Representation analysis

How to represent closure's lexical environment?

Option 1: **deep binding**

- represent environment as pointer to shared env record

+ simple
+ space-efficient
+ fast to create closure
– slow to access lexically-enclosing vars

Example:

```
(define (a i j)
  (define (b m n o)
    (define (c q)
      (+ q m i))
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```


Representation analysis, cont

Option 2: **shallow binding**

- copy needed values into closure when created

+ fast access to enclosing vars
+ may not need to heap-allocate enclosing environment
– slower closure creation
– more space consuming, if >1 var needed
– doesn't work for mutable vars
    - apply assignment conversion first

Example:

```
(define (a i j)
  (define (b m n o)
    (define (c q)
      (+ q m i))
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```


Control flow analysis for higher-order programs

If have first-class functions, then calls can invoke computed fns

- hard to build call graph, do interprocedural analysis

Could make worst-case assumptions

- not effective if heavy use of computed fns

Alternative [Shivers 88]:

- do analysis to determine possible values of called exprs

Circularity dilemma

Problem:

- to do interprocedural analysis, need a call graph
- to construct a call graph, need to know possible values of called expressions
- to know values of expressions, need to do analysis
- ...

How to break vicious cycle?

An algorithm

The task:
for each expression in the program,
compute the set of functions
that the expression might evaluate to

Analysis framework: **abstract interpretation**

Define syntax & denotational semantics for language

- lattice-theoretic domains used in den. semantics

Replace denotational semantics with
abstract non-standard semantics

- compute abstract property, not real value
  - **collecting interpretation**:
    compute set of possible values, not individual values
- prove abstraction is a sound approximation of real value
- set up abstraction so it's computable

+ syntax-directed analysis, not graph-based
+ easier to prove soundness of analysis

A simple language

$$
\begin{array}{lll}
E & ::= & \mathit{const} \\
 & | & \mathit{var} \\
 & | & \mathit{primop} \\
 & | & \textbf{(lambda (}\mathit{var}_{formal}\textbf{*) } E_{body}\textbf{)} \\
 & | & \textbf{(if } E_{test}\ E_{then}\ E_{else}\textbf{)} \\
 & | & \textbf{(letrec ((}\mathit{var}_{name}\ E_{value}\textbf{)*) } E_{body}\textbf{)} \\
 & | & \textbf{(}E_{fn}\ E_{arg}\textbf{*)}
\end{array}
$$

Constants: `3`, `4.5`, `"hi"`, `'foo`, `'(a list of length 5)`

Primops: `+`, `eq`, `cons`, `car`, `cdr`, `...`

The abstract semantics

Syntactic domains:

Expr: expressions in program

Var: variables in program (assumed unique)

Non-standard semantic domains:

Fn: functions in program (includes primops)

Fns = 2^{Fn}: sets of functions

CG = Expr $\rightarrow$ Fns: callees for call expr

Helper semantic domains:

Env = Var $\rightarrow$ Fns: bindings from vars to sets of functions

Computing result fns of expressions

Call graph for program:

$\mathcal{P}rog$: Expr $\rightarrow$ CG

$\mathcal{P}rog\,[\![E]\!] \equiv$
let <fns,cg> = $\mathcal{F}\,[\![E]\!]\,\varnothing$ in
cg

Functions & call graph returned by an expression:

$\mathcal{F}$: Expr $\times$ Env $\rightarrow$ Fns $\times$ CG

$\mathcal{F}\,[\![\mathit{const}]\!]\,\rho \equiv \langle\varnothing, \varnothing\rangle$

$\mathcal{F}\,[\![\mathit{var}]\!]\,\rho \equiv \langle\rho(\mathit{var}), \varnothing\rangle$

$\mathcal{F}\,[\![\mathit{primop}]\!]\,\rho \equiv \langle\{\mathit{primop}\}, \varnothing\rangle$

$\mathcal{F}\,[\![\texttt{(lambda (...) ...)}]\!]\,\rho \equiv$
$\langle\{\texttt{(lambda (...) ...)}\}, \varnothing\rangle$

$\mathcal{F}\,[\![\texttt{(if } E_1\ E_2\ E_3\texttt{)}]\!]\,\rho \equiv$
let $\langle fns_1,cg_1\rangle = \mathcal{F}\,[\![E_1]\!]\,\rho$
$\langle fns_2,cg_2\rangle = \mathcal{F}\,[\![E_2]\!]\,\rho$
$\langle fns_3,cg_3\rangle = \mathcal{F}\,[\![E_3]\!]\,\rho$ in
$\langle fns_2 \cup fns_3,\ cg_1 \cup cg_2 \cup cg_3\rangle$

$\mathcal{F}[\![$`(letrec ((`var_1 E_1`) ... (`var_n E_n`))` E_{body}`)`$]\!]\ \rho \equiv$
let $\langle fns_i, cg_i\rangle = \mathcal{F}[\![E_i]\!]\ \rho'$
where $\rho' = \rho[var_i \to fns_i]$ in
let $\langle fns_{body}, cg_{body}\rangle = \mathcal{F}[\![E_{body}]\!]\ \rho'$ in
$\langle fns_{body}, cg_1 \cup ... \cup cg_n \cup cg_{body}\rangle$

$\mathcal{F}[\![$`(`E_{fn} E_1 ... E_n`)`$]\!]\ \rho \equiv$
let $\langle fns_{fn}, cg_{fn}\rangle = \mathcal{F}[\![E_{fn}]\!]\ \rho$
$\langle fns_i, cg_i\rangle = \mathcal{F}[\![E_i]\!]\ \rho$ in
let $\langle fns, cg\rangle = \bigcup_{f \in fns_{fn}} \mathcal{A}\ f\ \langle fs_1, ..., fs_n\rangle\ \rho$ in
$\langle fns, cg_{fn} \cup cg_1 \cup ... \cup cg_n \cup cg\rangle$

$\mathcal{A}$: Fn × Fns* × Env → Fns × CG

Applying functions

$\mathcal{A}$: Fn × Fns* × Env → Fns × CG

Primops:
$\mathcal{A}[\![+]\!]\ \langle fs_1, fs_2\rangle\ \rho \equiv \langle\emptyset, \emptyset\rangle$

$\mathcal{A}[\![$`cons`$]\!]\ \langle fs_1, fs_2\rangle\ \rho \equiv \langle\emptyset, \emptyset\rangle$
- side-effect: *escaped* := *escaped* $\cup\ fs_1 \cup fs_2$
  (really, *escaped* as additional result of $\mathcal{F}$, $\mathcal{A}$)

$\mathcal{A}[\![$`car`$]\!]\ \langle fs_1\rangle\ \rho \equiv \langle$*escaped*$, \emptyset\rangle$

User-defined fns:
$\mathcal{A}[\![$`(lambda (`$v_1 ... v_n$`)` E`)`$]\!]\ \langle fs_1, ..., fs_n\rangle\ \rho \equiv$
$\mathcal{F}[\![E]\!]\ \rho'$
where ρ' = *lexical_env*$[v_i \to fs_i]$
- where to get *lexical_env*?
- what about recursion?
  ⇒ abstract semantics not computable

Approximation

One abstract binding per source variable,
one abstract activation record per procedure
- recursions folded together ⇒ computable approximation
- don't need to record lexically-enclosing env

$\mathcal{A}[\![$`(lambda (`$v_1 ... v_n$`)` E`)`$]\!]\ \langle fs_1, ..., fs_n\rangle\ \rho =$
$\rho(v_i) := \rho(v_i) \cup fs_i$;
$\mathcal{F}[\![E]\!]\ \rho$

Solve iteratively:
- initially assume empty fns set for each variable
- keep enlarging sets until no more change

Example

```
(define (a)
  (b (lambda1 (f) (c f))))

(define (b m)
  (c (lambda2 (g) (d d)))
  (m d))

(define (c q)
  (q q))

(define (d s)
  (c (lambda3 (w) s)))

(a)
```

Example after desugaring **`define`**'s

```
(letrec (
  (a (lambda_a ()
    (b (lambda_1 (f) (c f)))))
  (b (lambda_b (m)
    (c (lambda_2 (g) (d d)))
    (m d)))
  (c (lambda_c (q)
    (q q)))
  (d (lambda_d (s)
    (c (lambda_3 (w) s))))
  )
  (a))
```