Running FAE Programs Natively

So far: explained various language constructs by using Racket

Explaining Racket?

- Certain parts explained using simpler parts of Racket
- Leftover parts explained by reduction rules

Today: leftover parts of Racket to even simpler parts

...bottoming out in something like assembly language

Step 1: Simpler Representation of Continuations

Old — build continuations out of procedures

```
; FAE SubCache (FAE-Value -> alpha) -> alpha
(define (interp a-fae sc k)
  ...
  [add (l r) (interp l sc
                     (lambda (v1)
                       (interp r sc
                               (lambda (v2)
                                 (k (num+ v1 v2))))))]
  ...)
```

Step 1: Simpler Representation of Continuations

New — build continuations as records

```
; FAE SubCache FAE-Cont -> FAE-Value
(define (interp a-fae sc k)
  ...
  [add (l r) (interp l sc (addSecondK r sc k))]
  ...)
```

Step 1: Simpler Representation of Continuations

Old — apply continuations as procedures

```
; FAE SubCache (FAE-Value -> alpha) -> alpha
(define (interp a-fae sc k)
  ...
  [num (n) (k (numV n))]
  ...)
```

Step 1: Simpler Representation of Continuations

New — apply continuations as records

```
; FAE SubCache FAE-Cont -> FAE-Value
(define (interp a-fae sc k)
  ...
  [num (n) (continue k (numV n))]
  [add (l r) (interp l sc (addSecondK r sc k))]
  ...)

; continue : FAE-Cont FAE-Value -> FAE-Value
(define (continue k v)
  (type-case FAE-Cont k
    [mtK () v]
    [addSecondK (r sc k)
                (interp r sc (doAddK v k))]
    ...))
```

Step 1: Simpler Representation of Continuations

```
(define-type FAE-Cont
  [mtK]
  [addSecondK (r FAE?)
              (sc SubCache?)
              (k FAE-Cont?)]
  ...)
```

Count every **`lambda`** used to generate a continuation and add a corresponding variant to **`FAE-Cont`**

One field for each free variable in the **`lambda`**

Step 2: Replace Symbols with Numbers

We've done this step before:

- **`compile`** converts a **`FAE`** to a **`CFAE`**

```
(define-type FAE
  ...
  [id (name symbol?)]
  ...)

(define-type CFAE
  ...
  [cid (pos number?)]
  ...)
```

Step 2: Replace Symbols with Numbers

We've done this step before:

- pre-compute substitution positions

```
; compile : FAE CSubCache -> CFae
(define (compile a-fae sc)
  (type-case FAE a-fae
    ...
    [id (name) (cid (locate name sc))]
    [fun (param body-expr)
         (cfun (compile body-expr (aCSub param sc)))]
    ...))
```

Step 2: Replace Symbols with Numbers

We've done this step before:

- use simple list for substitutions at run-time

```
; interp : FAE SubCache FAE-Cont -> FAE-Value
(define (interp a-fae sc k)
  ...
  [cid (pos) (continue k (list-ref sc pos))]
  ...)

; continue : FAE-Cont FAE-Value -> FAE-Value
(define (continue k v)
  [doAppK (fun-val k)
          (interp (closureV-body fun-val)
                  (cons v
                        (closureV-sc fun-val))
                  k)])
```

Step 3: Replace Function Calls with Gotos

Aside from building records and using primitives like **+**, all function calls are in ***tail position***

```
(define (interp a-fae sc k)
  (type-case CFAE a-fae
    [cnum (n) (continue ...)]
    [cadd (l r) (interp ...)]
    [csub (l r) (interp ...)]
    [cid (pos) (continue ...)]
    [cfun (body-expr) (continue ...)]
    [capp (fun-expr arg-expr) (interp ...)]
    [cif0 (test-expr then-expr else-expr) (interp ...)]))

(define (continue k v)
  (type-case CFAE-Cont k
    [mtK () v]
    [addSecondK (r sc k) (interp ...)]
    [doAddK (v1 k) (continue ...)]
    [subSecondK (r sc k) (interp ...)]
    [doSubK (v1 k) (continue ...)]
    [appArgK (arg-expr sc k) (interp ...)]
    [doAppK (fun-val k) (interp ...)]
    [doIfK (then-expr else-expr sc k) (if (numzero? v)
                                          (interp ...)
                                          (interp ...))]))
```

Step 3: Replace Function Calls with Gotos

Aside from building records and using primitives like **+**, all function calls are in ***tail position***

Change each to **`set!`** plus a 0-argument call

- Old:

```
(define (interp a-fae sc k)
  (type-case CFAE a-fae
    ...
    [cadd (l r)
          (interp l sc
                  (addSecondK
                   r sc k))]
    ...))
```

- New:

```
(define fae-reg (cnum 0))
(define sc-reg empty)

; interp : -> void
(define (interp)
  (type-case CFAE fae-reg
    ...
    [cadd (l r)
          (begin
            (set! fae-reg l)
            (set! k-reg
                  (addSecondK
                   r sc-reg k-reg))
            (interp))]
    ...))
```

Step 4: Replace Datatypes with Kons

Eliminate **`define-datatype`** and **`type-case`** by using a single datatype and **`case`**

```
(define (any? x) true)

(define-type Pair
  [kons (first any?)
        (rest any?)])

(define fst kons-first)
(define rst kons-rest)
```

Step 4: Replace Datatypes with Kons

Eliminate **`define-datatype`** and **`type-case`** by using a single datatype and **`case`**

- Old:
```
(type-case CFAE fae-reg
  ...
  [cadd (l r)
        ...
        (set! k-reg (addSecondK r sc-reg k-reg))
        ...])
```

- New:
```
(case (fst fae-reg)
  ...
  [(9)
   ...
   (set! k-reg (kons 1
                     (kons (rst (rst fae-reg))
                           (kons sc-reg k-reg))))
   ...])
```

Step 4: Replace Datatypes with Kons

Eliminate `define-datatype` and `type-case` by using a single datatype and `case`

`mtK`	$\Rightarrow$	`0`
`addSecondK`	$\Rightarrow$	`1`
`doAddK`	$\Rightarrow$	`2`
`...`		
`cnum`	$\Rightarrow$	`8`
`cadd`	$\Rightarrow$	`9`
`...`		
`numV`	$\Rightarrow$	`15`
`closureV`	$\Rightarrow$	`16`

Step 4: Replace Datatypes with Kons

Eliminate **`define-datatype`** and **`type-case`** by using a single datatype and **`case`**

Use **`kons`** for substitutions, too

```
(define (interp)
  (case (fst fae-reg)
    ...
    [(11) (begin ; id
             (set! sc2-reg sc-reg)
             (set! v-reg (rst fae-reg))
             (sc-ref))]
    ...))

(define sc2-reg 0)
(define (sc-ref)
  (if (zero? v-reg)
      (begin (set! v-reg (fst sc2-reg))
             (continue))
      (begin (set! sc2-reg (rst sc2-reg))
             (set! v-reg (- v-reg 1))
             (sc-ref))))
```

Step 5: Replace Pair Datatype with Malloc

Simulate **`malloc`** using a vector:

```
(define memory (make-vector 2048))
(define ptr 0)

; kons : number number -> number
(define (kons a b)
  (begin
    (vector-set! memory ptr a)
    (vector-set! memory (+ ptr 1) b)
    (set! ptr (+ ptr 2))
    (- ptr 2)))

; fst : number -> number
(define (fst n)
  (vector-ref memory n))

; rst : number -> number
(define (rst n)
  (vector-ref memory (+ n 1)))
```

Step 6: Deallocation

Flatten pairs to arrays that start with the tag:

```
(kons 9 (kons (compile l ds) (compile r ds)))
```

$\Rightarrow$

```
(malloc2 9 (compile l ds) (compile r ds))
```

Last step: add a garbage collector...