A Recursive Definition of Factorial

(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))

Sample Factorial Execution

(fact 4)
-> (* 4 (fact 3))
-> (* 4 (* 3 (fact 2)))
-> (* 4 (* 3 (* 2 (fact 1))))
-> (* 4 (* 3 (* 2 (* 1 (fact 0))))))
-> (* 4 (* 3 (* 2 (* 1 1))))
-> ...
-> 24

Context and Reduction Expression

- Each call of fact is made with
  - a promise that the value returned will be multiplied by the value of n at the time of the call.
- fact is invoked in larger and larger context as the calculation proceeds.

Recursive control behavior

- With each recursive call
  - additional control information must be recorded,
  - information must be retained until the call returns.

Tail-Recursive Factorial

(define fact2
  (lambda (n)
    (fact-iter-acc n 1)))
(define fact-iter-acc
  (lambda (n a)
    (if (zero? n)
        a
        (fact-iter-acc (- n 1) (* n a)))))
Iterative Factorial Execution

(fact2 4)
\(\rightarrow\) (fact-iter-acc 4 1)
\(\rightarrow\) (fact-iter-acc 3 4)
\(\rightarrow\) (fact-iter-acc 2 12)
\(\rightarrow\) (fact-iter-acc 1 24)
\(\rightarrow\) (fact-iter-acc 0 24)
\(\rightarrow\) 24

Iterative Control Behavior

fact-iter-acc

- Always invoked in the same context.
- Calls itself at the tail-end of a call.
- No promise is made to do anything with the returned value.
- Records no control information.

Iterative Control Behavior Definition

- A process that uses a bounded amount of memory for control information exhibits iterative control behavior.

Continuation Passing Style Transformation

- Transforms recursive control behavior into iterative control behavior.
- Transformation from fact to fact2
  - Introduces another procedure, fact-iter-acc
  - Which takes an extra accumulator \(a\) into which the answer is accumulated.
- Useful, but not sufficiently general.

Example Derivation

\((\ast 4 (\ast 3 (\ast 2 (\text{fact 1}))))\)

(fact 1) is being computed in the context:
\((\ast 4 (\ast 3 (\ast 2 (\text{fact 1}))))\)

- Represent the context with a one argument lambda expression.

Continuation

- Lambda expression representing the context of the call to (fact 1)
  \((\lambda v \ (\ast 4 (\ast 3 (\ast 2 v))))\)
- Name this lambda expression \(k\)
  \((k \ (\text{fact 1})) \rightarrow \ (\ast 4 (\ast 3 (\ast 2 (\text{fact 1}))))\)
Continuation Passing Style Factorial

(define fact-cps
  (lambda (n k)
    (if (zero? n)
        (k 1)
        (fact-cps (– n 1)
          (lambda (v) (k (* n v)))))))

CPS Factorial computes factorial

Proposition:
  (k (fact m)) == (fact-cps m k)

Proof
(by induction on n)

n=0 (k (fact 0)) -> (fact-cps 0 k) -> (k 1)
Assume true for m = n – 1, want to show it holds for m = n

Induction step

(k (fact n))
  -> (k (* n (fact (– n 1)))) definition of fact
  -> ((lambda (v) (k (* n v))) reverse of application (fact (– n 1)))
  -> (fact-cps (– n 1)
      (lambda (v) (k (* n v))))) induction hypothesis
  -> (fact-cps n k) definition of fact-cps.

Execution of fact-cps

(fact-cps 4 k)
  -> (fact-cps 3 (lambda (v) (k (* 4 v))))
  -> (fact-cps 2 (lambda (v)
        ((lambda (v)
          (k (* 4 (* 3 v)))) (* 3 v))))
  -> (fact-cps 2 (lambda (v) (k (* 4 (* 3 v)))))
  -> (fact-cps 1 (lambda (v)
        ((lambda (v)
          (k (* 4 (* 3 v)))) (* 2 v)))))

More abstract version of CPS Factorial Execution

(fact-cps 4 k)
  -> (fact-cps 3 (lambda (v) (k (* 4 v))))
  -> (fact-cps 2 (lambda (v) (k (* 4 (* 3 v))))
  -> (fact-cps 1 lambda (v)
        ((lambda (v)
          (k (* 4 (* 3 v)))) (* 2 v))))
  -> (fact-cps 0 lambda (v)
        (k (* 4 (* 3 (* 2 v)))))
  -> (lambda (v)
        (k (* 4 (* 3 (* 2 v)))))
  -> (lambda (v)
        (k (* 4 (* 3 (* 2 v)))))
  -> (k 24)

(fact-cps 1 (lambda (v)
        ((lambda (v)
          (k (* 4 (* 3 v)))) (* 2 v))))
  -> (fact-cps 1 lambda (v)
        ((lambda (v)
          (k (* 4 (* 3 v)))) (* 2 v)))
  -> (fact-cps 0 lambda (v)
        (k (* 4 (* 3 (* 2 v)))))
  -> (k 24)

(fact-cps 4 k)
  -> (fact-cps 3 (lambda (v) (k (* 4 v))))
  -> (fact-cps 2 (lambda (v) (k (* 4 (* 3 v)))))
  -> (fact-cps 1 lambda (v)
        (k (* 4 (* 3 (* 2 v))))
        (k (* 4 (* 3 (* 2 v)))))
  -> (fact-cps 0 lambda (v)
        (k (* 4 (* 3 (* 2 v)))))
  -> (k 24)
Length of List

- Recursive Program
  ```lisp
  (define length
    (lambda (ls)
      (if (null? ls)
          0
          (+ (length (cdr ls)) 1)))))
  ```

CPS version of length

```lisp
(define length-cps
  (lambda (ls k)
    (if (null? ls)
        (k 0)
        (k
          (length (cdr ls) (lambda (v) (k (+ v 1))))))))
```

- How to invoke length-cps?
  ```lisp
  (length-cps '(a b c) (lambda (v) v))
  ```

Tail Position of Enclosing Expression

A subexpression is in a *tail position* of the enclosing expression if the result of the subexpression evaluation is immediately returned as the value of the enclosing expression.

Example

```lisp
( if <test-expr>
    <then-expr>
    <else-expr> )
```

*What is in tail position?*

Advantage of Tail Position

- When evaluating a subexpression in a tail position of the body of a procedure, > *no control information need be stored.*
- A subexpression $E'$ of $E$ is in a tail position with respect to $E$ if every subexpression of $E$ containing $E'$ is in a tail position of the expression within which it is immediately contained.
Tail Position Examples

(if (zero? x)
  (f (g x))
  (if (positive? x)
    1
    (f x)))

Tail Position

- An expression is in tail position if it is in tail position with respect to the nearest lexically enclosing lambda expression.

Head Position

- Those subexpressions that may be evaluated first are said to be in head position.

(lambda (v1 ... vn) E)
(primitive-operator H H ... H)
(if H T T)
(let (b H) ... (b H) T)

Available

- A subexpression $E'$ of $E$ is available in $E$ if it is not contained within a lambda subexpression of $E$.

(if (f a)
  (lambda () (g 2))
  (h x y))

Simple

An expression is simple if it does not contain an available application.

1. (car x)
2. (if p x (car (cdr x)))
3. (f (car x))
4. (car (f x))
5. (lambda (x) (car (f x)))

Tail forms

A tail-form expression is one where every subexpression in non-tail position is simple.

Which expressions are in tail form?

1. (car x)
2. (if p x (car (cdr x)))
3. (f (car x))
Tail form (examples contd.)

1. (car (f x))
2. (if p x (f (cdr x)))
3. (if (f x) x (f (cdr x)))
4. (lambda (x) (f x))
5. (lambda (x) (car (f x)))

CPS Transform

- No control information need be stored when evaluating tail form expressions.
- The CPS transformation converts any procedure into an equivalent procedure in tail form.

Steps in CSP Transform

- Rewrite every lambda expression so that it takes a continuation.
- The new body of each procedure is an application of \( k \) to the value of the old body.
  - (lambda (x1 ... xn) E) \( \rightarrow \) (lambda (x1 ... xn k) E)
- Repeatedly find an unprocessed application of the form (k E) and apply transformation rule.

Transformation Rules

- Transformation of Simple Expressions
- Transformation of Simple Applications
- Transformation of Applications

Transformation of Simple Expressions

\( C_{\text{simple}} \):

- If \( E \) is simple, stop.
- 1. (k (cons v w))
- 2. (k 0)
- But \( E \) may have unprocessed lambda expressions.
  - (k (lambda (x y) (g (~ x y))))
- 3. (k (lambda (x y k) (k (g (~ x y)))))

Transforming of Applications with Simple Subexpressions

\( C_\eta \): if \( E = (E_1 \ E_2 \ldots \ E_n) \) where \( E_1 \ E_2 \ldots \ E_n \) are simple:

- (k E) \( \rightarrow \) (E_1 E_2 \ldots E_n k)
- Example:
  - (k (lambda (x y) (g (~ x y)))) \( \rightarrow \) (lambda (x y k) (g (~ x y k)))
Observation

Let \( E = (E_1, E_2, ..., E_n) \) where \( E_1, E_2, ..., E_n \) are simple.

\( E_1, E_2, ..., E_n \) may still contain unprocessed lambda expressions.

Definitions

- Innermost Subexpression
  - a subexpression evaluated without requiring control space.
- Initial Subexpression
  - Is the subexpression first evaluated.

Innermost Subexpression

An expression whose immediate subexpressions in head position are all simple:

1. \( (f \ x \ y) \)
2. \( (f \ (+ \ x \ y) \ z) \)
3. \( (\text{if} \ (> \ x \ y) \ (f \ (g \ x)) \ (h \ (j \ x))) \)

Head position of entire expression marked in red

Initial Subexpression

An initial expression of \( E \) is an innermost non-simple subexpression that can be evaluated first in \( E \):

1. If \( E \) is non-simple but all its immediate subexpressions in head position are simple, \( E \) is its own initial expression.
2. Look for subexpressions in head position...

Examples

1. \( (p \ x \ y) \)
2. \( (\text{if} \ (\text{null?} \ x) \ (f \ x \ y) \ z) \) special form
3. \( (g \ (f \ a \ b)) \)
   - \( (h \ (p \ x \ y)) \)
   - \( (\text{if} \ (\text{null?} \ x) \ (f \ x \ y) \ z) \)
4. \( (f \ (* \ a \ b) \ 3) \)
5. \( (+ \ (\text{fact} \ 3) \ (* \ (\text{fact} \ (\text{fib} \ 2)) \ 5)) \)
6. \( (\text{let} \ ((y \ 4)) \ (f \ 4)) \) special form

Examples of Possible Initial Expressions

1. \( (p \ x \ y) \)
2. \( (\text{if} \ (\text{null?} \ x) \ (f \ x \ y) \ z) \) special form
3. \( (g \ (f \ a \ b)) \)
   - \( (h \ (p \ x \ y)) \)
   - \( (\text{if} \ (\text{null?} \ x) \ (f \ x \ y) \ z) \)
4. \( (f \ (* \ a \ b) \ 3) \)
5. \( (+ \ (\text{fact} \ 3) \ (* \ (\text{fact} \ (\text{fib} \ 2)) \ 5)) \)
6. \( (\text{let} \ ((y \ 4)) \ (f \ 4)) \) special form
Transforming Applications

- \( C_{app} \): If \((k \ E)\) has an initial expression \(I\), where \(I = (E_1 \ E_2 \ldots \ E_n)\)

\( (k \ E) \rightarrow (E_1 \ E_2 \ldots \ E_n \ (\lambda (v) \ (k \ E \ {v / I}))) \)

Examples

1. \((k \ (h \ [(p \ x \ y)])) \) initial expression
   \( \rightarrow (p \ x \ y \ (\lambda (v) \ (k \ (h \ v)))) \)
   \( \rightarrow (p \ x \ y \ (\lambda (v) \ (h \ v \ k))) \)

Special Forms

Conditionals

\( (k \ (if \ (null? \ ls) \ 0 \ (+ \ (length \ (cdr \ ls)) \ 1))) \)

\( \rightarrow (if \ (null? \ ls) \ (k \ 0) \ (k \ (+ \ (length \ (cdr \ ls)) \ 1))) \)

\( \rightarrow (if \ (null? \ ls) \ (k \ 0) \ (length \ (cdr \ ls) \ (\lambda (v) \ (k \ (+ \ v \ 1)))))) \)

Another Example

\( (k \ (* \ (if \ (null? \ ls) \ x \ (+ \ (length \ (cdr \ ls)) \ 1)) \ 3)) \)

\( \rightarrow (if \ (null? \ ls) \ (k \ (* \ x \ 3)) \ (k \ (* \ (+ \ (length \ (cdr \ ls)) \ 1) \ 3))) \)

Example

\( (k \ (* \ (if \ (null? \ ls) \ x \ (+ \ (length \ (cdr \ ls)) \ 1)) \ 3)) \)

\( \rightarrow (if \ (null? \ ls) \ (k \ (* \ x \ 3)) \ (k \ (* \ (+ \ (length \ (cdr \ ls)) \ 1) \ 3))) \)

Review Questions

1. What are simple expressions?
2. What are non-simple expressions?
3. What are innermost subexpressions?
4. What are initial expressions of the entire expression?
Variable Capture

• Naive application of the transformation may cause free variables captured incorrectly.
  
  \[ (k \ (g \ (let \ ((g \ 3)) \ (f \ g \ x))) \rightarrow (let \ ((g \ 3)) \ (k \ (g \ (f \ g \ x)))) \]  
  wrong transformation!

Avoiding Variable Capture

• Correct transformation -- need renaming bound variables: called \( \alpha \)-conversion
  
  \[ (k \ (g \ (let \ ((g: \ 3)) \ (f \ g: \ x))) \rightarrow (let \ ((g: \ 3)) \ (f \ g: \ x \ (lambda \ (v) \ (k \ (g \ v)))) \rightarrow (let \ ((g: \ 3)) \ (f \ g: \ x \ (lambda \ (v) \ (g \ v)))) ) \]

Example:

CPS transformation of remove

\[
\begin{align*}
&\text{(define remove} \\
&\quad \text{(lambda} \ (s \ los) \\
&\quad \quad (if \ (null? \ los) \\
&\quad \quad \quad '()) \\
&\quad \quad (if \ (eq? \ s \ (\text{car los})) \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los})) \\
&\quad \quad \text{(cons} \ \text{(car los)} \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los))))))))\end{align*}
\]

Introduce continuation

\[
\begin{align*}
&\text{(define remove} \\
&\quad \text{(lambda} \ (s \ los \ k) \\
&\quad \quad (k \ (if \ (null? \ los) \\
&\quad \quad \quad '()) \\
&\quad \quad (if \ (eq? \ s \ (\text{car los})) \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los})) \\
&\quad \quad \text{(cons} \ \text{(car los)} \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los))))))))} \end{align*}
\]

Apply the \( C_f \) rule

\[
\begin{align*}
&\text{(define remove} \\
&\quad \text{(lambda} \ (s \ los \ k) \\
&\quad \quad (if \ (null? \ los) \\
&\quad \quad \quad (k \ '()) \\
&\quad \quad (k \ (if \ (eq? \ s \ (\text{car los})) \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los})) \\
&\quad \quad \text{(cons} \ \text{(car los)} \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los))))))))))\end{align*}
\]

Apply the \( C_f \) rule (again)

\[
\begin{align*}
&\text{(define remove} \\
&\quad \text{(lambda} \ (s \ los \ k) \\
&\quad \quad (if \ (null? \ los) \\
&\quad \quad \quad (k \ '()) \\
&\quad \quad (if \ (eq? \ s \ (\text{car los})) \\
&\quad \quad \quad \text{(k} \ \text{(remove s} \ (\text{cdr los})) \\
&\quad \quad \text{(k} \ \text{(cons} \ \text{(car los)} \\
&\quad \quad \quad \text{(remove s} \ (\text{cdr los))))))))))))\end{align*}
\]
Applying $C_\eta$

```
(define remove
  (lambda (s los k)
    (if (null? los)
        (k '( ))
        (if (eq? s (car los))
            (remove s (cdr los) k)
            (k (cons (car los)
                (remove s (cdr los))))) )))
```

Applying $C_{app}$

```
(define remove
  (lambda (s los k)
    (if (null? los)
        (k ' )
        (if (eq? s (car los))
            (remove s (cdr los) k)
            (remove s (cdr los)
              (lambda (v)
                (k (cons (car los) v)))))))
```

Converting to CPS

1. $\{\cdot\}$ transforms the entire program into a procedure accepting a continuation which performs the same computations
   - P is the program, $\langle\langle P \rangle\rangle$ proc (val) val returns the same answer as P

Converting to CPS (contd)

2. $\langle\langle\cdot\rangle\rangle$ takes a simple expression and transforms it by modifying each procedure definition.
   - Add the formal parameter k for continuation
3. $\langle\langle E \rangle\rangle \circ \langle\langle K \rangle\rangle$ puts the expression E in tail form with simple expression K as continuation.

Execution Sequence of a Procedure Call

1. Save the current control context
2. Extend the environment with new variable bindings for the arguments of the call, and
3. Transfer control to the beginning of the called procedure's body.
   - The CPS transformation eliminates step 1.
   - If we eliminate step 2, leaving only step 3, then the calling sequence may be implemented by simple goto statement (i.e., unconditional jump).

Observation

- When a procedure calls itself from a tail position, the bindings of its formal parameters will never be used again.
- The new values required by the call could be assigned to the bindings prior to the call.
Example

```scheme
(define member? (lambda (s los) (cond ((null? los) #f) ((equal? (car los) s) #t) (else (member? s (cdr los))))))
```

Registers

```scheme
(define member? (lambda (s los) (cond ((null? los) #f) ((equal? (car los) s) #t) (else (begin (set! los (cdr los)) (member? s los))))))
```

- s and los are not changed by the tail call, they could both be global variables (registers)

Imperative version of member?

```scheme
(define s-reg '*)
(define los-reg '*)
(define member? (lambda (s los) (set! s-reg s) (set! los-reg los) (member?/reg)))
```

Imperative version of member? (contd.)

```scheme
(define member?/reg (lambda () (cond ((null? los-reg) #f) ((equal? (car los-reg) s-reg) #t) (else (begin (set! los-reg (cdr los-reg)) (member?/reg))))))
```

Imperative Version of fact-iter

```scheme
(define n-reg '*)
(define a-reg '*)
(define fact-iter (lambda (n) (set! n-reg n) (set! a-reg 1) (fact-iter-acc/reg))))
```

Imperative version (contd.)

```scheme
(define fact-iter-acc/reg (lambda () (if (zero? n-reg) a-reg (begin (set! a-reg (* n-reg a-reg)) (set! n-reg (- n-reg 1)) (fact-iter-acc/reg))))))
```
(define remove
 (lambda ( )
 (if (null? los)
 (k `() )
 (if (eq? s (car los))
 (begin
 (set! los (cdr los))
 (remove s los k))
 (begin
 (set! los (cdr los))
 (set! k (lambda (v) (k (cons (car los) v)))))
 (remove s los k))))