# Removing Accidental Traversal Complexity from Programs

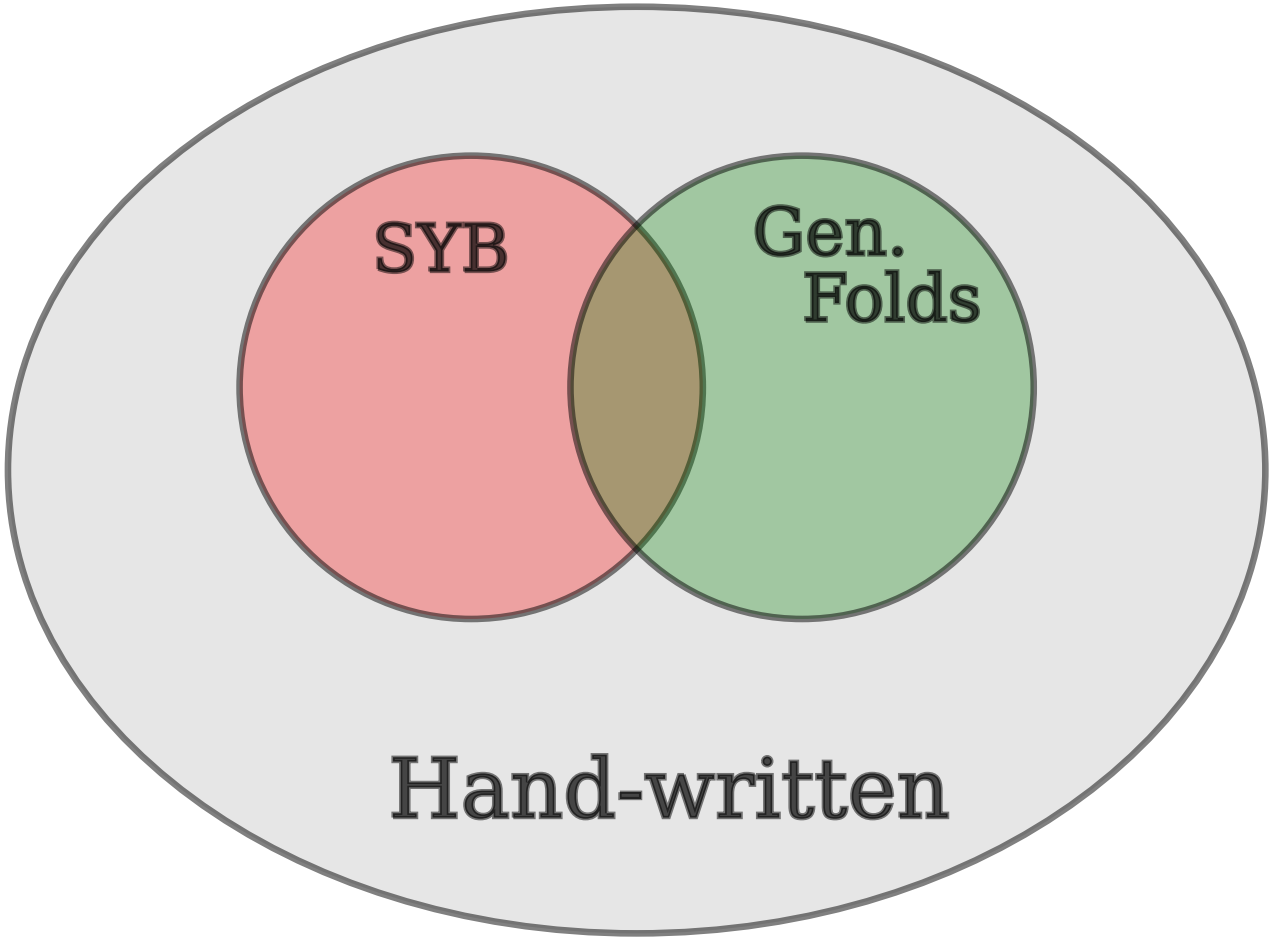
Bryan Chadwick

PL Seminar
April 23rd

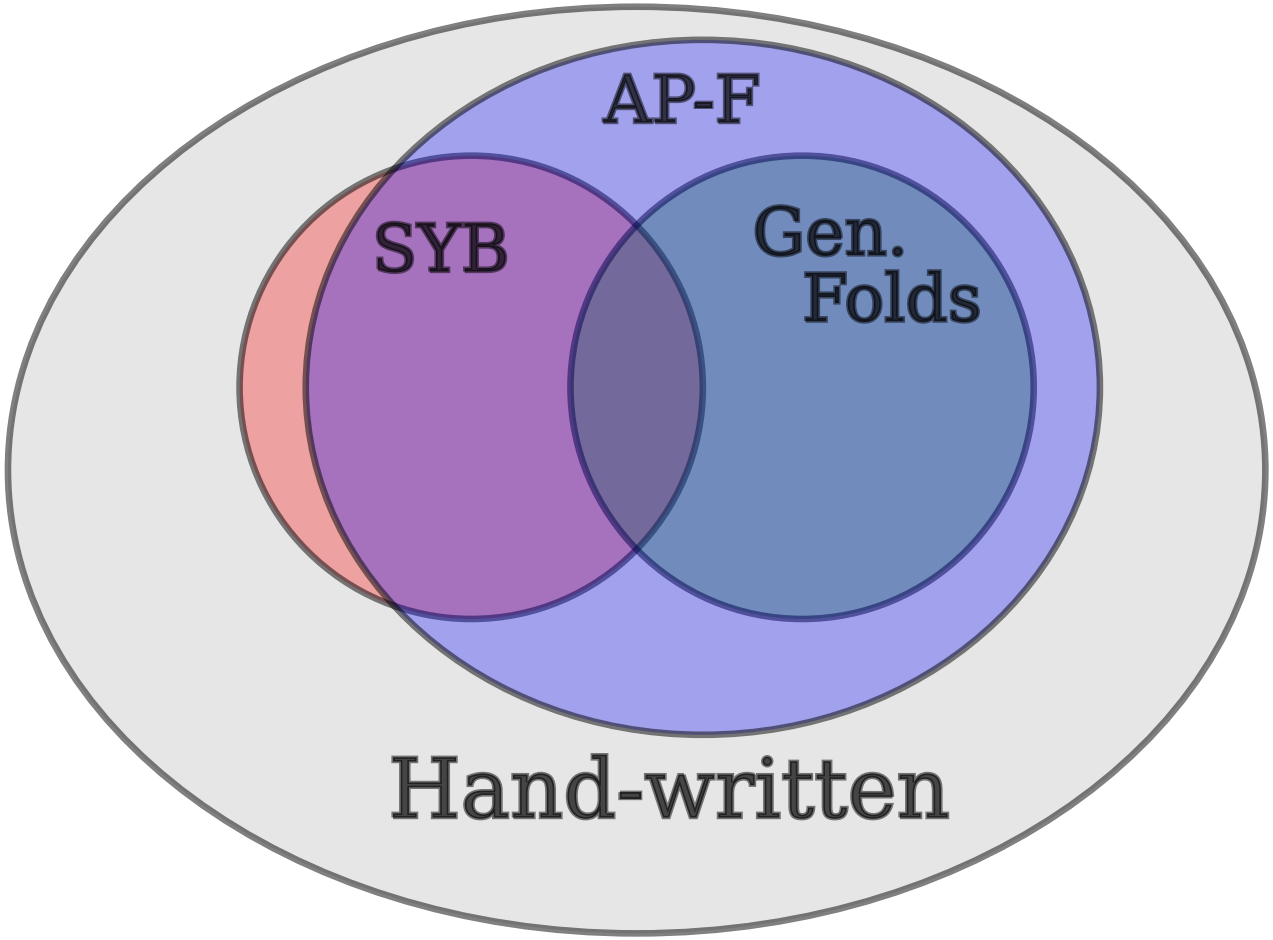
# AP-F

* Flexibility
  - Implicit traversal
  - Closer to hand written
* Modularity
  - Abstraction/decomposition
  - Building up function sets
* A Solution to the "*Expression Problem*"
  - "Extension" of functionality and data structures

# Where We Fit



# Where We Fit



# Outline

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

* Why use `fold`, `map`, and others?

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

* Why use `fold`, `map`, and others?
  + Easier... but why?

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

* Why use `fold`, `map`, and others?
  + Easier... but why?
  + Eliminate list "field" accesses (`car`/`cdr`)

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

* Why use `fold`, `map`, and others?
  + Easier... but why?
  + Eliminate list "field" accesses (`car`/`cdr`)
  + Eliminate explicit recursion

# Traversals: Scheme Lists

* Structural Recursion

```
;; fold-r : (X Y -> Y) Y [listof X] -> Y
(define (fold-r xy->y y lox)
  (cond [(null? lox) y]
        [else (xy->y (car lox)
                     (fold-r xy->y y (cdr lox)))]))
```

* Why use `fold`, `map`, and others?
  + Easier... but why?
  + Eliminate list "field" accesses (`car`/`cdr`)
  + Eliminate explicit recursion
* Abstract... hide structure!

# Motivation

* Want a "*Write-Once*" Traversal
* Flexible (solve many problems, like `fold`)
* Handle multi-dimensional structures consistently
* Make things easier (shorthand)

# Quick Example: Lambda Terms

```
;; An exp is either:
;;   -- (make-var-e symbol)
;;   -- (make-lambda-e symbol exp)
;;   -- (make-app-e exp exp)
(define-struct var-e (id))
(define-struct lambda-e (id body))
(define-struct app-e (rator rand))
```

# Quick Example: Lambda Terms

```
;; An exp is either:
;;   -- (make-var-e symbol)
;;   -- (make-lambda-e symbol exp)
;;   -- (make-app-e exp exp)
(define-struct var-e (id))
(define-struct lambda-e (id body))
(define-struct app-e (rator rand))
```

Calculate the free variables of a term:

```
;; free-vars : exp -> [setof symbol]
;; Collect the free variables in an expression
(define (free-vars e)
  (cond [(var-e? e)    (set-single (var-e-id e))]
        [(lambda-e? e) (set-rm (free-vars (lambda-e-body e))
                               (lambda-e-id e))]
        [(app-e? e)    (set-union (free-vars (app-e-rator e))
                                  (free-vars (app-e-rand e)))]))
```

# Quick Example: Lambda Terms

```
;; An exp is either:
;;   -- (make-var-e symbol)
;;   -- (make-lambda-e symbol exp)
;;   -- (make-app-e exp exp)
(define-struct var-e (id))
(define-struct lambda-e (id body))
(define-struct app-e (rator rand))
```

Calculate the free variables of a term:

```
;; free-vars : exp -> [setof symbol]
;; Collect the free variables in an expression
(define (free-vars e)
  (cond [(var-e? e)    (set-single (var-e-id e))]
        [(lambda-e? e) (set-rm (free-vars (lambda-e-body e))
                               (lambda-e-id e))]
        [(app-e? e)    (set-union (free-vars (app-e-rator e))
                                  (free-vars (app-e-rand e)))]))

;; AP-F traversal free variable calculation
(define (free-vars-apf e)
  (let ((B (func-set
            [(var-e symbol)        (v id)      (set-single id)]
            [(lambda-e symbol set) (l id fv)   (set-rm fv id)]
            [(app-e set set)       (c fvl fvr) (set-union fvl fvr)])))
    (traverse-b e B)))
```

# Outline

# Traversal Abstraction

* Generic traversal function
* 3 function sets: `F`, `B`, and `A`
* Control function: `C`
* Custom type-based dispatch

# General Traversal

Primitives: `number`, `string`, etc...

- Just apply `F`

Structures:

- Update the traversal argument
- For each Field:

  If `C` returns `#t`, recursively traverse
  Otherwise apply `F`

- Use `B` to rebuild the structure

# Traversal Example: map

```
;; map-t : (X -> Y) [listof X] -> [listof Y]
;; Map for any dimensional lists of primitives
(define (map-t x->y lst)
  (traverse lst
            ;; F
            (func-set [(object) (x) (x->y x)])
            ;; B
            (func-set [(empty) (e) e]
                      [(cons object list) (c f r) (cons f r)])
            ;; A
            (func-set [(object object) (o arg) arg])
            ;; C
            (lambda (obj i) #t)
            ;; traversal argument (ignored)
            0))

(map-t add1 '(1 2 3))  ;; ==> '(2 3 4)
(map-t sub1 '(1 2 3))  ;; ==> '(0 1 2)

(map-t add1 '(((1)) ((2 (3)))));; ==> '(((2)) ((3 (4))))
```

# Traversal Example: map

```
;; map-t : (number -> Y) [listof number] -> [listof Y]
;; Map for lists of numbers.
;;  * Really, it works for any structure containing numbers
(define (map-t n->y lon)
  (traverse-f lon (union-idF [(number) (n) (n->y n)])))

(map-t add1 '(1 2 3))  ;; ==> '(2 3 4)
(map-t sub1 '(1 2 3))  ;; ==> '(0 1 2)

(map-t add1 '(((1)) ((2 (3)))));; ==> '(((2)) ((3 (4))))
```

# Function Dispatch: `delta`

* Sets of *typed* functions
* Compares formal/actual types
* *Best* one wins
* Applies to a prefix of arguments

# Definition: better?

```
;; better? : Function Function -> boolean
;; Is the first function 'better' than the second
(define (better? f1 f2)
    (let ((n1 (func-arity f1))
          (n2 (func-arity f2)))
      (or (> n1 n2)
          (and (= n1 n2)
               (more-specific? (func-types f1) (func-types f2) n1))))))
```

# Examples: delta

```
(define a-func
  (func-set [(number)       (n)   (- n 1)]
            [(object char) (o c) (char->integer c)]
            [(object)       (o)   5]))


(delta a-func (list 7 #\A ))    ;; ==>  65

(delta a-func (list 7 'test)) ;; ==>  6

(delta a-func (list 'test 7)) ;; ==>  5
```

# Traversal Control

* control : (`struct number` $\rightarrow$ `boolean`)
* Bypass structure fields
* Dynamic... but not necessarily
* `everywhere`
* (`make-bypass` (type fieldname) ...)

```
;; Define a simple 'pair'
(def-prod a-pair [(n number)
                  (c char)])

;; Define a control/bypass function
(define skip (make-bypass (a-pair n)))

(skip (a-pair 5 #\B) 0) ;;  ==> #f
(skip (a-pair 5 #\B) 1) ;;  ==> #t
```

# Function Sets

* `func-set` builds them
* `union-func` "extends" them
* Others for unions with defaults

# Default Behavior

* Free to build fresh, or...
* `union-_` for each default

idF : `id` transform

```
(lambda (obj) obj)
```

idA : `id` for arguments

```
(lambda (obj targ) targ)
```

Bc : Calls original constructors

```
(func-set
  [(cons object list) (f r) (cons f r)]
  [(empty) (e) e]
   ...)
```

# Finally... traversing

General:

```
(traversal obj F B A C . targ)
```

Defaults with control:

```
(traversal-fc obj F C)          : Bc & idA
(traversal-bc obj B C)          : idF & idA
(traversal-fac obj F A C targ) : Bc
(traversal-bac obj B A C targ) : idF
```

Aliases with `everywhere`:

```
(traversal-f obj F)
(traversal-b obj B)
```

...

# Outline

# Example: BSTs

* Data-Definitions
  - New 'language' for structure definitions
* Simple Transforms
  - Increment
  - Strings
  - Reverse
  - Height

# Example: BST Data-Definitions

```
;; Mixed concrete/abstract syntax
(def-sum  tree [leaf node])
(def-prod leaf ["*"])
(def-prod node ["(" (data number)
                    (left tree)
                    (right tree) ")"])
```

* Sum types (abstract 'interfaces')
* Product types (constructors)
* Uses `define-struct`

# Example: BST Data-Definitions

```
;; Mixed concrete/abstract syntax
(def-sum  tree [leaf node])
(def-prod leaf ["*"])
(def-prod node ["(" (data number)
                    (left tree)
                    (right tree) ")"])
```

* Supports parsing

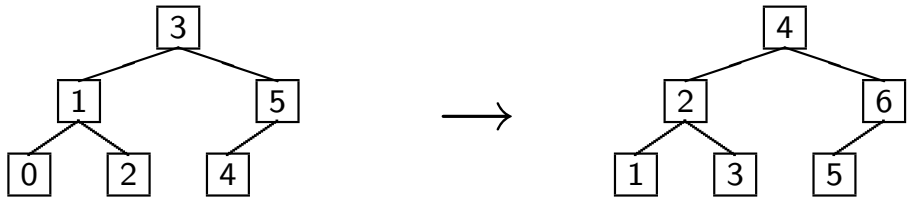
  `"(3 (1 (0 **) (2 **)) (5 (4 **) *))"`

* Introduces constructors

  `node :` $\texttt{number tree tree} \longrightarrow \texttt{tree}$

  `leaf :` $\longrightarrow \texttt{tree}$

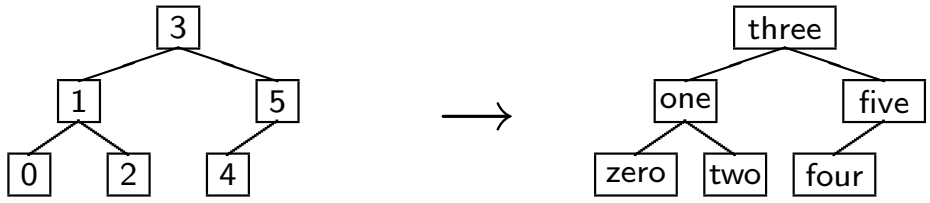
# Example: Increment




```
;; tree-incr : tree -> tree
;; Increment each data element in the given tree
(define (tree-incr t)
  (cond [(leaf? t) t]
        [else (node (add1 (node-data t))
                    (tree-incr (node-left t))
                    (tree-incr (node-right t)))]))


;; AP-F function
(define (incr t)
  (traverse-f t (union-idF ((number) (n) (add1 n)))))
```

# Example: Strings

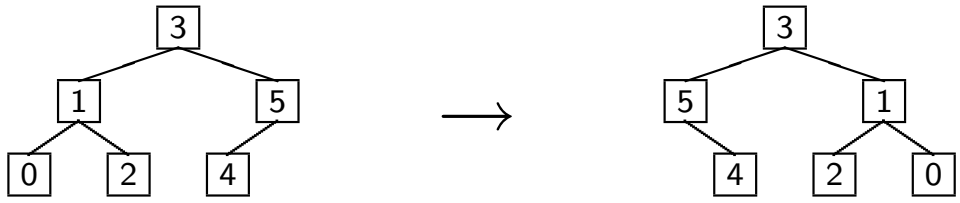



```
(define names '("zero" "one" "two"
                "three" "four" "five"))

;; tree-strs : tree -> tree
;; Replace each data element by its English word
(define (tree-strs t)
  (cond [(leaf? t) t]
        [else (node (list-ref names (node-data t))
                    (tree-strs (node-left t))
                    (tree-strs (node-right t)))]))


;; AP-F function
(define (strs t)
  (traverse-f t (union-idF ((number) (n) (list-ref names n)))))
```

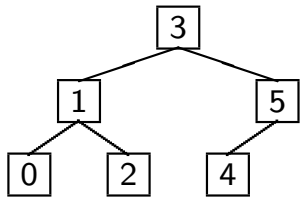
# Example: Reverse




```
;; tree-rev : tree -> tree
;; Reverse all the data elements in the tree
(define (tree-rev t)
  (cond [(leaf? t) t]
        [else (node (node-data t)
                    (tree-rev (node-right t))
                    (tree-rev (node-left t)))]))


;; AP-F function
(define (rev t)
  (traverse-b t (union-Bc [(node object tree tree)
                           (n d lt rt) (node d rt lt)])))
```

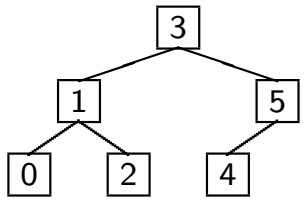
# Example: Height




$\longrightarrow$ 3

```
;; height : tree -> number
;; Calculate the height of the given tree
(define (height t)
  (let ((B (func-set
             [(leaf) (l) 0]
             [(node object number number)
              (n d lt rt) (add1 (max lt rt))])))
    (traverse-b t B)))
```

# Example: Height - Top Down




$\longrightarrow$ 3

```
;; height : tree -> number
;; Calculate the height of the given tree using an argument
(define (height t)
  (let ((A (union-idA
             [(node number) (n h) (add1 h)]))
        (B (func-set
             [(leaf number) (l h) h]
             [(node object number number) (n d lt rt) (max lt rt)]))
    (traverse-ba t B A 0)))
```

# Outline

# More Lambda Examples

* Addition of *let*
  - Missing case is caught

* de Bruijn Indices (different ways)
  - idF (transform *var-e*)
  - Bc (rebuild *var-e*)
  - But scope rules not captured!

# Lambda: adding *let*

```
;; Lambda expressions
(def-sum  exp [var-e lambda-e app-e let-e])

(def-prod var-e [(id symbol)])
(def-prod lambda-e ["(" "lambda" "(" (id symbol) ")"
                                     (body exp) ")"])
(def-prod app-e ["(" (rator exp) (rand exp) ")"])
(def-prod let-e ["(" "let" (id symbol) "=" (rand exp)
                      "in" (body exp) ")"])


;; free-vars: exp -> [setof symbol]
(define (free-vars-apf e)
  (let ((B (func-set
             [(var-e symbol)          (v id)       (set-single id)]
             [(lambda-e symbol set)   (l id fv)    (set-rm fv id)]
             [(app-e set set)         (c fvl fvr) (set-union fvl fvr)])))
    (traverse-b e B)))
```

```
No applicable function found for:
   (let-e symbol set set)
```

# Lambda: adding *let*

```
;; Lambda expressions
(def-sum  exp [var-e lambda-e app-e let-e])

(def-prod var-e [(id symbol)])
(def-prod lambda-e ["(" "lambda" "(" (id symbol) ")"
                                     (body exp) ")"])
(def-prod app-e ["(" (rator exp) (rand exp) ")"])
(def-prod let-e ["(" "let" (id symbol) "=" (rand exp)
                      "in" (body exp) ")"])


;; free-vars: exp -> [setof symbol]
(define (free-vars-apf e)
  (let ((B (func-set
             [(var-e symbol)          (v id)      (set-single id)]
             [(lambda-e symbol set)   (l id fv)   (set-rm fv id)]
             [(app-e set set)         (c fvl fvr) (set-union fvl fvr)]
             [(let-e symbol set set) (l id fvl fvr)
                                      (set-union fvl (set-rm fvr id))])))
    (traverse-b e B)))
```

# Lambda: de Bruijn Indices (hand-written)

```
;; Lambda expressions
(def-sum  exp [ ... addr-e])

;;  ...
(def-prod addr-e ["idx" (n number)])


;; deBruijn: exp -> exp
;; Replace variable exps with their de Bruijn indices (addr)
(define (deBruijn e)
  (letrec ((db* (lambda (e env)
                  (cond [(var-e? e)    (addr-e (lookup env (var-e-id e)))]
                        [(app-e? e)    (app-e (db* (app-e-rator e) env)
                                              (db* (app-e-rand e) env))]
                        [(let-e? e)
                         (let-e (let-e-id e)
                                (db* (let-e-rand e) env)
                                (db* (let-e-body e) (cons (let-e-id e) env)))]
                        [(lambda-e? e)
                         (lambda-e (lambda-e-id e)
                                   (db* (lambda-e-body e)
                                        (cons (lambda-e-id e) env)))])))))
    (db* e '())))
```

# Lambda: de Bruijn Indices

```
;; Lambda expressions
(def-sum  exp [ ... addr-e])

;;  ...
(def-prod addr-e ["idx" (n number)])


;; deBruijn: exp -> exp
;; Replace variable exps with their de Bruijn indices (addr)
(define (deBruijn e)
  (let ((B (union-Bc
             [(var-e symbol list) (v s env) (addr-e (lookup env s))]))
        (A (union-idA
             [(lambda-e list) (l env) (cons (lambda-e-id l) env)])))
    (traverse-ba (let->lambda e) B A '())))


;; let->lambda: exp -> exp
;; Transform 'let' into 'lambda'
(define (let->lambda e)
  (let ((B (union-Bc
             [(let-e symbol exp exp)
              (l id rand body) (app-e (lambda-e id body) rand)])))
    (traverse-b e B)))
```

# Outline

# Traversal Checks

* What are “errors”?
  - No applicable function

* Traversal Results
  - Must match function types
  - Check using structures & control
  - Can provide everything statically

# Example: Errors

```
;; Some Data Structures
(def-sum  W [X Y])
(def-prod X [(n number)])
(def-prod Y [(s symbol)])
(def-prod Z ["z" (w W)])

(define z1 (parse-string Z "z 5"))
(define z2 (parse-string Z "z hello"))

;; number -> string ... not safe for X
(define F (union-idF [(number) (n) (number->string n)]))


(traverse-f z1 F) ;; ==> (Z (X "5"))    -- BAD!

(traverse-f z2 F) ;; ==> (Z (Y 'hello)) -- OK
```

# Example: Errors

```
;; Some Data Structures
(def-sum  W [X Y])
(def-prod X [(n number)])
(def-prod Y [(s symbol)])
(def-prod Z ["z" (w W)])

(define z1 (parse-string Z "z 5"))
(define z2 (parse-string Z "z hello"))

(define B (func-set
            [(X number) (x n) (number->string n)]
            [(Y symbol) (y s) (symbol->string s)]

            ;; Z W -> W
            [(Z W) (z w) w]))


(traverse-b z1 B) ;; ==> error: No (Z string) function

(traverse-b z2 B) ;; ==> error:           ""
```

# Example: Errors

```
;; Some Data Structures
(def-sum  W [X Y])
(def-prod X [(n number)])
(def-prod Y [(s symbol)])
(def-prod Z ["z" (w W)])

(define z1 (parse-string Z "z 5"))
(define z2 (parse-string Z "z hello"))

(define B (func-set
            [(X number) (x n) (number->string n)]
            [(Y symbol) (y s) (symbol->string s)]

            ;; fix: Z string -> string
            [(Z string) (z w) (string-append "(z " w ")")]))


(traverse-b z1 B) ;; ==> "(z 5)"

(traverse-b z2 B) ;; ==> "(z hello)"
```

# Example: Errors

```
;; Some Data Structures
(def-sum  W [X Y V])
(def-prod X [(n number)])
(def-prod Y [(s symbol)])
(def-prod Z ["z" (w W)])

;; New variant of W
(def-prod V [(c char)])


(define z1 (parse-string Z "z '5'"))

(define B (func-set
            [(X number) (x n) (number->string n)]
            [(Y symbol) (y s) (symbol->string s)]
            [(Z string) (z w) (string-append "(z " w ")")]))


(traverse-b z1 B) ;; ==> error: No (V char) function
```

# Example: Errors

```
;; Some Data Structures
(def-sum  W [X Y V])
(def-prod X [(n number)])
(def-prod Y [(s symbol)])
(def-prod Z ["z" (w W)])

;; New variant of W
(def-prod V [(c char)])


(define z1 (parse-string Z "z '5'"))

(define B (func-set
            [(X number) (x n) (number->string n)]
            [(Y symbol) (y s) (symbol->string s)]
            [(Z string) (z w) (string-append "(z " w ")")]

            ;; fix: V char -> string
            [(V char) (v c) (list->string (list c))]))


(traverse-b z1 B) ;; ==> (z 5)
```

# Check Algorithm

* Primitives: transformations
* Products: constructor replacements
* Sums: do subtype traversals unify?
* Recursively simulate structure traversal
    - Traversal is what AP-F is good at!

# AP-F

* Flexibility
  - Implicit traversal
  - Closer to hand written
* Modularity
  - Abstraction/decomposition
  - Building up function sets
* A Solution to the "*Expression Problem*"
  - "Extension" of functionality and data structures

# Outline

# Present/Future Work

* Formal types / safety
* Relation to other AP concepts
* What else can be done statically
* Performance & optimizations
* Composition of function sets/traversals

# Discussion...

Thanks!

Any Questions?