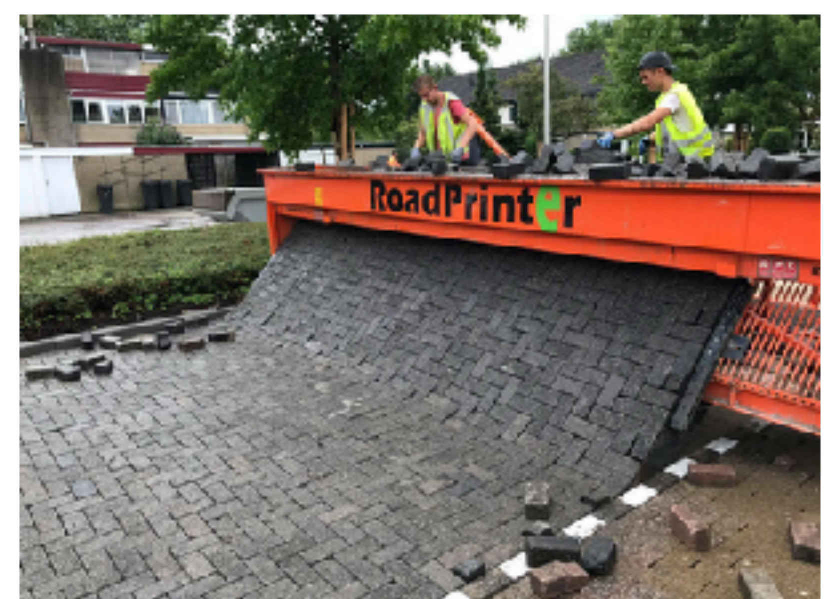
Abstract machines and A-Normal Form

CIS 700—Kristopher Micinski (some slides Thomas Gilray, UAB)

Program Analysis: Foundations and Applications

Fall ’19, Syracuse University



Last time…

$$\langle C, E, K \rangle$$

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
```

k ::= **halt** | **ar**(e, env, k) | **fn**(v, k)

$$((\texttt{e}_0\ \ \texttt{e}_1), env, k) \rightarrow (\texttt{e}_0, env, \textbf{ar}(\texttt{e}_1, env, k))$$

$$(\texttt{x}, env, \textbf{ar}(\texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1, \textbf{fn}(env(\texttt{x}), k_1))$$

$$((\texttt{λ}\ \ (\texttt{x})\ \ \texttt{e}), env, \textbf{ar}(\texttt{e}_1, env_1, k_1)) \rightarrow (\texttt{e}_1, env_1, \textbf{fn}(((\texttt{λ}\ \ (\texttt{x})\ \ \texttt{e}), env), k_1))$$

$$(\texttt{x}, env, \textbf{fn}(((\texttt{λ}\ (\texttt{x}_1)\ \texttt{e}_1), env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[\texttt{x}_1 \mapsto env(\texttt{x})], k_1)$$

$$((\texttt{λ}\ (\texttt{x})\ \texttt{e}), env, \textbf{fn}(((\texttt{λ}\ (\texttt{x}_1)\ \texttt{e}_1), env_1), k_1)) \rightarrow (\texttt{e}_1, env_1[\texttt{x}_1 \mapsto ((\texttt{λ}\ (\texttt{x})\ \texttt{e}), env)], k_1)$$

Warmup: adding literals

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | c

c ::= n
    | #t | #f
```

k ::= **halt** | **ar**(e, env, k) | **fn**(v, k)

$$((e_0\ e_1), env, k) \rightarrow (e_0, env, \mathbf{ar}(e_1, env, k))$$

$$(x, env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(env(x), k_1))$$

$$((\lambda\ (x)\ e), env, \mathbf{ar}(e_1, env_1, k_1)) \rightarrow (e_1, env_1, \mathbf{fn}(((\lambda\ (x)\ e), env), k_1))$$

Need to have another rule here to handle the literal case…

$$(x, env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1))$$
$$\rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

$$((\mathtt{e_0}\ \ \mathtt{e_1}),\ \text{env},\ k) \ \rightarrow\ (\mathtt{e_0},\ \text{env},\ \textbf{ar}(\mathtt{e_1},\ \text{env},\ k))$$

$$(\mathtt{x},\ \text{env},\ \textbf{ar}(\mathtt{e_1},\ \text{env}_1,\ k_1)) \ \rightarrow\ (\mathtt{e_1},\ \text{env}_1,\ \textbf{fn}(\text{env}(\mathtt{x}),\ k_1))$$

$$((\lambda\ \ (\mathtt{x})\ \ \mathtt{e}),\ \text{env},\ \textbf{ar}(\mathtt{e_1},\ \text{env}_1,\ k_1)) \ \rightarrow\ (\mathtt{e_1},\ \text{env}_1,\ \textbf{fn}(((\lambda\ \ (\mathtt{x})\ \ \mathtt{e}),\ \text{env}),\ k_1))$$

When we see a constant as the **return value**…

$$(\mathtt{c},\ \text{env},\ \textbf{fn}(((\lambda\ (\mathtt{x_1})\ \mathtt{e_1}),\ \text{env}_1),\ k_1)) \ \rightarrow\ (\mathtt{e_1},\ \text{env}_1[x_1 \mapsto c],\ k_1)$$

$$(\mathtt{x},\ \text{env},\ \textbf{fn}(((\lambda\ (\mathtt{x_1})\ \mathtt{e_1}),\ \text{env}_1),\ k_1)) \ \rightarrow\ (\mathtt{e_1},\ \text{env}_1[x_1 \mapsto \text{env}(\mathtt{x})],\ k_1)$$

$$((\lambda\ (\mathtt{x})\ \mathtt{e}),\ \text{env},\ \textbf{fn}(((\lambda\ (\mathtt{x_1})\ \mathtt{e_1}),\ \text{env}_1),\ k_1))$$
$$\rightarrow\ (\mathtt{e_1},\ \text{env}_1[x_1 \mapsto ((\lambda\ (\mathtt{x})\ \mathtt{e}),\ \text{env})],\ k_1)$$

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

How do we evaluate `(prim` e_0 e_1`)`?

- Lookup prim, assume we have an implementation in Racket
- Next, evaluate e_0 to v_0
  - Intuitively this requires a context of `(prim` $\bullet$ e_1`)`
- Then, evaluate e_1 to v_1, keep context `(prim` v_0 $\bullet$`)`
- Last, perform `(prim v0 v1)` in Racket, then return

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

To handle prim, we need two more continuations

(prim • e_1)

(prim v_0 •)

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

To handle prim, we need two more continuations

prim-rhs(prim, e_1, env, k)

apply-prim(prim, n, k)

```
(prim • e1)
(prim v0 •)
```

Careful: remember env and k!

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

To handle prim, we need two more continuations

prim-rhs(prim, e_1, env, k)

apply-prim(prim, v, k)

```
(prim • e1)
(prim v0 •)
```

```
(define (kont? k)
  (match k
    …
    [`(prim-rhs ,prim ,env ,(? expr?) ,k) #t]
    [`(apply-prim ,prim ,env ,(? value?) ,k) #t]))
```

call/cc semantics

$$((\texttt{call/cc}\ (\lambda\ (x)\ e_0)), \mathrm{env}, k) \rightarrow (e_0, \mathrm{env}[x \mapsto k], k)$$

$$((\lambda\ (x)\ e_0), \mathrm{env}, \mathbf{fn}(k_0, k_1)) \rightarrow ((\lambda\ (x)\ e_0), \mathrm{env}, k_0)$$

$$(x, \mathrm{env}, \mathbf{fn}(k_0, k_1)) \rightarrow (x, \mathrm{env}, k_0)$$

Administrative normal form (ANF)

- Partitions the grammar into complex expressions (e) and atomic expressions (ae)—variables, datums, etc.

- Expressions cannot contain sub-expressions, except possibly in tail position, and therefore must be `let`-bound.

- ANF-conversion syntactically enforces an evaluation order as an explicit stack of let forms binding each expression in turn.

- **Replaces a multitude of different continuations with fewer continuations (usually a `let` continuation).**

```
((f g) (+ a 1) (* b b))
```

ANF conversion

```
(let ([t0 (f g)])
  (let ([t1 (+ a 1)])
    (let ([t2 (* b b)])
      (t0 t1 t2))))
```

```
x = a+1;
y = b*2;
y = (3*x) + (y*y);
```

```
(let ([x (+ a 1)])
  (let ([y (* b 2)])
    (let ([y (+ (* 3 x) (* y y))])
      ...)))
```

ANF conversion & alpha-renaming

```
(let ([x0 (+ a0 1)])
  (let ([y0 (* b0 2)])
    (let ([t0 (* 3 x0)])
      (let ([t1 (* y0 y0)])
        (let ([y1 (+ t0 t1)])
          ...)))))
```

ANF Conversion...

```
(define (anf-convert e)
  (define (normalize-ae e k)
    ...)
  (define (normalize-aes es k)
    ...)
  (define (normalize e k)
    (match e
      [(? number? n) (k n)]
      [(? symbol? x) (k x)]
      [`(lambda (,x) ,e0) (k `(lambda (,x) ,(anf-convert e0)))]
      [`(if ,e0 ,e1 ,e2)
       (normalize-ae e0
                     (lambda (ae)
                       (k `(if ,ae
                               ,(anf-convert e1)
                               ,(anf-convert e2)))))]
      [`(,es ...)
       (normalize-aes es k)]))
  (normalize e (lambda (x) x)))
```

```
(define (normalize-ae e k)
    (normalize e (lambda (anf)
                   (match anf
                     [(? number? n) (k n)]
                     [(? symbol? x)
                      (k x)]
                     [`(lambda ,xs ,e0)
                      (k `(lambda ,xs ,e0))]
                     [else
                      (define ax (gensym 'a))
                      `(let ([,ax ,anf])
                         ,(k ax))]))))

(define (normalize-aes es k)
  (if (null? es)
      (k '())
      (normalize-ae (car es) (lambda (ae)
                         (normalize-aes (cdr es)
                                        (lambda (aes)
                                          (k
`(,ae ,@aes))))))))
```

Next: adding let

$$e ::= \ldots \mid (\texttt{let}\ ([x\ e_0])\ e_1)$$

$$k ::= \ldots \mid \textbf{let}(x, e, env, k)$$

$$(x, env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, \textbf{let}(x_1, e_1, env_1, k_1)) \rightarrow (e_1, env_1[x_1 \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

What about other language features?

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (call/cc (λ (x) e))
    | (prim e e)
    | (if e e e)
```

What about `if`?

(if • e_t e_f)

Only need **one** additional continuation frame

Once we know guard, go to e_t / e_f as appropriate **immediately**

Also need to add the rules

set!

set! allows mutation

```
(define x 23)
(set! x (add1 x))
;; x now 24
```

In Racket, any variable can be set!'d

```
((lambda (y)
    (displayln y)
    ((lambda (x) (set! x 4) (displayln x)) y)
    (displayln y))
 23)
```

```
;; Prints
23
4
23
```

How can we model set!…?

At an ultra-high level, we need a store (heap)

A few choices:

- Encode by translating it into another feature
- Definitional interpreter: reuse store from Racket
- Implement abstract machine w/ **heap**

CESK Machine

Extend CEK machine to include **store**

Environment
var? -> addr?

$$\langle C, E, S, K \rangle$$

Control
(expr?)

Store:
addr? -> value?

Continuation

CESK Machine

Extend CEK machine to include **store**

Notice that variables map to **addresses**, which map to **values** via the store

Environment
var? -> addr?

$$\langle C, E, S, K \rangle$$

Control
(expr?)

Store:
addr? -> value?

Continuation

$$\langle C, E, S, K \rangle$$

For lambda calculus…

C — Expression

E — Environment: `Var -> Addr`

Note the change! From `Var->Var` to `Var-> Addr`!

S — Store: `Addr -> Val`

K — Continuation

This can **either** stay the same **or** we can store-allocate continuations (i.e., hold a pointer to them.)

S — Store: `Addr -> Val + Kont`

Addr can be any set you want, e.g., the naturals…

Later we will tune addr to tune analysis precision

New parts in red…

$$((\mathtt{e}_0\ \mathtt{e}_1), \text{env}, \text{sto}, \text{k})$$
$$\rightarrow (\mathtt{e}_0, \text{env}, \text{sto}, \textbf{ar}(\mathtt{e}_1, \text{env}, \text{k}))$$

New parts in red…

$$((\mathtt{e}_0\ \mathtt{e}_1), \text{env}, \text{sto}, k)$$
$$\to (\mathtt{e}_0, \text{env}, \text{sto}, \textbf{ar}(\mathtt{e}_1, \text{env}, k))$$

Lookup **through store**

$$(x, \text{env}, \text{sto}, \textbf{ar}(\mathtt{e}_1, \text{env}_1, k_1))$$
$$\to (\mathtt{e}_1, \text{env}_1, \text{sto}, \textbf{fn}(\text{sto}(\text{env}(x)), k_1))$$

New parts in red…

$((e_0\ \ e_1))$, env, sto, k)
→ (e_0, env, sto, **ar**(e_1, env, k))

Lookup **through store**

(x, env, sto, **ar**(e_1, env_1, k_1))
→ (e_1, env_1, sto, **fn**(sto(env(x)), k_1))

((λ (x) e), env, sto, **ar**(e_1, env_1, k_1))
→ (e_1, env_1, sto, **fn**(((λ (x) e), env), k_1))

New parts in red…

$$((\mathtt{e}_0\ \mathtt{e}_1), \text{env}, \text{sto}, k) \to (\mathtt{e}_0, \text{env}, \text{sto}, \mathbf{ar}(\mathtt{e}_1, \text{env}, k))$$

Lookup **through store**

$$(x, \text{env}, \text{sto}, \mathbf{ar}(\mathtt{e}_1, \text{env}_1, k_1)) \to (\mathtt{e}_1, \text{env}_1, \text{sto}, \mathbf{fn}(\text{sto}(\text{env}(x)), k_1))$$

$$((\lambda\ (x)\ \mathtt{e}), \text{env}, \text{sto}, \mathbf{ar}(\mathtt{e}_1, \text{env}_1, k_1)) \to (\mathtt{e}_1, \text{env}_1, \text{sto}, \mathbf{fn}(((\lambda\ (x)\ \mathtt{e}), \text{env}), k_1))$$

$$(x, \text{env}, \text{sto}, \mathbf{fn}(((\lambda\ (x_1)\ \mathtt{e}_1), \text{env}_1), k_1)) \to (\mathtt{e}_1, \text{sto}, \text{env}_1[x_1 \mapsto \text{env}(x)], k_1)$$

New parts in red…

$$((e_0\ \ e_1), env, sto, k) \to (e_0, env, sto, \mathbf{ar}(e_1, env, k))$$

Lookup **through store**

$$(x, env, sto, \mathbf{ar}(e_1, env_1, k_1)) \to (e_1, env_1, sto, \mathbf{fn}(sto(env(x)), k_1))$$

$$((\lambda\ (x)\ e), env, sto, \mathbf{ar}(e_1, env_1, k_1)) \to (e_1, env_1, sto, \mathbf{fn}(((\lambda\ (x)\ e), env), k_1))$$

$$(x, env, sto, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \to (e_1, sto, env_1[x_1 \mapsto env(x)], k_1)$$

$$((\lambda\ (x)\ e), env, sto, \mathbf{fn}(((\lambda\ (x_1)\ e_1), env_1), k_1)) \to (e_1, env_1[x_1 \mapsto a_{new}], sto[a_{new} \mapsto ((\lambda\ (x)\ e), env)], k_1)$$

$$\mathbf{a_{new} \notin dom(sto)}$$

Allocate **new** address, then update env so that x is this new **pointer**, update store to point at closure

Predictable changes to accommodate if, call/cc, etc…

But mostly all pretty easy transformations from CEK

What about `(set! x e)`

```
e ::= (λ (x) e)
    | (e e)
    | x
    | (set! x e)
```

k ::= **halt** | **ar**(e, env, k) | **fn**(v, k) | **set**(addr,k)

(`(set! x e)`, env, sto, k)
→ (e_1, env_1, sto, **set**(env(x),k))

(v, env, sto, **set**(a,k))
→ (e_1, env_1, sto[a↦v], k)