Mixing Mutability into the Nanopass Framework

Andy Keep

Background

- Nanopass framework is a DSL for writing compilers
- Provides a syntax for defining the grammar of an intermediate representation
  - Intermediate representations are immutable*
  - Mutability can be introduced by adding mutable terminals
  - We will look at using this for variables and basic block labels

* technically the lists are just Scheme lists, which are mutable

A simple compiler

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (immediate (imm))
    (symbol (x))
    (primitive (pr)))
  (Expr (e)
    x
    imm
    (quote d)
    (if e0 e1)
    (if e0 e1 e2)
    (and e* ...)
    (or e* ...)
    (not e)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (immediate (imm))
    (symbol (x))
    (primitive (pr)))
  (Expr (e)
    x
    imm
    (quote d)
    (if e0 e1)
    (if e0 e1 e2)
    (and e* ...)
    (or e* ...)
    (not e)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (symbol (x))
    (primitive (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (symbol (x))
    (primitive (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (var (x))
    (primitive (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (var (x))
    (primitive-info (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e* ... e)
    (let ([x* e*] ...) e* ... e)
    (letrec ([x* e*] ...) e* ... e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (var (x))
    (primitive-info (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e)
    (let ([x* e*] ...) e)
    (letrec ([x* e*] ...) e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (var (x))
    (primitive-info (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e)
    (let ([x* e*] ...) e)
    (letrec ([x* e*] ...) e)
    (e e* ...)
    (pr e* ...)))
```

Source language

```
(define-language Lsrc
  (terminals
    (datum (d))
    (var (x))
    (primitive-info (pr)))
  (Expr (e)
    x
    (quote d)
    (if e0 e1 e2)
    (set! x e)
    (begin e* ... e)
    (lambda (x* ...) e)
    (let ([x* e*] ...) e)
    (letrec ([x* e*] ...) e)
    (callable e* ...))
  (Callable (callable)
    e
    pr))
```

Target language?

Target language

- LLVM 10
  - A bit lower level than C
  - Better handling of tail calls
  - Brand new (may require installing llvm and clang 10)
  - Required a bit of SSA conversion

Overall compiler

- parse-scheme
- convert-complex-datum
- uncover-assigned!
- purify-letrec
- convert-assignments
- optimize-direct-call
- remove-anonymous-lambda
- sanitize-binding-forms
- uncover-free
- convert-closures
- optimize-known-call
- introduce-procedure-primitives
- lift-letrec
- normalize-context
- specify-representation
- uncover-locals
- remove-let
- remove-complex-opera*
- flatten-set!
- expose-basic-blocks
- optimize-blocks
- convert-to-ssa
- flatten-functions
- eliminate-simple-moves
- generate-llvm-code

Overall compiler

parse-scheme
convert-complex-datum
uncover-assigned!
purify-letrec
convert-assignments
optimize-direct-call
remove-anonymous-lambda
sanitize-binding-forms
uncover-free
convert-closures
optimize-known-call
introduce-procedure-primitives
lift-letrec
normalize-context
specify-representation
uncover-locals
remove-let
remove-complex-opera*
flatten-set!
expose-basic-blocks
optimize-blocks
convert-to-ssa
flatten-functions
eliminate-simple-moves
generate-llvm-code

Parsing Scheme

- Start with initial environment with syntax and primitives
- Extend environment mapping symbols to a variable record at binding sites
- Replace references to the symbols in the environment with variable records
- Variable records contain a mutable flags field and a mutable "slot"
- References and binding locations share variable record
- No longer need to build environments for variables later
- This is also how Chez Scheme handles variables

Assignment conversion

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

```
(let ([t 5] [y 7])
  (let ([x (cons t (void))])
    (set-car! x (* (car x) 2))
    (+ (car x) y)))
```

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

```
(let ([t 5] [y 7])
  (let ([x (cons t (void))])
    (set-car! x (* (car x) 2))
    (+ (car x) y)))
```

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

```
(let ([t 5] [y 7])
  (let ([x (cons t (void))])
    (set-car! x (* (car x) 2))
    (+ (car x) y)))
```

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

```
(let ([t 5] [y 7])
  (let ([x (cons t (void))])
    (set-car! x (* (car x) 2))
    (+ (car x) y)))
```

Assignment conversion

```
(let ([x 5] [y 7])
  (set! x (* x 2))
  (+ x y))
```

```
(let ([t 5] [y 7])
  (let ([x (cons t (void))])
    (set-car! x (* (car x) 2))
    (+ (car x) y)))
```

Uncover assigned variables

```
(define-pass uncover-assigned! : Ldatum (ir) -> Ldatum ()
  (Expr : Expr (ir) -> Expr ()
    [(set! ,x ,[e]) (var-flags-assigned-set! x #t) ir]))
```

Convert assignments

```
(define-pass convert-assignments : Lletrec (ir) -> Lno-assign ()
  (Lambda : Lambda (ir) -> Lambda ()
    [(lambda (,x* ...) ,e)
     (let-values ([(x* e) (convert-bindings x* e)])
       `(lambda (,x* ...) ,e))])
  (Expr : Expr (ir) -> Expr ()
    [,x (if (var-flags-assigned? x) `(,car-pr ,x) x)]
    [(set! ,x ,[e]) `(,set-car!-pr ,x ,e)]
    [(let ([,x* ,[e*]] ...) ,e)
     (let-values ([(x* e) (convert-bindings x* e)])
       `(let ([,x* ,e*] ...) ,e))]))
```

Convert assignments

```
(define convert-bindings
  (lambda (x* e)
    (with-assigned x*
      (case-lambda
        [(x*) (values x* (Expr e))]
        [(x* assigned-x* new-x*)
         (values x*
           (with-output-language (Lno-assign Expr)
             (let ([pr* (map
                          (lambda (new-x)
                            `(,cons-pr ,new-x (,void-pr)))
                          new-x*)])
               `(let ([,assigned-x* ,pr*] ...)
                  ,(Expr e))))))]))))
```

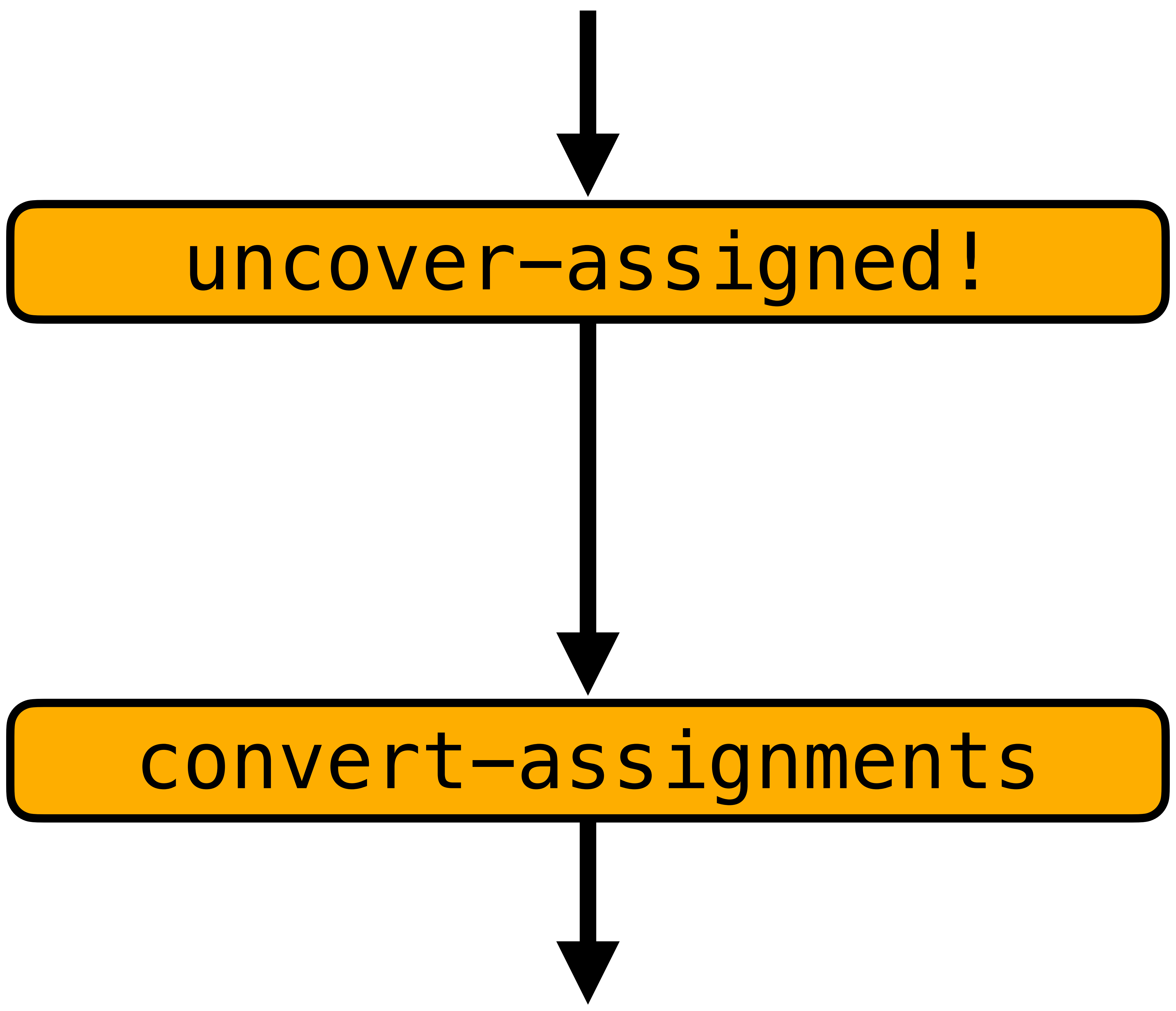
Convert assignments

```
(define with-assigned
  (lambda (x* f)
    (let l ([x* x*] [rx* '()] [rset-x* '()] [rnew-x* '()])
      (if (null? x*)
          (if (null? rset-x*)
              (f (reverse rx*))
              (f (reverse rx*) (reverse rset-x*)
                 (reverse rnew-x*)))
          (let ([x (car x*)] [x* (cdr x*)])
            (if (var-flags-assigned? x)
                (let ([new-x (make-var x)])
                  (l x* (cons new-x rx*)
                    (cons x rset-x*) (cons new-x rnew-x*)))
                (l x* (cons x rx*) rset-x* rnew-x*)))))))
```

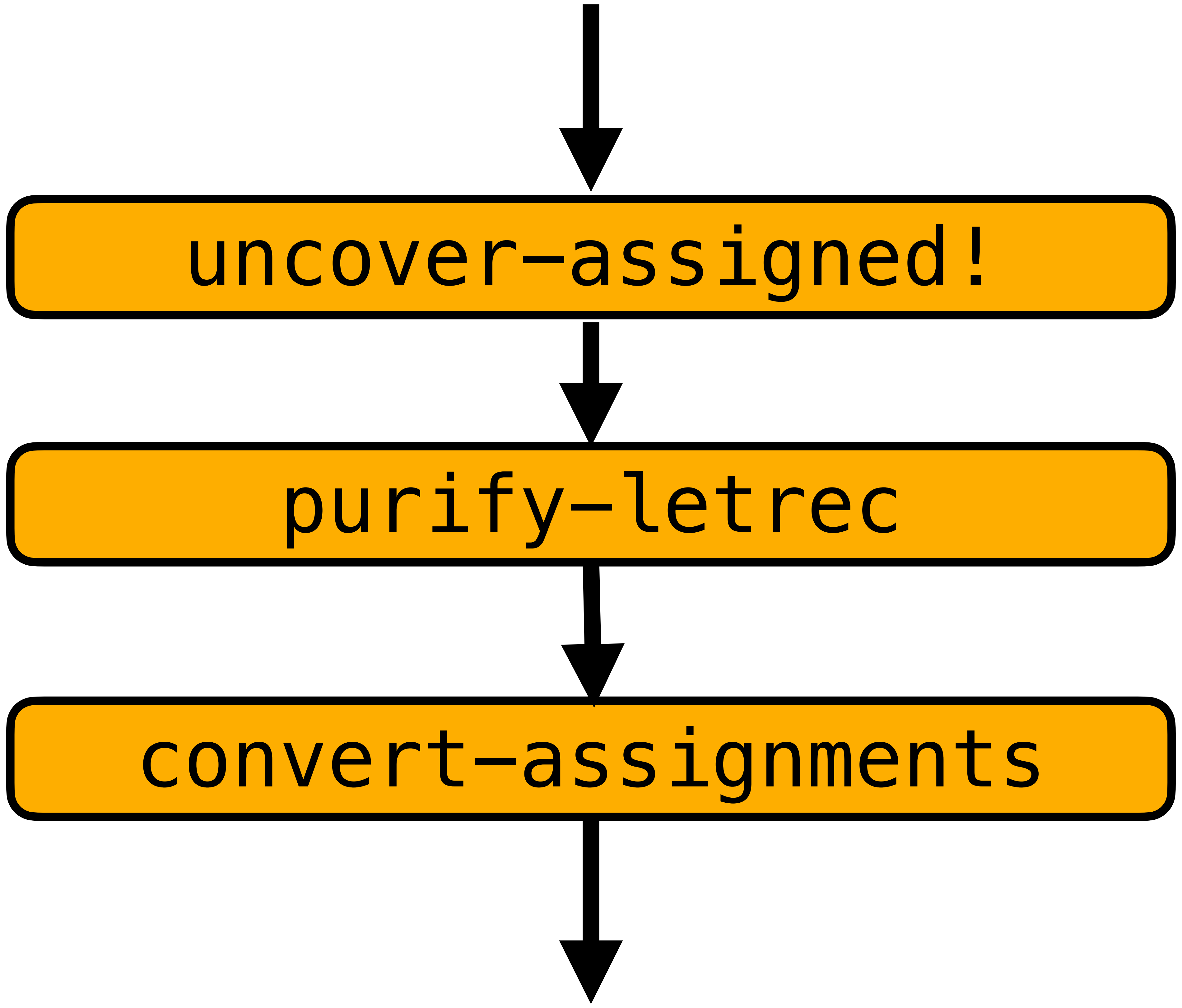
Convert assignments

```
(define-pass convert-assignments : Lletrec (ir) -> Lno-assign ()
  (Lambda : Lambda (ir) -> Lambda ()
    [(lambda (,x* ...) ,e)
     (let-values ([(x* e) (convert-bindings x* e)])
       `(lambda (,x* ...) ,e))])
  (Expr : Expr (ir) -> Expr ()
    [,x (if (var-flags-assigned? x) `(,car-pr ,x) x)]
    [(set! ,x ,[e]) `(,set-car!-pr ,x ,e)]
    [(let ([,x* ,[e*]] ...) ,e)
     (let-values ([(x* e) (convert-bindings x* e)])
       `(let ([,x* ,e*] ...) ,e))]))
```

One small problem



One small problem



Purify letrec

- Categorizes `letrec` bindings into: assigned, simple, lambda, and complex
- Assigned are already marked assigned, no problem there
- Simple and lambda are not assigned, and don't become assigned
- Complex on the other hand, become assigned where they were not before
- We need to track this assignment.

Purify letrec

```
(cond
  [(var-flags-assigned? x)
   (loop (cdr tx*) (cdr e*) xs* es* xl* el*
     (cons x xc*) (cons (Expr e) ec*))]
  [(lambda-expr? e)
   (loop (cdr tx*) (cdr e*) xs* es* (cons x xl*)
     (cons (Expr e) el*) xc* ec*)]
  [(simple-expr? e)
   (loop (cdr tx*) (cdr e*) (cons x xs*)
     (cons (Expr e) es*) xl* el* xc* ec*)]
  [else
   ;; we made an unassigned variable assigned, mark it.
   (var-flags-assigned-set! x #t)
   (loop (cdr tx*) (cdr e*) xs* es* xl* el*
     (cons x xc*) (cons (Expr e) ec*))])
```

Purify letrec

```
(cond
  [(var-flags-assigned? x)
   (loop (cdr tx*) (cdr e*) xs* es* xl* el*
     (cons x xc*) (cons (Expr e) ec*))]
  [(lambda-expr? e)
   (loop (cdr tx*) (cdr e*) xs* es* (cons x xl*)
     (cons (Expr e) el*) xc* ec*)]
  [(simple-expr? e)
   (loop (cdr tx*) (cdr e*) (cons x xs*)
     (cons (Expr e) es*) xl* el* xc* ec*)]
  [else
   ;; we made an unassigned variable assigned, mark it.
   (var-flags-assigned-set! x #t)
   (loop (cdr tx*) (cdr e*) xs* es* xl* el*
     (cons x xc*) (cons (Expr e) ec*))])
```

Closure conversion

Free variable analysis

```
(lambda (x)
  (lambda (y)
    (lambda (z)
      (+ x (+ y z)))))
```

Free variable analysis

```
(lambda (x)
  (lambda (y)    x
    (lambda (z)    x y
      (+ x (+ y z))))))
```

Free variable analysis

```
(lambda (x^0) [ ]
  (lambda (y^1) [x]
    (lambda (z^2) [x y]
      (+ x^0 (+ y^1 z^2)))))
```

Free variable analysis

```
(lambda (x^0)                       000
  (lambda (y^1)          [x]        001
    (lambda (z^2)        [x y]      011
      (+ x^0 (+ y^1 z^2)))))
```

Free variable analysis

```
(lambda (x^0)  [ ]     0 0 0|
  (lambda (y^1)  [x]     0 0|1
    (lambda (z^2)  [x y]   0|1 1
      (+ x^0 (+ y^1 z^2)))))
```

Uncover free

```
(define-pass uncover-free : Lsanitized (ir) -> Lfree ()
  (Callable : Callable (e index fv-info) -> Callable ())
  (Expr : Expr (e index fv-info) -> Expr ()
    [,x (record-ref! x fv-info) x]
    [(let ([,x* ,[e*]] ...) ,e)
     (with-offsets (index x*)
       `(let ([,x* ,e*] ...) ,(Expr e index fv-info)))]
    [(letrec ([,x* ,f*] ...) ,e)
     (with-offsets (index x*)
       (let ([f* (map (lambda (f) (Lambda f index fv-info)) f*)]
             [e (Expr e index fv-info)])
         `(letrec ([,x* ,f*] ...) ,e))))])
  (Lambda : Lambda (e index outer-fv-info) -> Lambda ()
    [(lambda (,x* ...) ,e)
     (let ([fv-info (make-fv-info index)])
       (with-offsets (index x*)
         (let ([e (Expr e index fv-info)])
           (let ([fv* (fv-info-fv* fv-info)])
             (for-each (lambda (fv) (record-ref! fv outer-fv-info)) fv*)
             `(lambda (,x* ...) (free (,fv* ...) ,e))))))])
  (Expr ir 0 (make-fv-info 0)))
```

Uncover free

```
(define-pass uncover-free : Lsanitized (ir) -> Lfree ()
```

```
(define (set-offsets! x* index)
  (fold-left
    (lambda (index x)
      (var-slot-set! x index)
      (fx+ index 1))
    index x*))
(define ($with-offsets index x* p)
  (let ([index (set-offsets! x* index)])
    (let ([v (p index)])
      (for-each (lambda (x) (var-slot-set! x #f)) x*)
      v)))
(define-syntax with-offsets
  (lambda (x)
    (syntax-case x ()
      [(_ (index ?x*) ?e ?es ...)
       (identifier? #'index)
       #'($with-offsets index ?x* (lambda (index) ?e ?es ...))])))
```

```
  (Expr ir 0 (make-fv-info 0)))
```

Uncover free

```
(define-pass uncover-free : Lsanitized (ir) -> Lfree ()
  (Callable : Callable (e index fv-info) -> Callable ())
  (Ex
    [
    [

    [

  (La
    [
        `(lambda (,x* ...) (free (,fv* ...) ,e))))))])
  (Expr ir 0 (make-fv-info 0)))
```

```
(define-record-type fv-info
  (nongenerative)
  (fields lid (mutable mask) (mutable fv*))
  (protocol
    (lambda (new)
      (lambda (index)
        (new index 0 '())))))
(define (record-ref! x info)
  (let ([idx (var-slot x)])
    (when (fx<? idx (fv-info-lid info))
      (let ([mask (fv-info-mask info)])
        (unless (bitwise-bit-set? mask idx)
          (fv-info-mask-set! info (bitwise-copy-bit mask idx 1))
          (fv-info-fv*-set! info (cons x (fv-info-fv* info))))))))
```

Uncover free

```
(define-pass uncover-free : Lsanitized (ir) -> Lfree ()
  (Callable : Callable (e index fv-info) -> Callable ())
  (Expr : Expr (e index fv-info) -> Expr ()
    [,x (record-ref! x fv-info) x]
    [(let ([,x* ,[e*]] ...) ,e)
     (with-offsets (index x*)
       `(let ([,x* ,e*] ...) ,(Expr e index fv-info)))]
    [(letrec ([,x* ,f*] ...) ,e)
     (with-offsets (index x*)
       (let ([f* (map (lambda (f) (Lambda f index fv-info)) f*)]
             [e (Expr e index fv-info)])
         `(letrec ([,x* ,f*] ...) ,e)))])
  (Lambda : Lambda (e index outer-fv-info) -> Lambda ()
    [(lambda (,x* ...) ,e)
     (let ([fv-info (make-fv-info index)])
       (with-offsets (index x*)
         (let ([e (Expr e index fv-info)])
           (let ([fv* (fv-info-fv* fv-info)])
             (for-each (lambda (fv) (record-ref! fv outer-fv-info)) fv*)
             `(lambda (,x* ...) (free (,fv* ...) ,e))))))])
  (Expr ir 0 (make-fv-info 0)))
```

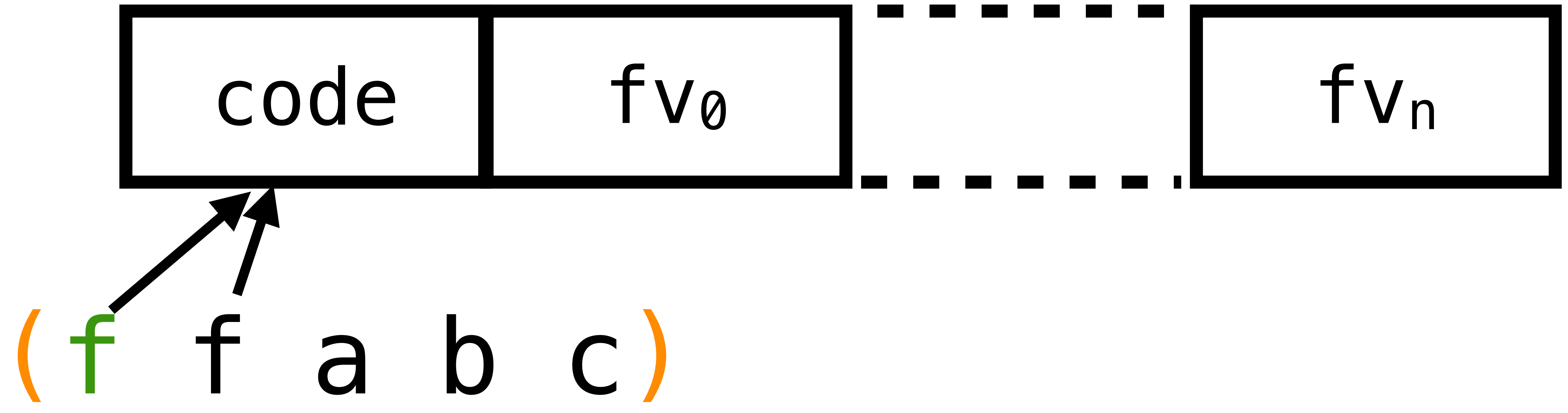
Compiling function calls

```
(f a b c)
```

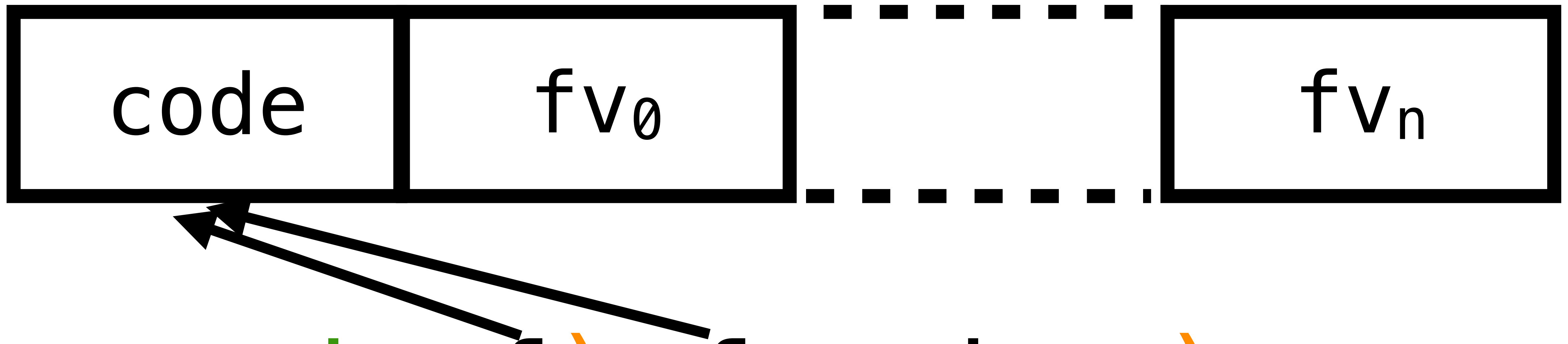
Compiling function calls

```
(f f a b c)
```

Compiling function calls



Compiling function calls




```
(($procedure-code f) f a b c)
```

Compiling function calls

```
(letrec ([lf (lambda (x y z) ---)])
  (closures ([f lf ---])
    ---
    (lf f a b c)
    ---))
```

Optimize known call

```
(define-pass optimize-known-call : Lclosure (ir) -> Lclosure ()
  (Lambda : Lambda (f) -> Lambda ())
  (Expr : Expr (ir) -> Expr ()
    [(,x ,[e*] ...)
     (cond
       [(var-slot x) => (lambda (l) `(,l ,e* ...))]
       [else `(,x ,e* ...)])]
    [(letrec ([,l0* ,f*] ...)
       (closures ([,x* ,l* ,x** ...] ...) ,e))
     (for-each (lambda (x l) (var-slot-set! x l)) x* l*)
     (let ([f* (map Lambda f*)] [e (Expr e)])
       (for-each (lambda (x) (var-slot-set! x #f)) x*)
       `(letrec ([,l0* ,f*] ...)
          (closures ([,x* ,l* ,x** ...] ...) ,e)))]
    ;; NB: should be unnecessary
    [(letrec ([,l* ,f*] ...) ,clbody) (errorf who "unreachable")]))
```

Introduce procedure primitives

```
(letrec ([lf (lambda (cp z)
                (bind-free (cp x y)
                  (+ x (+ y z))))])
  (closures ([f lf x y])
    f))
```

Introduce procedure primitives

```
(letrec ([lf (lambda (cp z)
               (bind-free (cp x y)
                 (+ x (+ y z))))])
  (closures ([f lf x y])
    f))
```

```
(letrec ([lf (lambda (cp z)
               (+ ($procedure-ref cp '0)
                  (+ ($procedure-ref cp '1)
                     z)))])
  (let ([f ($make-closure lf '2)])
    ($procedure-set! f '0 x)
    ($procedure-set! f '1 y)
    f))
```

Introduce procedure primitives

```
(letrec ([lf (lambda (cp z)
               (bind-free (cp x y)
                 (+ x (+ y z))))])
  (closures ([f lf x y])
    f))
```

```
(letrec ([lf (lambda (cp z)
               (+ ($procedure-ref cp '0)
                  (+ ($procedure-ref cp '1)
                     z)))])
  (let ([f ($make-closure lf '2)])
    ($procedure-set! f '0 x)
    ($procedure-set! f '1 y)
    f))
```

Introduce procedure primitives

```
(letrec ([lf (lambda (cp z)
               (bind-free (cp x y)
                 (+ x (+ y z))))])
  (closures ([f lf x y])
    f))
```

```
(letrec ([lf (lambda (cp z)
               (+ ($procedure-ref cp '0)
                  (+ ($procedure-ref cp '1)
                     z)))])
  (let ([f ($make-closure lf '2)])
    ($procedure-set! f '0 x)
    ($procedure-set! f '1 y)
    f))
```

Introduce procedure primitives

```
(letrec ([lf (lambda (cp z)
               (bind-free (cp x y)
                 (+ x (+ y z))))])
  (closures ([f lf x y])
    f))
```

```
(letrec ([lf (lambda (cp z)
               (+ ($procedure-ref cp '0)
                  (+ ($procedure-ref cp '1)
                     z)))])
  (let ([f ($make-closure lf '2)])
    ($procedure-set! f '0 x)
    ($procedure-set! f '1 y)
    f))
```

Introduce procedure primitives

```
(define-pass introduce-procedure-primitives : Lclosure (ir) -> Lproc ()
  (var : var (x) -> Expr ()
    (cond
      [(var-slot x) => build-procedure-ref]
      [else x]))
  (Expr : Expr (e) -> Expr ()
    [,x (var x)]
    [(letrec ([,l0* ,[f*]] ...)
       (closures ([,x* ,l1* ,[e**] ...] ...) ,[e]))
     `(letrec ([,l0* ,f*] ...)
        (let ([,x* ,(build-make-proc! l1* e**)] ...)
          ,(build-procedure-set! x* e** e)))]
    [(,l ,[e*] ...) `(,l ,e* ...)]
    [(,pr ,[e*] ...) `(,pr ,e* ...)]
    [(,[e] ,[e*] ...) `((,procedure-code-pr ,e) ,e* ...)])
  (Lambda : Lambda (f) -> Lambda ()
    [(lambda (,x* ...) (bind-free (,x ,x0* ...) ,e))
     (with-fv* x x0* (lambda () `(lambda (,x* ...) ,(Expr e))))]))
```

Introduce procedure primitives

```
(define (build-procedure-ref pr)
  `(,procedure-ref-pr ,(car pr) (quote ,(cdr pr))))
(define (build-make-proc! l* e**)
  (map
    (lambda (l e*)
      `(,make-procedure-pr ,l (quote ,(length e*))))
    l* e**))
(define (build-procedure-set! x* e** e)
  (let ([ps* (fold-right
               (lambda (x e* ps*)
                 (fold-right
                   (lambda (e i ps*)
                     (cons `(,procedure-set!-pr ,x (quote ,i) ,e) ps*))
                   ps* e* (enumerate e*)))
               '() x* e**)])
    (if (null? ps*)
        e
        `(begin ,ps* ... ,e))))
```

Introduce procedure primitives

```
(define-pass introduce-procedure-primitives : Lclosure (ir) -> Lproc ()
  (var : var (x) -> Expr ()
    (cond
      [(var-slot x
      [else x]))
  (Expr : Expr (e)
    [,x (var x)]
    [(letrec ([,l0
       (closures (
     `(letrec ([,l
        (let ([,x*
          ,(build-
    [(,l ,[e*] ...
    [(,pr ,[e*] ..
    [(,[e] ,[e*] .
  (Lambda : Lambda
    [(lambda (,x* ...) (bind-free (,x ,x0* ...) ,e))
     (with-fv* x x0* (lambda () `(lambda (,x* ...) ,(Expr e))))]))
```

```
(define with-fv*
  (lambda (cp fv* th)
    (let ([ov* (map var-slot fv*)])
      (fold-left
        (lambda (i fv)
          (var-slot-set! fv (cons cp i))
          (fx+ i 1))
        0 fv*)
      (let ([v (th)])
        (for-each var-slot-set! fv* ov*)
        v))))
```

Optimize and reorder blocks

Optimize blocks

```
(labels ([,l* ,t*] ...) ,l)
```

Optimize blocks

```
(labels ([,l* ,t*] ...) ,l)
```

```
(build-graph! l* t*)
```

Optimize blocks

```
(labels ([,l* ,t*] ...) ,l)

(for-each
  (lambda (l t)
    (label-slot-set! l
      (make-graph-node t)))
  l* t*)
```

Optimize blocks

```
(let loop ([wl (list l)] [rl* '()] [rt* '()])
  (if (null? wl)
      (begin
        (for-each (lambda (l) (label-slot-set! l #f)) l*)
        `(labels ([,(reverse rl*) ,(reverse rt*)] ...) ,l))
      (let ([l (car wl)] [wl (cdr wl)])
        (let ([node (label-slot l)])
          (if (graph-node-written? node)
              (loop wl rl* rt*)
              (begin
                (graph-node-written?-set! node #t)
                (let-values ([(t wl)
                              (rewrite-tail
                                (graph-node-tail node) wl)])
                  (loop wl (cons l rl*) (cons t rt*))))))))))
```

Optimize blocks

```
(rewrite-tail : Tail (t wl) -> Tail (wl)
  [(begin ,ef* ... ,t)
   (let*-values ([(ef* wl) (rewrite-effect* ef* wl)]
                 [(t wl) (rewrite-tail t wl)])
     (values `(begin ,ef* ... ,t) wl))]
  [(goto ,l)
   (let ([l (extract-final-target l)])
     (values `(goto ,l) (extend-worklist l wl)))]
  [(return ,l)
   (let ([l (extract-final-target l)])
     (values `(return ,l) (extend-worklist l wl)))]
  [(return ,tr) (values `(return ,tr) wl)]
  [(if (,relop ,tr0 ,tr1) (,l0) (,l1))
   (let ([l0 (extract-final-target l0)]
         [l1 (extract-final-target l1)])
     (values `(if (,relop ,tr0 ,tr1) (,l0) (,l1))
       (extend-worklist l0 l1 wl)))])
```

Other uses of mutation

Mutation in the compiler

- **`convert-complex-datum`** uses **`fluid-let`** for creating constant bindings
- **`lift-letrec`** use **`fluid-let`** for binding top-level labels and functions
- **`uncover-locals`** uses **`fluid-let`** for binding locals list
- **`remove-complex-opera*`** uses **`fluid-let`** for binding locals list
- **`expose-basic-blocks`** uses **`fluid-let`** for binding locals list

Mutation in the compiler

- **`convert-to-ssa`** uses var slot for variable renaming
- **`convert-to-ssa`** uses multiply assigned flag to find variables that need phi
- **`convert-to-ssa`** use label slot for creating control-flow graph
- **`eliminate-simple-moves`** uses var slot for replacing reference with value

Wrapping up

Wrapping up

- Limited and controlled use of mutable storage can be useful
- Mutable information that lasts across passes needs to be maintained
- When using a mutable cell for a single pass, we must cleanup at the end
- We assume only one thread will have a program at a given time
- We can avoid the cost of reconstructing environments using records
- You can try it out yourself:
  https://github.com/akeep/scheme-to-llvm

Thanks!

https://github.com/akeep/scheme-to-llvm

Questions?

https://github.com/akeep/scheme-to-llvm