

Higher-order Flow Analysis with DDP

Olin Shivers¹

Northeastern University

Wand Symposium, August 2009

¹Work done with Dimitrios Vardoulakis and Lex Spoon

DDP: Dependency-directed with pruning

- ▶ Lex Spoon's PhD thesis
- ▶ Dependency-directed:
 - ▶ Don't solve *all* problems;
 - ▶ start with a particular goal & work backward.
- ▶ Lex's motivation:
interactive response in Squeak development environment
to a single user query
with 300,000-line code base.
 $O(n^3)$ algorithms not on.

The big idea

- ▶ AI goal-directed backwards search
- ▶ Prune search frontier with conservative oracle when search resources expended.

Algorithm constructs *subgoal graph*.

The bet: using oracle far away from root goal mitigates impact on precision. (*E.g.*: chess)

DDP: Dependency-directed with pruning

Problem

Smalltalk type analysis described in dissertation
unreadably complex.

DDP: Dependency-directed with pruning

Problem

Smalltalk type analysis described in dissertation unreadably complex.

Solution: DDP/CFA

Do a DDP analysis for the dumbest, simplest problem we can find.

DDP: Dependency-directed with pruning

Problem

Smalltalk type analysis described in dissertation unreadably complex.

Solution: DDP/CFA

Do a DDP analysis for the dumbest, simplest problem we can find.

Bonus

Retelling familiar tale in new setting sheds new light on interaction of semantic structures.

Informal algorithm

Note: all goals initially created with “perfect” answer.

Post initial goal onto worklist.

While worklist not empty:

 if not tired

 Remove query from worklist.

 Recompute query’s current answer & subgoals.

 If new subgoals, add them to worklist.

 Answer changed \Rightarrow add dependent super-goals to worklist.

 else

 Prune goal graph (and worklist)

 Answer goals at pruned fringe with oracle,
 and add their dependent goals to worklist

Core CPS language

Syntax

$$\begin{aligned} e \in EXP & ::= x \\ & \quad | (\lambda (x_1 \cdots x_n) \text{ call}) \\ \text{call} \in CALL & ::= (e_0 e_1 \cdots e_n) \end{aligned}$$

Core CPS language

Syntax

$$\begin{aligned} e \in EXP & ::= x \\ & \quad | (\lambda (x_1 \dots x_n) \text{ call}) \\ \text{call} \in CALL & ::= (e_0 e_1 \dots e_n) \end{aligned}$$

Labelled syntax

$$\begin{aligned} e \in EXP & ::= x \\ & \quad | (\lambda_\gamma (x_1 \dots x_n) \text{ call}) \\ \text{call} \in CALL & ::= \gamma(e_0 e_1 \dots e_n) \\ \gamma & \in Labels \end{aligned}$$

Contours factor the environment

$(\lambda_1 (a b)$

...

$(\lambda_2 (d)$

...

$(\lambda_3 (x y) \dots x \dots d \dots b \dots)$

...)

...)

Contour environments: λ -binder \rightarrow contour

$$\beta_1 = [\lambda_1 \mapsto 3, \lambda_2 \mapsto 35, \lambda_3 \mapsto 41]$$

$$\beta_2 = [\lambda_1 \mapsto 3, \lambda_2 \mapsto 17, \lambda_3 \mapsto 105]$$

Variable environment: variable \times contour \rightarrow value

$$ve = \left[\begin{array}{l} \langle a, 3 \rangle \mapsto 0, \langle b, 3 \rangle \mapsto \text{false}, \\ \langle d, 35 \rangle \mapsto 3.1415, \langle d, 17 \rangle \mapsto \text{"fred"}, \dots \end{array} \right]$$

Eval-to-Apply Transition

$$\frac{proc = \mathcal{A}(f, \beta, ve) \quad d_i = \mathcal{A}(e_i, \beta, ve)}{(\llbracket (f e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \mathbf{d}, ve, t + 1)}$$

Apply-to-Eval Transition

$$\frac{proc = (\llbracket (\lambda (v_1 \cdots v_n) call) \rrbracket, \beta')}{(proc, \mathbf{d}, ve, t) \Rightarrow (call, \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d_i], t)}$$

Domains

| | | | | |
|-------------|-------|--------------|-----|--|
| ς | \in | <i>Eval</i> | $=$ | $CALL \times BEnv \times VEnv \times Time$ |
| $+$ | | <i>Apply</i> | $=$ | $Proc \times D^* \times VEnv \times Time$ |
| β | \in | <i>BEnv</i> | $=$ | $VAR \rightarrow Time$ |
| ve | \in | <i>VEnv</i> | $=$ | $VAR \times Time \rightarrow D$ |
| $proc$ | \in | <i>Proc</i> | $=$ | $Clo + \{halt\}$ |
| clo | \in | <i>Clo</i> | $=$ | $LAM \times BEnv$ |
| d | \in | <i>D</i> | $=$ | <i>Proc</i> |
| t | \in | <i>Time</i> | $=$ | infinite set of times (contours) |

Argument eval

$$\begin{aligned} \mathcal{A}(lam, \beta, ve) \\ = (lam, \beta) \end{aligned}$$

$$\begin{aligned} \mathcal{A}(v, \beta, ve) \\ = ve(v, \beta(v)) \end{aligned}$$

Eval-to-Apply Transition

$$\frac{\widehat{proc} \in \widehat{A}(f, \widehat{\beta}, \widehat{ve}) \quad \widehat{d}_i = \widehat{A}(e_i, \widehat{\beta}, \widehat{ve})}{([\![f e_1 \cdots e_n]\!] , \widehat{\beta}, \widehat{ve}, \widehat{t}) \approx (\widehat{proc}, \widehat{d}, \widehat{ve}, \widehat{succ}(\widehat{t}))}$$

Apply-to-Eval Transition

$$\frac{\widehat{proc} = ([\!(\lambda (v_1 \cdots v_n) call)\!] , \widehat{\beta}')}{(\widehat{proc}, \widehat{d}, \widehat{ve}, \widehat{t}) \approx (call, \widehat{\beta}'[v_i \mapsto \widehat{t}], \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i], \widehat{t})}$$

Domains

$$\begin{aligned}\widehat{s} &\in \widehat{Eval} = \widehat{CALL} \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\ &+ \widehat{Apply} = \widehat{Proc} \times \widehat{D}^* \times \widehat{VEnv} \times \widehat{Time} \\ \widehat{\beta} &\in \widehat{BEnv} = \widehat{VAR} \rightarrow \widehat{Time} \\ \widehat{ve} &\in \widehat{VEnv} = \widehat{VAR} \times \widehat{Time} \rightarrow \widehat{D} \\ \widehat{proc} &\in \widehat{Proc} = \widehat{Clo} + \{halt\} \\ \widehat{clo} &\in \widehat{Clo} = \widehat{LAM} \times \widehat{BEnv} \\ \widehat{d} &\in \widehat{D} = \mathcal{P}(\widehat{Proc}) \\ \widehat{t} &\in \widehat{Time} = \text{finite set of times (contours)}\end{aligned}$$

Argument eval

$$\begin{aligned}\widehat{A}(lam, \widehat{\beta}, \widehat{ve}) \\ &= \{(lam, \widehat{\beta})\} \\ \widehat{A}(v, \widehat{\beta}, \widehat{ve}) \\ &= \widehat{ve}(v, \widehat{\beta}(v))\end{aligned}$$

Eval-to-Apply Transition

$$\frac{\widehat{proc} \in \widehat{A}(f, \widehat{\beta}, \widehat{ve}) \quad \widehat{d}_i = \widehat{A}(e_i, \widehat{\beta}, \widehat{ve})}{([\![f e_1 \cdots e_n]\!] , \widehat{\beta}, \widehat{ve}, \widehat{t}) \approx (\widehat{proc}, \widehat{d}, \widehat{ve}, \widehat{succ}(\widehat{t}))}$$

Apply-to-Eval Transition

$$\frac{\widehat{proc} = ([\!(\lambda (v_1 \cdots v_n) call)\!] , \widehat{\beta}')}{(\widehat{proc}, \widehat{d}, \widehat{ve}, \widehat{t}) \approx (call, \widehat{\beta}'[v_i \mapsto \widehat{t}], \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i], \widehat{t})}$$

Domains

$$\begin{aligned} \widehat{s} &\in \widehat{Eval} = \widehat{CALL} \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\ &+ \widehat{Apply} = \widehat{Proc} \times \widehat{D}^* \times \widehat{VEnv} \times \widehat{Time} \\ \widehat{\beta} &\in \widehat{BEnv} = \widehat{VAR} \rightarrow \widehat{Time} \\ \widehat{ve} &\in \widehat{VEnv} = \widehat{VAR} \times \widehat{Time} \rightarrow \widehat{D} \\ \widehat{proc} &\in \widehat{Proc} = \widehat{Clo} + \{halt\} \\ \widehat{clo} &\in \widehat{Clo} = \widehat{LAM} \times \widehat{BEnv} \\ \widehat{d} &\in \widehat{D} = \mathcal{P}(\widehat{Proc}) \\ \widehat{t} &\in \widehat{Time} = \text{finite set of times (contours)} \end{aligned}$$

Argument eval

$$\begin{aligned} \widehat{A}(lam, \widehat{\beta}, \widehat{ve}) \\ &= \{(lam, \widehat{\beta})\} \\ \widehat{A}(v, \widehat{\beta}, \widehat{ve}) \\ &= \widehat{ve}(v, \widehat{\beta}(v)) \end{aligned}$$

Eval-to-Apply Transition OCFA

$$\frac{\widehat{proc} \in \widehat{A}(f, \widehat{ve}) \quad \widehat{d}_i = \widehat{A}(e_i, \widehat{ve})}{(\llbracket (f \ e_1 \cdots e_n) \rrbracket, \widehat{ve}) \approx \approx (\widehat{proc}, \widehat{\mathbf{d}}, \widehat{ve})}$$

Apply-to-Eval Transition

$$\frac{}{(\llbracket (\lambda (v_1 \cdots v_n) \ call) \rrbracket, \widehat{\mathbf{d}}, \widehat{ve}) \approx \approx (\call, \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i])}$$

Domains

$$\begin{aligned} \widehat{\varsigma} &\in \widehat{Eval} = \widehat{CALL} \times \widehat{VEnv} \\ &+ \widehat{Apply} = \widehat{Proc} \times \widehat{D}^* \times \widehat{VEnv} \\ \widehat{ve} &\in \widehat{VEnv} = \widehat{VAR} \rightarrow \widehat{D} \\ \widehat{proc} &\in \widehat{Proc} = \widehat{Clo} + \{\text{halt}\} \\ \widehat{clo} &\in \widehat{Clo} = \widehat{LAM} \\ \widehat{d} &\in \widehat{D} = \mathcal{P}(\widehat{Proc}) \\ \widehat{t} &\in \widehat{Time} = \{0\} \end{aligned}$$

Argument eval

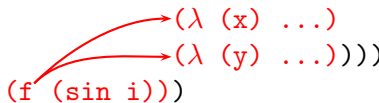
$$\begin{aligned} \widehat{A}(\text{lam}, \widehat{ve}) &= \{\text{lam}\} \\ \widehat{A}(v, \widehat{ve}) &= \widehat{ve}(v) \end{aligned}$$

Control flow

```
(let ((f (if (< i 0)
             (λ (x) ...)
             (λ (y) ...))))
      (f (sin i)))
```

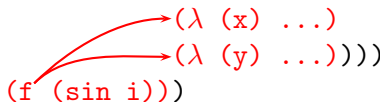
Control flow

```
(let ((f (if (< i 0)
             (λ (x) ...)
             (λ (y) ...))))
      (f (sin i)))
```

The diagram consists of two red arrows. The first arrow starts at the 'f' argument of the function call '(f (sin i))' and points to the lambda function '(λ (x) ...)' in the 'let' expression. The second arrow starts at the 'f' argument and points to the lambda function '(λ (y) ...)' in the 'let' expression.

Control flow

```
(let ((f (if (< i 0)
             (λ (x) ...)
             (λ (y) ...))))
      (f (sin i)))
```



...but we could add context for precision:

$$\begin{aligned} \llbracket (f \text{ (sin } i)) \rrbracket, \beta_1 &\xrightarrow{c} \llbracket (\lambda (x) \dots) \rrbracket, \beta_2 \\ \llbracket (f \text{ (sin } i)) \rrbracket, \beta_3 &\xrightarrow{c} \llbracket (\lambda (y) \dots) \rrbracket, \beta_4 \end{aligned}$$

Data flow: classic version

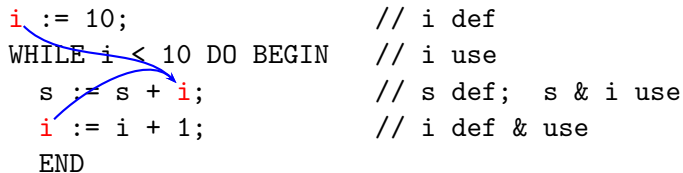
Data flows from **def** to **use** to **def**...

```
i := 10;           // i def
WHILE i < 10 DO BEGIN // i use
  s := s + i;      // s def; s & i use
  i := i + 1;      // i def & use
END
```

Data flow: classic version

Data flows from **def** to **use** to **def**...

```
i := 10;           // i def
WHILE i < 10 DO BEGIN // i use
  s := s + i;     // s def; s & i use
  i := i + 1;     // i def & use
END
```



Data flow: λ -calculus version

Data flows from **binding** to **reference** to **binding**...

```
( $\lambda$  (i f)
  (if (< i 0)
      (f i)
      (- i)))
```


...

```
( $\lambda$  (j) (- (sin j))) ; passed as f
```

Data flow: λ -calculus version

Data flows from **binding** to **reference** to **binding**...

```
( $\lambda$  (i f)
  (if (< i 0)
      (f i)
      (- i)))
```



...

```
( $\lambda$  (j) (- (sin j))) ; passed as f
```

Data flow: λ -calculus version

Data flows from **binding** to **reference** to **binding**...

```
(λ (i f)
  (if (< i 0)
      (f i)
      (- i)))
...
(λ (j) (- (sin j))) ; passed as f
```

Data flow: λ -calculus version

Data flows from **binding** to **reference** to **binding**...

```
( $\lambda$  (i f)
  (if (< i 0)
      (f i)
      (- i)))
...
( $\lambda$  (j) (- (sin j))) ; passed as f
```

A blue arrow starts at the red 'i' in the lambda binding of the first function, goes down to the red 'i' in the function call '(f i)', and then continues down to the red 'j' in the lambda binding of the second function '(lambda (j) ...)'. This illustrates the flow of data from a binding to a reference and then to another binding.

... but we could add context for precision:

$$\begin{aligned} \text{def}(\llbracket \mathbf{i} \rrbracket, \beta_1) &\xrightarrow{D} \text{use}(\llbracket (- \mathbf{i}) \rrbracket, 1, \beta_1) \\ \text{use}(\llbracket (\mathbf{f} \mathbf{i}) \rrbracket, 1, \beta_2) &\xrightarrow{D} \text{def}(\llbracket \mathbf{j} \rrbracket, \beta_3) \end{aligned}$$

Three flow relations: control, data & env

Data-flow: $fp \xrightarrow{D} fp'$

| | | | |
|---------------|-------|-----------------------|---------------------------------------|
| $fp \in FPos$ | $::=$ | $use(call, i, \beta)$ | Position i of $call$ in env β |
| | | $def(e, \beta)$ | Expression e in env β |

Control-flow: $(call, \beta) \xrightarrow{C} (lam, \beta')$

Env-flow: $\beta \xrightarrow{E} \beta[\gamma \mapsto contour]$

- ▶ Capture three different slices of behaviour.
- ▶ Mutually dependent!
- ▶ Classic cflow & dflow are “OCFA” abstractions.

Deriving data-flow relation from state trace

$$\text{Trace} = s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \dots$$

$$\frac{(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, \beta, ve, t) \in \text{States}}{\text{def}(e_i, \llbracket \beta \rrbracket_{e_i}) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta)}$$

$$\frac{(\text{call}, \beta, ve, t) \Rightarrow ((\llbracket \lambda(x_1 \dots x_n) \text{call}' \rrbracket), \beta'), \mathbf{d}, ve', t')}{\text{use}(\text{call}, i, \beta) \xrightarrow{D} \text{def}(x_i, \beta'[\gamma' \mapsto t'])}$$

Flow relations, inductively (sorta)

$$\frac{\text{def}(lam, \beta) \xrightarrow{D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

Flow relations, inductively (sorta)

$$\frac{\text{def}(lam, \beta) \xrightarrow{D} \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

$$\frac{\beta \xrightarrow{E} \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \lfloor \beta \rfloor_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

Flow relations, inductively (sorta)

$$\frac{\text{def}(lam, \beta) \xrightarrow{D} \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \lfloor \beta \rfloor_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

$$\frac{(call, \beta) \xrightarrow{C} (lam_\gamma, \beta')}{\beta' \xrightarrow{E} \beta'[\gamma \mapsto t]}$$

Flow relations, inductively (sorta)

$$\frac{\text{def}(lam, \beta) \xrightarrow{D} \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

$$\frac{\beta \xrightarrow{E} \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \llbracket \beta \rrbracket_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

$$\frac{(call, \beta) \xrightarrow{C} (lam_\gamma, \beta')}{\beta' \xrightarrow{E} \beta'[\gamma \mapsto t]}$$

$$\frac{(call, \beta) \xrightarrow{C} (\llbracket \lambda_{\gamma'}(x_1 \dots x_n) call' \rrbracket, \beta')}{\text{use}(call, i, \beta) \xrightarrow{D} \text{def}(x_i, \beta'[\gamma' \mapsto t])}$$

With less math. . .

cflow induced by dflow

dflow induced by eflow

eflow induced by cflow

With less math. . .

cflow induced by dflow

dflow induced by eflow def→use

eflow induced by cflow

With less math. . .

cflow induced by dflow

dflow induced by eflow def→use

eflow induced by cflow

dflow induced by cflow use→def

Approximation and propagation direction

Right approximation: $fp \xrightarrow{D} fp'$

fp might flow to fp'

Left approximation: $fp \xleftarrow{D} fp'$

fp' might flow from fp

From rule to backwards-chained queries: control flow

Rule

$$\frac{\text{def}(lam, \beta) \xrightarrow{D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

Queries

$$\frac{\text{def}(lam, \beta) \xrightarrow{\bullet D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{\bullet C} (lam, \beta)}$$

$$\frac{\text{def}(lam, \beta) \xrightarrow{\bullet D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{\bullet C} (lam, \beta)}$$

From rule to backwards-chained queries: control flow

Rule

$$\frac{\text{def}(lam, \beta) \xrightarrow{D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} (lam, \beta)}$$

Queries

$$\frac{\text{def}(lam, \beta) \bullet \xrightarrow{D}^+ \text{use}(call, 0, \beta')}{(call, \beta') \xrightarrow{C} \bullet (lam, \beta)}$$

$$\frac{\text{def}(lam, \beta) \xrightarrow{D} \bullet^+ \text{use}(call, 0, \beta')}{(call, \beta') \bullet \xrightarrow{C} (lam, \beta)}$$

From rule to backwards-chained queries: data-flow

Rule

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket_{\gamma}(e_0 \ e_1 \dots e_n)\rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \lfloor \beta \rfloor_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

Queries

$$\frac{\beta \xrightarrow{\bullet E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{\bullet D} \text{use}(\llbracket_{\gamma}(e_0 \ e_1 \dots e_n)\rrbracket, i, \beta')} \quad \begin{array}{l} \beta' \text{ covers call } \gamma \\ e_i \in \text{Var} \end{array}$$

From rule to backwards-chained queries: data-flow

Rule

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \lfloor \beta \rfloor_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

Queries

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \bullet \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta' \text{ covers call } \gamma \\ e_i \in \text{Var} \end{array}$$

$$\frac{}{\text{def}(e_i, \beta) \xrightarrow{D} \bullet \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta)} \quad e_i \in \text{Lam}$$

From rule to backwards-chained queries: data-flow

Rule

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta = \lfloor \beta \rfloor_{e_i} \\ \beta' \text{ covers call } \gamma \end{array}$$

Queries

$$\frac{\beta \xrightarrow{E}^* \beta'}{\text{def}(e_i, \beta) \xrightarrow{D} \bullet \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta')} \quad \begin{array}{l} \beta' \text{ covers call } \gamma \\ e_i \in \text{Var} \end{array}$$

$$\frac{}{\text{def}(e_i, \beta) \xrightarrow{D} \bullet \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta)} \quad e_i \in \text{Lam}$$

$$\frac{}{\text{def}(e_i, \lfloor \beta \rfloor_{e_i}) \bullet \xrightarrow{D} \text{use}(\llbracket \gamma(e_0 \ e_1 \dots e_n) \rrbracket, i, \beta)}$$

From rule to backwards-chained queries: env-flow

Rule

$$\frac{(call, \beta) \xrightarrow{C} (lam_{\gamma}, \beta')}{\beta' \xrightarrow{E} \beta'[\gamma \mapsto t]}$$

Queries (1CFA version)

$$\frac{(call_{\gamma'}, \beta') \bullet \xrightarrow{C} (lam_{\gamma}, \beta)}{\beta \xrightarrow{E} \bullet \beta[\gamma \mapsto \gamma']} \quad lam_{\gamma} \in \text{InnerLam}(\beta).call.args$$

$$\frac{}{[\beta]_1 \bullet \xrightarrow{E} \beta}$$

From rule to backwards-chained queries: data-flow

Rule

$$\frac{(call, \beta) \xrightarrow{C} (\llbracket \lambda_{\gamma'} (x_1 \dots x_n) call' \rrbracket, \beta')}{use(call, i, \beta) \xrightarrow{D} def(x_i, \beta'[\gamma' \mapsto t])}$$

Queries (1CFA version)

$$\frac{(call_{\gamma}, \beta) \xrightarrow{C} (\llbracket \lambda_{\gamma'} (x_1 \dots x_n) call' \rrbracket, \beta')}{use(call, i, \beta) \xrightarrow{D} def(x_i, \beta'[\gamma' \mapsto \gamma])}$$

$$\frac{(call, \beta) \xrightarrow{\bullet C} (\llbracket \lambda (x_1 \dots x_n) call' \rrbracket, \llbracket \beta' \rrbracket_1)}{use(call, i, \beta) \xrightarrow{\bullet D} def(x_i, \beta')}$$

Multi-step closure queries

Data flow / backward

$$\frac{\text{def}(lam, \beta) \xrightarrow{D} \text{use}(call, i, \beta')}{\text{def}(lam, \beta) \xrightarrow{D^+} \text{use}(call, i, \beta')}$$

$$\frac{\text{def}(lam, \beta_4) \xrightarrow{D^+} \text{use}(call', j, \beta_3) \quad \text{use}(call', j, \beta_3) \xrightarrow{D} \text{def}(x, \beta_2) \quad \text{def}(x, \beta_2) \xrightarrow{D} \text{use}(call, i, \beta_1)}{\text{def}(lam, \beta_4) \xrightarrow{D^+} \text{use}(call, i, \beta_1)}$$

Oracle for pruning

In typed language, can use function & call types.

In Scheme, can still use function & call arities.

Smalltalk method names quite selective.

Todo

- ▶ Do it for CPA
- ▶ Theorems & proofs for 1CFA & CPA
- ▶ Try it out, test precision.