similar.

x = y:

$$
\begin{array}{l}
\text{let } \mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x})) \\
\quad \mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y})) \text{ in} \\
\text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{cjoin}(\tau_1, \tau_2) \\
\text{if } \lambda_1 \neq \lambda_2 \text{ then } \mathbf{cjoin}(\lambda_1, \lambda_2)
\end{array}
$$

Note: This is where the inequality comes in.

Rules

The most complex rules are for functions.

This is the rule for function invocation.

$$
\begin{array}{l}
\mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)\text{:} \\
\quad \text{let } \mathbf{ref}(\_ \times \lambda) = \mathbf{type}(\mathbf{ecr}(\mathsf{p})) \text{ in} \\
\quad\quad \text{if } \mathbf{type}(\lambda) = \perp \text{ then} \\
\quad\quad\quad \mathbf{settype}(\lambda, \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\
\quad\quad\quad \text{where} \\
\quad\quad\quad\quad \alpha_i = \tau_i \times \lambda_i \\
\quad\quad\quad\quad [\tau_i, \lambda_i] = \mathbf{MakeECR}(2) \\
\quad\quad \text{let } \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda) \text{ in} \\
\quad\quad\quad \text{for } i \in [1 \ldots n] \text{ do} \\
\quad\quad\quad\quad \text{let } \tau_1 \times \lambda_1 = \alpha_i \\
\quad\quad\quad\quad\quad \mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}_i)) \text{ in} \\
\quad\quad\quad\quad \text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{cjoin}(\tau_1, \tau_2) \\
\quad\quad\quad\quad \text{if } \lambda_1 \neq \lambda_2 \text{ then } \mathbf{cjoin}(\lambda_1, \lambda_2) \\
\quad\quad\quad \text{for } i \in [1 \ldots m] \text{ do} \\
\quad\quad\quad\quad \text{let } \tau_1 \times \lambda_1 = \alpha_{n+i} \\
\quad\quad\quad\quad\quad \mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}_i)) \text{ in} \\
\quad\quad\quad\quad \text{if } \tau_1 \neq \tau_2 \text{ then } \mathbf{cjoin}(\tau_2, \tau_1) \\
\quad\quad\quad\quad \text{if } \lambda_1 \neq \lambda_2 \text{ then } \mathbf{cjoin}(\lambda_2, \lambda_1)
\end{array}
$$

How has it held up?

It still performs very well

Which Pointer Analysis Should I Use?

Michael Hind
IBM Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, USA
hind@watson.ibm.com

Anthony Pioli
Register.com
575 11th Ave
New York, NY 10018, USA
anthony@register.com

It has multiple implementations

“Alias Analysis in LLVM,”
Sheng-Hsiu Lin, M.S. 2015

Abstract

Alias analysis is a study of the relations between pointers. It has important applications in code optimization and security. This research introduces the fundamental concepts of alias analysis. It explains different approaches of alias analysis with examples. It provides a survey of some very important pointer analysis algorithms. LLVM interface is introduced along with the alias analyses that are currently available on it. This research implementes a Steensgaard’s pointer analysis on LLVM. The philosophy of this implementation is explained in detail. Evaluations of rule based basic alias analysis, Andersen’s pointer analysis, Steensgaard’s pointer analysis and data structure analysis are provided with experimental results on their precision, time and memory usage.

Homework
Due: Tuesday, April 14

Structuring 1

Starting with the provided solution to the basic block homework, or with your own solution, apply the constructive proof of the structure theorem. We will do this in pieces. For next time:

1. Find the entry point and see if it appears the C runtime is in use. If so, figure out where `main` is and add it to the addresses to extract.
2. Create a Python class for a Node. There should be three types of Node, possibly by subclass.
   a. A function node that holds a basic block that has a single next address or no next address ("unknown")
   b. A predicate node that holds a basic block that has two next addresses, one for true and one for false
   c. A label assignment node that holds an address to assign to the label
3. Package all basic blocks into Node instances. Don't worry about the label assignments yet.
4. Don't worry about output yet. Turn your program in by Tuesday.

Please name your program `structure.py`.

Next Time: More!