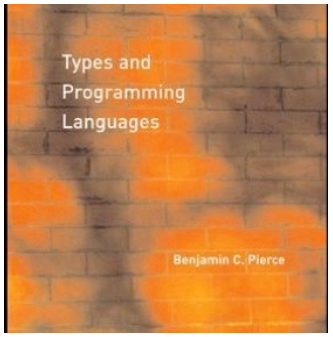
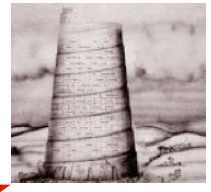
Continuations

CS 565

Lecture 6

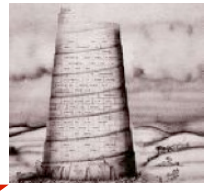


Continuations



- Evaluation contexts provide a syntactic formulation to specify evaluation order.
- Can we understand evaluation contexts from the perspective of language-level constructs?
  - How does one reify the notion of a “hole” in a program?

Continuations and CPS



- Starting point:
  - How do we represent a program's control-flow?

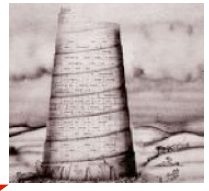
    Loops

    Procedure call and return

    Order of evaluation
  - Our previous semantic characterizations kept these notions implicit.
  - Can we make these sequences explicit?

    How can we express evaluation contexts within a program?

Example: Recursion

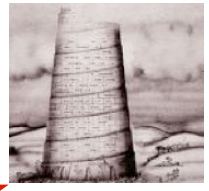


Consider a factorial function:

```
fun fact(n:int):int = if n = 0
                      then 1
                      else n * fact(n-1)
```

Each call to fact is made with a “promise” that the value returned will be multiplied by the value of n at the time of the call.

Example: Tail Recursion

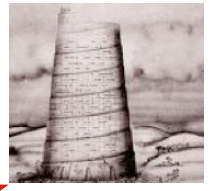


- Now, consider:

```
let fun fact-iter(n:int):int =
    let fun loop(n:int,acc:int):int =
              if n = 0
              then acc
              else loop(n - 1, n * acc)
    in loop(n,1)
    end
```

There is no promise made in the call to loop by fact-iter, or in the inner calls to loop: each call simply is obligated to return its result. Unlike fact, no extra control state (e.g., promise) is required; this information is supplied explicitly in the recursive calls.
What is the implication of these different approaches?
    Recursive vs. iterative control

Tail position



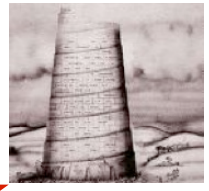
- An expression in tail position requires no additional control-information to be preserved.
  - Intuitively, no state information needs to be saved.
  - Examples:

    The true and false branches of an if-expression.

    A loop iteration.

    A function call that occurs as the last expression of its enclosing definition.
  - Tail recursive implementations can execute an arbitrary number of tail-recursive calls without requiring memory proportional to the number of these calls.

Continuation-passing style



- Is a technique that can translate any procedure into a tail recursive one.
- Example:

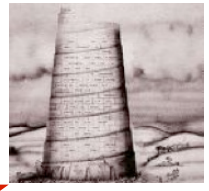
```
4 * 3 * 2 * fact(1)
```

  - Define the context of fact(1) to be

```
fun Context(v:int):int = 4 * 3 * 2 * v
```

  - The context is a function that given the value produced by `fact(1)` returns the result of `fact(4)`

Example revisited

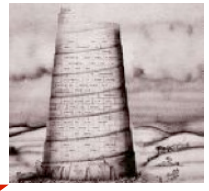


```
fun fact-cps(n:int, k: int -> int): int =
    if n = 0
    then k(1)
    else fact-cps(n-1, fn v => k (n * v))
```

- The `k` represents the *function's continuation*: it is a function that given a value returns the "rest of the computation"
- By making `k` explicit in the program, we make the control-flow properties of `fact` also explicit, which will enable improved compiler decisions.

  Observe that `k(fact(n))=fact-cps(n,k)` for any `k`.

Example revisited

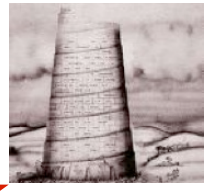


- `fact-cps(4,k)` ⟶

```
fact-cps(3, fn v => k(4 * v))
fact-cps(2, fn v => (fn v => k(4 * v))(3 * v))
fact-cps(2, fn v => (fn v => k(4 * v))(3 * v))
fact-cps(2, fn v =>  k ( 4 * 3 * v))
fact-cps(1, fn v => (fn v => k ( 4 * 3 * v)) (2 * v))
fact-cps(1, fn v => (fn v => k ( 4 * 3 * v))(2 * v))
fact-cps(1, fn v => k (4 * 3 * 2 * v))
                  …
fact-cps(0, fn v => k (4 * 3 * 2 * 1 * v))
(fn v => k (4 * 3 * 2 * 1 * v)) 1
k 24
```

The initial k supplied to fact-cps represents the “context” in which the call was made.

First cut

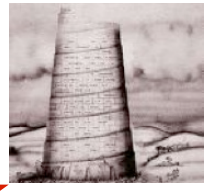


- Start with a very simple language:
  - Variables, functions, applications, and conditionals.
- Define a translation function:

  `C : Exp x Cont → Exp`

  - A continuation will be represented as a function that takes a single argument, and performs “the rest of the computation”
  - The translation will ensure that functions never directly return -- they always invoke their continuation when they have a value to provide.
  - In an interpreter, the top-level continuation simply prints the value passed as argument to the screen.

The CPS Algorithm



- $C[x]\ k = k\ x$

  Returning the value of a variable simply passes that value to the current continuation.

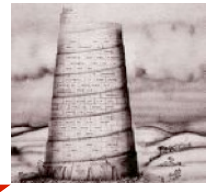
- $C[\lambda x.e]\ k = k\ (\lambda x.\lambda k_2.C[e]\ k_2)$

  A function takes an extra argument which represents the continuation(s) of its call point(s), and its body is evaluated in this context.

- $C[e_1(e_2)]\ k = C[e_1]\ (\lambda v.C[e_2]\ \lambda v_2.v\ v_2\ k)$

  An application evaluates its first argument in the context of a continuation that evaluates its second argument in the context of a continuation that performs the application and supplies the result to its context.

The Initial Algorithm

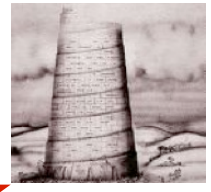


- $C[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\ k =$
  $C[e_1]\ (\lambda v.\text{if } v \text{ then } C[e_2]\ k \text{ else } C[e_3]\ k)$

Evaluate the test expression in a context that evaluates the true and false branch in the context of the conditional.
Note that `k` is duplicated in both branches. We would like to avoid this.

```
C[if true then x else y] k                =
C[true](λv.if v then C[x]k else C[y]k)    =
C[true](λv.if v then k x else k y )       =
(λv.if v then k x else k y )true          →
k x
```

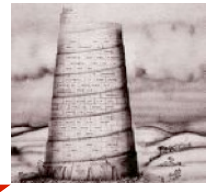
Example



$$
\begin{aligned}
&C[(\text{plus1 } x)]\ k = \\
&C[\text{plus1}]\ (\lambda v.C[x]\ (\lambda v_2.v\ v_2\ k)) = \\
&(\lambda v.C[x]\ (\lambda v_2.v\ v_2\ k))\ \text{plus1} = \\
&(\lambda v.(\lambda v_2.v\ v_2\ k)\ x)\ \text{plus1} = \\
&(\lambda v.(\lambda v_2.v\ v_2\ k)\ x)\ \text{plus1} \rightarrow \\
&(\lambda v_2.\ \text{plus1}\ v_2\ k)\ x \rightarrow \\
&\text{plus1 } x\ k \rightarrow
\end{aligned}
$$

What's plus1 in CPS?

Example



What's plus1 in CPS form?

$\lambda x.\text{succ } x$ =

$\lambda x.\lambda k.C[\text{succ } x]\ k$ =

$\lambda x.\lambda k.C[\text{succ}]\ (\lambda v.C[x](\lambda v_2.v\ v_2\ k))$ =

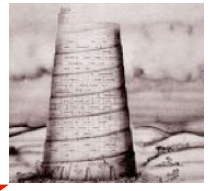
$\lambda x.\lambda k.(\lambda v.C[x](\lambda v_2.v\ v_2\ k))\text{succ}$ =

$\lambda x.\lambda k.(\lambda v.(\lambda v_2.v\ v_2\ k)x)\text{succ}$ →

$\lambda x.\lambda k.(\lambda v.(v\ x\ k))\text{succ}$ →

$\lambda x.\lambda k.(\text{succ } x\ k)$

Correspondence



- What do continuations (and CPS) have to do with evaluation contexts?
  - CPS also fixes evaluation order.
  - Is a continuation a representation of an evaluation context?

    It represents an expression with a hole (its argument). When supplied a value, the hole is "plugged" and a new expression is produced.

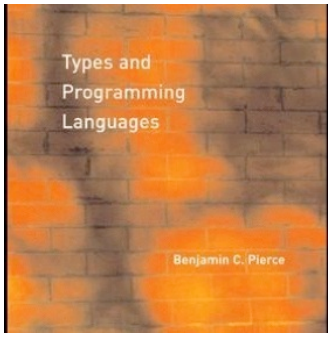
    Not quite: CPS produces continuations that given an intermediate result return the final result of the computation.

    Two incompatible definitions:

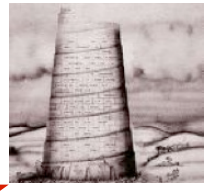
    - a context expects an expression, and returns another expression
    - a continuation expects a value and returns a final result.

Lambda-Calculus (cont): Fixpoints, Naming

Lecture 10
CS 565



Lambda Calculus



$termvar,\ x$ term variable

t				term
	$\mid$	x		variable
	$\mid$	$\lambda\, x\, .\, t$	bind x in t	lambda
	$\mid$	$t\ t'$		app

v				value
	$\mid$	$\lambda\, x\, .\, t$		lambda

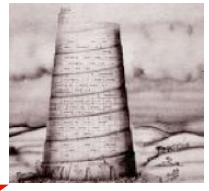
$\boxed{t_1 \longrightarrow t_2}$ t_1 reduces to t_2

$$\frac{}{(\lambda\, x\, .\, t_{12})\ v_2 \longrightarrow \{\, v_2 \,/\, x \,\}\, t_{12}} \quad \text{AX\_APP}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t \longrightarrow t_1'\ t} \quad \text{CTX\_APP\_FUN}$$

$$\frac{t_1 \longrightarrow t_1'}{v\ t_1 \longrightarrow v\ t_1'} \quad \text{CTX\_APP\_ARG}$$

Recursion and Divergence



Consider the application:

$$\Omega \equiv ((\lambda\ x.(x\ x))\ (\lambda\ x.(x\ x)))$$

Ω evaluates to itself in one step.

It has no normal form.

A lambda term is in normal form if it does not contain any redex (i.e., a term that is subject to β-reduction)

Now, consider: $Y \equiv ((\lambda\ x.(f\ (x\ x)))(\lambda\ x.(f\ (x\ x))))$

$Y \rightarrow$

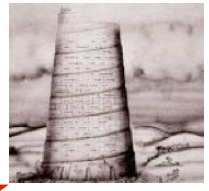
$(f\ ((\lambda\ x.(f\ (x\ x)))\ (\lambda\ x.(f\ (x\ x))))) \rightarrow$

$(f\ (f\ (\lambda\ x.(f\ (x\ x)))\ (\lambda\ x.(f\ (x\ x)))))) \rightarrow$

...

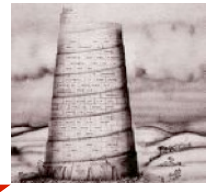
$(f(f(...(f\ (\lambda\ x.(f\ (x\ x)))(\lambda\ x.(f(x\ x)))...)))$

Normal forms and order of evaluation



- No expression can be converted to two distinct normal forms (Church-Rosser Theorem 1)
- Is there an order of evaluation guaranteed to terminate whenever a particular expression is reducible to normal form?
  - Normal-order: leftmost, outermost reduction: no expression in the argument position of a redex is reduced until the redex is reduced
  - If there is a reduction from `A` to `B` and `B` is in normal form, then there exists a normal order reduction from `A` to `B` (Church-Rosser Theorem 2)

Recursion



The previous definition applies f an infinite number of times

Basis for iterated application

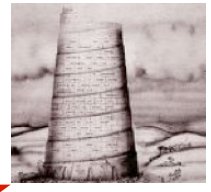
But, how can we slow its rate of unfolding?

Consider:

$$\Omega_v \equiv (\lambda\ y.(\ (\lambda\ x.(\lambda\ y.(x\ x\ y)))\ (\lambda\ x.(\lambda\ y.(x\ x\ y)))\ y)\ )$$

Ω_v is in normal form. However, if it is applied to an argument it diverges.

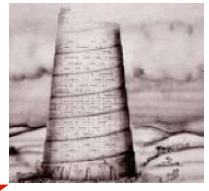
Recursion (cont)



$(\Omega_v\ v) \rightarrow$

```
((λ y.((λ x.(λ y.(x x y)))
       (λ x.(λ y.(x x y)))
     y)
 v) →
( ((λ x.(λ y.(x x y)))
   (λ x. (λ y. (x x y))
) v)  →
(λ x. (λ y. (x x y))) (λ x. (λ y. (x x y))) v))))
))  →
```

Recursion (cont)



Now, consider

$$Z_f \equiv (\lambda\ y.((\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ (\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ y))$$

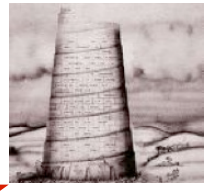
If we apply Z_f to an argument:

$$((\lambda\ y.((\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ (\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ y)\ v) \rightarrow$$

$$(f\ (\lambda\ y.((\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ (\lambda\ x.(f\ (\lambda\ y.(x\ x\ y))))\ y))\ v) = f\ Z_f\ v$$

Since the arguments to f are all values, this expression is equivalent to: f Z_f v

Recursion (cont)

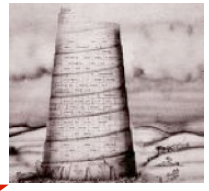


How do we apply these insights?

```
f ≡ λ fact.
        λ n. if n = 0
                then 1
                else n * (fact (n – 1))
```

We can use Z_f to turn `f` into a real factorial function:

Fixpoints



Z_f 3 →

f Z_f 3 =

(λ fact. λ n. ...) Z_f 3 →

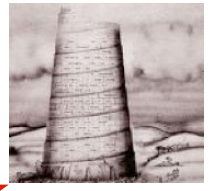
if 3 = 0 then 1 else 3 * (Z_f 2) →

3 * (f Z_f 2)

...

We'll stop when n = 0

Fixpoints



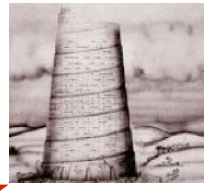
Define $Z = \lambda f.Z_f$

Now, Z defines a fixpoint for any f:

$$Z \equiv \lambda f.\ (\lambda y.((\lambda x.(f\ (\lambda y.(x\ x\ y))))\ (\lambda x.(f\ (\lambda y.(x\ x\ y))))\ y))$$

Z computes the least fixpoint of a function.

Fixpoints and order of evaluation



- Consider an alternative definition:

  $Y \equiv \lambda h.(\lambda x.h(x\ x))(\lambda x.h(x\ x))$

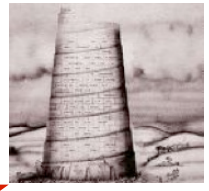
  - What happens if we apply Y to f (the factorial functional) with argument 3?
  - Under normal-order evaluation:

    $Y\ f\ 3 \equiv (\lambda x.f(x\ x))(\lambda x.f(x\ x))\ 3 \rightarrow$

    $f\ ((\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x)))\ 3$

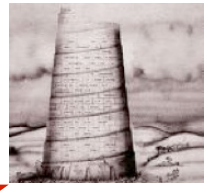
  - What happens under applicative-order?

Naming and substitution



- Although we claimed that lambda calculus essentially manipulates functions (it does), we've spent a lot of time thinking about variables
  - substitutions
  - free variables
  - equivalence upto renaming
- Implementations must consider these issues seriously
  - Rename bound variables when performing substitutions with "fresh" names.
  - Impose a condition that all bound variables be distinct from each other, and other free variables.
  - Derive a canonical representation that does not require renaming at all.

Terms and Contexts



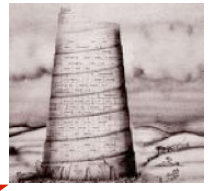
- De Brujin indices:
  - Have variable occurrences “point” directly to their binders rather than referring to them by name.
  - Do so by replacing variable occurrences with numbers:

    number k stands for “the variable bound by the kth enclosing λ-term

    Example: λ x.λ y.x (y x) ≡ λ.λ.1(0 1)

    Similar to static offsets in an activation record or display.

Examples



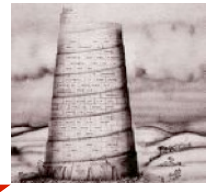
identity ≡ λ x. x ≡ λ .0

true ≡ λ x. λ y. x ≡ λ.λ. 1

false ≡ λ x. λ y. y ≡ λ.λ. 0

two ≡ λ s. λ z. s (s z) ≡ λ . λ . (1 (1 0))

Contexts



How do we replace free variables with their binders?

Assume an ordered context listing all free variables that can occur, and map free variables to their index in this context (counting right to left)

Context: a, b

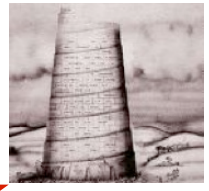
$a \mapsto 1,\ b \mapsto 0$

$\lambda\ x.\ a \equiv \lambda\ .\ 2$

$\lambda\ x.\ b \equiv \lambda.\ 1$

$\lambda\ x.b\ (\lambda\ y.\ a) \equiv \lambda.1(\lambda.3)$

Shifting and substitution



- When substituting into a λ term, indices must be adjusted:

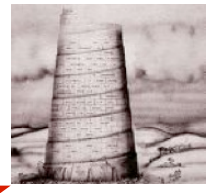
  λ y.x[z/x] in context x,y,z

  [2↦0]λ.2 ≡ λ.[3↦1]3 ≡ λ.1

- Key point: context becomes longer when substituting inside an abstraction. Need to be careful to adjust free variables, not bound ones.

```
shift(d,c)(k) = k      if k < c
                k + d  if k ≥ c
shift(d,c)(λ.t) = (λ.shift(d,c+1)(t))
shift(d,c)(t t')= (shift(d,c)(t))
                  (shift(d,c)(t'))
```

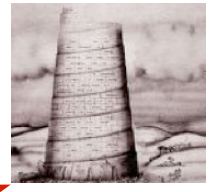
Example



shift(2,0)(λ.λ. 1 (0 2)) = λ.λ. 1 (0 4)

shift(2,0)(λ.0 1(λ.0 1 2)) = λ.0 3 (λ.0 1 4)

Substitution



$$[\ j \mapsto s]\ k = s \text{ if } k = j$$
$$k \text{ otherwise}$$

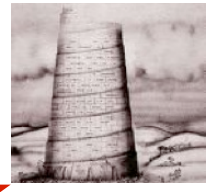
$$[j \mapsto s](\lambda\ .t\ ) = \lambda.\ [j+1 \mapsto \text{shift}(1,0)s]\ t$$

$$[j \mapsto s](t1\ t2) = ([j \mapsto s)\ t1)\ ([j \mapsto s]\ t2)$$

- Beta-reduction:

$$(\lambda\ t)\ v \rightarrow \text{shift}(-1,0)([0 \mapsto \text{shift}(1,0)(v)]\ t)$$

Examples

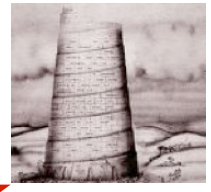


- Assume context <a,b>
  - Then, a ↦ 1, b ↦ 0

```
[a / b] b λ x. λ y. b
        [0 ↦ 1] 0 λ . λ . 2
        1 λ . λ . 3 ≡  a λ x. λ y. a

[(a (λ z. a)) / b] (b (λ x.b))
        [0 ↦ (1 (λ. 2))] (0 λ. 1)
        1 (λ . 2) (λ . (2 (λ . 3)))
        (a (λ z. a)) (λ x. (a (λ z. a)))
```

Examples



```
[a/b] (λ b. (b a))
      [0 ↦ 1] (λ . (0 2))
      (λ . (0 2))
      (λ b. (b a))
[a/b] (λ a. (b a))
      [0 ↦ 1] λ . (1 0)
      λ . (2 0)
     (λ a′. a a′)
```