Continuations and Functions

We've made continuations explicit:

```
; 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))))))]
  ...)
```

... but we're back to relying on Racket functions

A Function as Data

What information is in the function?

```
(lambda (v1)
  (interp r sc
          (lambda (v2)
            (k (num+ v1 v2)))))
```

The function captures **`r`**, **`sc`**, and **`k`**, and the rest is constant

A Function as Data

What information is in the function?

```
(lambda (v1)
  (interp r sc
          (lambda (v2)
            (k (num+ v1 v2)))))
```

The same information:

```
(addSecondK r sc k)
```

with

```
(define (continue k v)
  (type-case FAE-Cont k
    ....
    [addSecondK (r sc k)
                (interp r sc (lambda (v2)
                               (k (num+ v v2))))]))
```

A Function as Data

What information is in the function?

```
(lambda (v1)
  (interp r sc
          (lambda (v2)
            (k (num+ v1 v2)))))
```

The same information:

```
(addSecondK r sc k)
```

with

```
(define (continue k v)
  (type-case FAE-Cont k
    ....
    [addSecondK (r sc k)
                (interp r sc (doAddK k v))]
    [doAddK (v1 k) (k (num+ v1 v))]))
```

One Record Per Continuation Lambda

```
(define (interp a-fae ds k)
  (type-case FAE a-fae
    [num (n) (k (numV n))]
    [add (l r) (interp l ds
                       (lambda (v1)
                         (interp r ds
                                 (lambda (v2)
                                   (k (num+ v1 v2))))))]
    [sub (l r) (interp l ds
                       (lambda (v1)
                         (interp r ds
                                 (lambda (v2)
                                   (k (num- v1 v2))))))]
    [id (name) (k (lookup name ds))]
    [fun (param body-expr)
         (k (closureV param body-expr ds))]
    [app (fun-expr arg-expr)
         (interp fun-expr ds
                 (lambda (fun-val)
                   (interp arg-expr ds
                           (lambda (arg-val)
                             (interp (closureV-body fun-val)
                                     (aSub (closureV-param fun-val)
                                           arg-val
                                           (closureV-ds fun-val))
                                     k)))))]
    [if0 (test-expr then-expr else-expr)
         (interp test-expr ds
                 (lambda (v)
                   (if (numzero? v)
                       (interp then-expr ds k)
                       (interp else-expr ds k))))]))
```

```
(define-type FAE-Cont
  [mtK]
  [addSecondK (r FAE?)
              (ds DefrdSub?)
              (k FAE-Cont?)]
  [doAddK (v1 FAE-Value?)
          (k FAE-Cont?)]
  [subSecondK (r FAE?)
              (ds DefrdSub?)
              (k FAE-Cont?)]
  [doSubK (v1 FAE-Value?)
          (k FAE-Cont?)]
  [appArgK (arg-expr FAE?)
           (ds DefrdSub?)
           (k FAE-Cont?)]
  [doAppK (fun-val FAE-Value?)
          (k FAE-Cont?)]
  [doIfK (then-expr FAE?)
         (else-expr FAE?)
         (ds DefrdSub?)
         (k FAE-Cont?)])
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(interp (add (num 1) (num 2))
        (mtSub)
        (mtK))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))
```

```
(interp (add (num 1) (num 2))
        (mtSub)
        (mtK))
→ (interp (num 1)
          (mtSub)
          (addSecondK (num 2) (mtSub) (mtK)))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(interp (num 1)
        (mtSub)
        (addSecondK (num 2) (mtSub) (mtK)))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))
```

```
(interp (num 1)
        (mtSub)
        (addSecondK (num 2) (mtSub) (mtK)))
→ (continue (addSecondK (num 2) (mtSub) (mtK))
            (numV 1))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(continue (addSecondK (num 2) (mtSub) (mtK))
          (numV 1))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))
```

```
(continue (addSecondK (num 2) (mtSub) (mtK))
          (numV 1))
→ (interp (num 2)
          (mtSub)
          (doAddK (numV 1) (mtK)))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(interp (num 2)
        (mtSub)
        (doAddK (numV 1) (mtK)))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))
```

```
(interp (num 2)
        (mtSub)
        (doAddK (numV 1) (mtK)))
→ (continue (doAddK (numV 1) (mtK))
            (numV 2))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(continue (doAddK (numV 1) (mtK))
          (numV 2))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))
```

```
(continue (doAddK (numV 1) (mtK))
          (numV 2))
→ (continue (mtK)
            (numV 3))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(continue (mtK)
          (numV 3))
```

Interp and Continue

```
; interp : 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))]
    [doAddK (v1 k) (continue k (num+ v1 v))]
    ...))

(continue (mtK)
          (numV 3))
→ (numV 3)
```

Running FAE Programs Natively

High-level features of Racket that we're still using:

- Symbols
- Functions and function calls
- Variants and **`type-case`**
- Memory management

Next: convert to even simpler parts, to arrive at something like assembly language

Version 1 is **`fae-k.rkt`**...

Version 2: Replace Symbols with Numbers

(we've done this step before)

A **compile** converts a **FAE** to a **CFAE**:

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

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

Version 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)))]
    ...))
```

Version 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)])
```

Version 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 ...))]))
```

Version 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))]
    ...))
```

Version 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)
```

Version 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))))
   ...])
```

Version 4: Replace Datatypes with Kons

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

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

Version 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))))
```

Version 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)))
```

Version 6: Prepare for Deallocation

Our interpreter is complete, but it runs out of space easily...

First, flatten pairs to arrays that start with the tag:

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

To do: garbage collection