CS 456

Programming Languages
Fall 2024

Week 3
Recursion, Fixpoints, Continuations

Reasoning about Control

λ-calculus provides no explicit support for

- loops
- recursive functions
- other forms of control

But, Church's thesis claims that any computable algorithm can be implemented using it. How?

Recursion and Divergence

Consider the application:

Ω ≡ `((λ x. (x x)) (λ 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 ≡ ((λ x. (f (x x))) (λ x. (f (x x))))`

```
Y →
(f ((λ x. (f (x x))) (λ x. (f (x x))))) →
(f (f (λ x. (f (x x))) (λ x. (f (x x)))))) →
...
(f (f ( ... (f (λ x. (f (x x))) (λ x. (f (x x))) ...)))
```

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:

```
Ωv ≡ (λ y. ((λ x. (λ y. (x x y)))
            (λ x . (λ y. (x x y)))
            y))
```

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

Recursion (cont)


```
(Ωv v) →

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

Ωv ≡ ((λ x. (λ y. (x x y)))
      (λ x. (λ y. (x x y)))
     y)
  →
...
```

Recursion (cont)

Now, consider

```
Zf ≡ (λ y. ((λ x . (f (λ y. (x x y))))
            (λ x. (f (λ y. (x x y))))
            y))
```

If we apply Z_f to an argument:

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

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

Since the arguments to f are all values, this expression is equivalent to: `f Zf 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 = λ f. z`$_f$

Now, Z defines a fixpoint for any f:

```
Z ≡ λ f. (λ y. ((λ x. (f (λ y. (x x y))))
                (λ x. (f (λ y. (x x y))))
                y))
```

Z computes the *least fixpoint* of a function.

Fixpoints and order of evaluation

Consider an alternative definition:

```
Y ≡ λh.(λx.h( x x))(λx.h(x x))
```

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

```
Y f ≡ (λx. f(x x))(λx. f(x x)) 3 → f ((λx. f(x x))(λx. f(x x))) 3
```

- What happens under applicative-order?

Control-Flow

- Programs manifest control in a number of ways:
  - loops
  - exceptions
  - gotos
  - procedure call
  - argument evaluation
  - message-passing
  - threads and scheduling
  - ...

- Is there a uniform way to represent these different constructs in the λ-calculus?

Example

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 (cont)

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.

More generally, it makes explicit the "linearization" of control that is otherwise implicit in a program

Example:

4 * 3 * 2 * fact(1)

Define the *context* of fact(1) to be

```
fn v => 4 * 3 * 2 * v
```

Here, 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))   by def. of fact-cps
   fact-cps(2,  fn v =>  k ( 4 * 3 * v)             by beta-conversion
   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.

Translation

Start with a very simple λ-calculus based 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 perform "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.

A Simple Algorithm

$C\,[\,x\,]\,k \;=\; k\;x$

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

$C[\,\lambda x.e\,]\,k = k\,(\lambda x\;k'\,.\,C\,[\,e\,]\,k')$

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

$C[\,e1(e2)\,]\,k = C\,[\,e1\,]\,\lambda v.\,C\,[\,e2\,]\,\lambda v'.v\,(v', k)$

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.

Algorithm (cont)

C [if e1 then e2 else e3] k =

C[e1] λv. if v then C[e2]k else C[e3]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.

Example

C [(x1(x2) x3)] k →

C [x1(x2)] λv1 . C [x3] λv2. v1(v2,k) →

C [x1(x2)] λv1 . (λv2. v1(v2,k)) x3 →

k'

C [x1] λv3 . C [x2] λv4. v3(v4,k') →

(λv3. (λv4. v3(v4,k')) x2)

x1)

Example (cont)

C [x1] λv3 . C [x2] λv4. v3(v4,k′) →

(λv3. (λv4. v3(v4, k′) x2) x1) →

(λv3. (λv4. v3(v4, k′) x2) x1) →

x1(x2, k′) →

x1(x2, (λv1 . (λv2. v1(v2,k)) x3)) →

x1(x2, (λv1 . v1(x3, k)))

Some Observations

CPS addresses two aspects of a program's control-flow:

- order of evaluation of arguments
- call/return sequences

Can we separate these two concerns?

Can we construct a theory that captures the essence of tail and non-tail calls?

Can we reason about a program's control-flow without the need to introduce explicit continuations?

A-normal form

Consider a language with the following grammar:

```
M ::= v                                [RETURN]
    | let x = V in M                   [BIND]
    | if V then M else M               [BRANCH]
    | V(V1, … ,VN)                        [TAIL-CALL]
    | let x = (V V1… VN) in M          [NON-TAIL]
    | P(V1 … Vn)                       [PRIMITIVE CALL]
v ::= c | x | λx1…xn.M                 [VALUES]
```

A-normal form

- All continuations are implicit.
  - But, like CPS all intermediate expressions are named
  - And, control-flow is apparent from syntactic structure of the program
  - Tail calls distinguished from non-tail calls. Recall that a tail call is a function call that occurs as the last statement in the calling function.

A Calculus of A-Reductions

- How do we think of continuations without an explicit lambda term to capture control-flow?
- An *evaluation context* is a term with a "hole" corresponding to the next expression to be evaluated. (The context surrounding the "hole" is an implicit representation of the continuation for any term substituted for the hole.)

```
E ::= [ ]
    | let x = E  in M
    | if E then M else M
    | F(V … V E M … M) (where F = V or F = O)
```

`M` is a term and `V` is a value as defined earlier; neither contain "holes." Thus, the structure of this grammar forces a left-to-right evaluation.

Evaluation Contexts

Example:

```
E [ let x = [ ] in M ]
```

defines an evaluation context that consists of the let expression and outer context E. We can substitute a term for the hole, treating this context as its continuation.

A-reductions

Rule A1

```
E [ let x = M in N)] →
 let x = M in  E[N] where E != [] and x not in FV(E)
```

Purpose:

Lifts out nested let declarations from expressions by merging them with an outer context.

Role of side conditions:

- An empty context requires no transformation
- Free variable capture rule assumes program is not alpha-converted

Example

Original expression:

```
let x = let y = M in N
in  N1
```

Pick `E` as `let x = [   ] in N1` and "fill" `let y = M in N` for that hole:

```
E [ let y = M in N ]
→ let y = M in E [ N ]
→ let y = M
  in let x = N
     in N1
```

Net effect: Complex intermediate expressions defined via `let` lifted out.

A-reductions

Rule A2

```
E [if V then M1 else M2] →
    if V then E [M1] else E [M2]  where E != [ ]
```

Purpose:

Lifts out nested expressions from conditionals by merging the expression with an outer context. Note duplication of contexts in conditional branches

Example

Original expression:

```
F(V1,  if V then N1 else N2,  M1, …, Mn)
```

Pick `E = F(V1, [ ], M1, …, Mn)` and fill `if V then N1 else N2` for the hole.

```
E [ if V then N1 else N2 ] →
  if V then F(V1, N1, M1, … Mn)
       else F(V1, N2, M1, … Mn)
```

A-reductions

Rule A3

```
E [ F(V1, … ,Vn) ]
→ let t = F(V1, …, VN)
   in E [t]
```

where `F = V or F = 0,`

```
     E != E' [let z = [ ] in M]
     E != [ ]
     t not in FV (E)
```

Purpose: lift and name nested applications

Role of side-conditions:

Second side condition prevents extraneous reductions, and to prevent non-termination of the transformation; subsumed by rule A1

Last condition can be prevented by alpha-conversion

Example

Original expression:

```
f(g(x))
```

Pick `E` to be `(f [ ])` and substitute `g(x)` for the hole in `E`.

```
E [ g(x) ] →
let t = g(x)
in f(t)
```

Net effect: nested applications lifted out of complex expressions, and intermediate values named. Clear identification of non-tail calls.

Putting it all together


```
(2 + 2) + (let x = 1 in f(x))
  --> let t1 = 2 + 2                  (By rule A3)
       in t1 + (let x = 1 in f(x))
  --> let t1 = 2 + 2                  (By rule A1)
       in let x 1
           in t1 + f(x)
  --> let t1 = 2 + 2                  (By rule A3)
       in let x = 1
           in let t2 = f(x)
               in t1 + t2
```