Lambda Calculus and Recursion

Overview

- Nameless functions
  - Functionals
- The Genie solution
- Fixpoints
  - The f91 function
- The Combinator solution
  - The Y combinator
  - Fixpoints

Factorial Definition

- Objective
  - Write a lambda expression for factorial

```
false  = λx.λy.y
true   = λx.λy.x
isZero = λn.n (λx.FALSE) TRUE
1      = λs.λz.s z
mult   = λn.λm. <multiplication expression>
if     = λc.λt.λe.c t e
```

```
fact  = λn.if (isZero n) (1)
            (mult n (fact (pred n)))
```

Nameless Function

- Problem
  - The definition of fact uses the name fact itself
  - Lambda expression are unnamed
- Solution ?

```
fact  = λf.λn.if (isZero n) (1) (mult n (f (pred n)))
```

- This is called a *functional*
  - A function that
    - Given an *f*
    - Performs one application of factorial on the value of *f*.

Genie Solution

- Do we have a solution ?
- Assume the existence of a genie
  - He gives us a lambda expression that
    - Computes *(n-1)!*
    - *Cannot* compute *n!*
- Can we compute *n!* With the Genie's help?
  - Genie gives
    - F: int -> int = `λn.<genie expression>`
  - We have

```
fact = λf.λn.if (isZero n) (1)(mult n (f (pred n)))
```

Status

- Good news!
  - With a Genie, the problem is solved!
- Bad news.....
  - We do not have a Genie...
- What we *really* want is
  - A lambda expression whose ***ultimate meaning*** is *n!*

Overview

- Nameless functions
  - Functionals
- The Genie solution
- Fixpoints
  - The f91 function
- The Combinator solution
  - The Y combinator
  - Fixpoints

A Funny Function

- Consider the function

```
fun M x = if x>100
           then x-10
           else M(M(x+11))
```

- Behavior
  - For x > 100   x – 10
  - For x <= 100   91

```
fun M x = if x>100
           then x-10
           else 91
```

Ultimate Meaning

Restriction

- If we restrict the input between 0..100 then

$$\forall x \in [0..100] M(x) = 91$$

- In particular

$$M(91) = 91$$

- 91 is a fixed point of M
- For any function f, x is a fixpoint of f iff

$$x = f(x)$$

- The fixpoint is the ultimate meaning of f.

Questions

- Do fixpoints always exists ?
- Are fixpoints unique ?
- With
  - Proper pre-requisites
    - Continuity
    - Complete partial order (cpo) on underlying domain
  - Functions have a unique least fix point.

Back to Factorial

- The least fixpoint is the ultimate meaning
- Question
  - Give our definition of fact

```
fact = λf.λn.if (isZero n) (1)(mult n (f (pred n)))
```

  - Can we derive a fixpoint for it ?

```
Z = fact Z
```

- Answer...
  - Yes!
  - The Y combinator
  - Z = Y fact

```
Y fact = fact (Y fact)
```

The Y combinator

- The combinator must satisfy the fixpoint equation

```
Y f = f (Y f)
```

```
Y = λf.(λx.f (x x)) (λx.f (x x))
```

$$
\begin{aligned}
\texttt{Y f} &= \texttt{(λf.(λx.f (x x)) (λx.f (x x))) f} \\
&=_{\beta} \texttt{(λx.f (x x)) (λx.f (x x))} \\
&=_{\beta} \texttt{f (λx.f (x x)) (λx.f (x x))} \\
&=_{\eta} \texttt{f ((λg.((λx.g (x x))(λx.g (x x))) f)} \\
&= \texttt{f (Y f)}
\end{aligned}
$$

Multiple Application

- Keep applying Y

```
Y f →β f (Y f)
    →β f (f (Y f))
    →β f (f (f (Y f)))
    →β ....
```

Finally, Factorial!

- Putting it together
  - The function *n!* Is *(Y fact) n*

```
fact = λf.λn.if (isZero n) (1)(mult n (f (pred n)))
     = λf.M
```

```
Y fact 2    →β fact (Y fact) 2
            →β λf.M (Y fact) 2
            →β [(Y fact)/f]M n 2
            →β λn.if (isZero n)1(mult n ((Y fact)(pred n)))2

                 →β if (isZero 2) 1 (mult 2 ((Y fact)(pred 2)))
                 →β if false 1 (mult 2 ((Y fact)(pred 2)))
            →β (mult 2 ((Y fact)(pred 2)))
```

Finally, Factorial!

```
→β (mult 2 ((Y fact)(pred 2)))
→β (mult 2 ((Y fact) 1))

→β (mult 2 ((fact (Y fact) 1))
→β (mult 2 ((λf.M (Y fact) 1))
→β (mult 2 (([(Y fact)/f]M 1))

→β (mult 2  (λn.if (isZero n)1(mult n ((Y fact)(pred n)))1)
→β (mult 2 (if (isZero 1) 1 (mult 1 ((Y fact) (pred 1)))))

→β (mult 2 (mult 1 ((Y fact) 0)))
→β (mult 2 (mult 1 (fact (Y fact) 0)))
→β (mult 2 (mult 1 (λf.M (Y fact) 0)))
→β (mult 2 (mult 1 ([(Y fact)/f]M 0)))
```

Finally, Factorial!

$\rightarrow_\beta$ (mult 2 (mult 1 ([(Y fact)/f]M 0)))

$\rightarrow_\beta$ (mult 2 (mult 1 (λn.if (isZero n)1(mult n ((Y fact)(pred n)))0)))

$\rightarrow_\beta$ (mult 2 (mult 1 (if (isZero 0) 1 (mult 0 ((Y fact)(pred 0))))))

$\rightarrow_\beta$ (mult 2 (mult 1 (1)))

$\rightarrow_\beta$ 2

Observation

- For this to work we need
  - Normal order reduction!
  - Why ?
- The Y combinator is
  - Independent of fact
  - It works for any functional

Combinators and λ Equivalence

- The pair of combinators S,K
  - Form a turing-complete base
  - You can write any program with them
  - Completely equivalent to `λ-calcul`
- How to establish that ?
  - Provide two mappings
    - Combinators to λ
    - λ to combinators

Combinators to λ

- Quite easy!
  - Use the combinators definition
  - Create a *mapping* L: C → λ

L[I]	= λx.x
L[K]	= λx.λy.x
L[S]	= λx.λy.λz.x z (y z)
L[C_1 C_2]	= L[C_1] L[C_2]

λ to combinators

- A bit harder...
  - Same idea though!
  - Provide a *mapping* T: λ → C

```
T[x]              = x

T[(E₁ E₂)]        = (T[E₁] T[E₂])

T[λx.x]           = I

T[λx.E]           = (K  T[E])          x is not free in E

T[λx.λy.E]        = T[λx.T[λy.E]]          x is free in E

T[λx.(E₁ E₂)]      = (S T[λx.E₁] T[λx.E₂])
```

Doing a Translation

- Translate the following lambda term

```
T[λx.λy.y x] =  ?????
```

Bottom Line

- Facts
  - You can translate *any* lambda term into an SK-term
  - You can translate *any SK term* into a lambda term
- SK is a combinatory calculus
  - It also has a syntax
  - It has even simpler semantic rules
  - It is as powerful as lambda-calculus
  - Therefore it is as powerful as a Turing Machine

SK Syntax

- Quite simple!

$$
\begin{aligned}
\mathit{Term} \quad &::= \mathrm{S}\\
&::= \mathrm{K}\\
&::= \mathrm{I}\\
&::= (\mathit{Term}\ \mathit{Term})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Term} \quad &::= \mathrm{S}\\
&::= \mathrm{K}\\
&::= (\mathit{Term}\ \mathit{Term})
\end{aligned}
$$

SK Semantics

- Similar to lambda calculus
  - 3 Rules…

SK Semantics: Rule 1

- Rule 1 : Identity
  - Assume that you have a *derivation* $\triangle$ ending with….
  - $\triangle = e_0 \to e_1 \to \ldots \to \alpha\ (I\ \beta)\ \iota$
    - Then you can extend it to
  - $\triangle = e_0 \to e_1 \to \ldots \to \alpha\ (I\ \beta)\ \iota \to \alpha\ \beta\ \iota$

SK Semantics: Rule 2

- Rule 2 : Constant combinator
  - Assume that you have a *derivation* $\triangle$ ending with….
  - $\triangle = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow \alpha\ ((K\ \beta)\ \gamma)\ \iota$
    - Then you can extend it to
  - $\triangle = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow \alpha\ ((K\ \beta)\ \gamma)\ \iota \rightarrow \alpha\ \beta\ \iota$

SK Semantics: Rule 3

- Rule 3 : Composition
  - Assume that you have a *derivation* $\triangle$ ending with….
  - $\triangle = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow \alpha\ (((S\ \beta)\ \gamma)\ \delta)\ \iota$
    - Then you can extend it to
  - $\triangle = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow \alpha\ (((S\ \beta)\ \gamma)\ \delta)\ \iota \rightarrow \alpha\ ((\beta\delta)\ (\gamma\delta))\ \iota$

Stop when you become irreducible
(when you can't extend anymore!

Summary

- Lambda calculus
  - A very simple language
  - Very expressive
    - Can compute any recursive function
    - Same expressive power as Turing machine
  - Useful to study
    - Programs as expression
    - Computation as reduction
    - Meaning as normal form

Summary

- Lambda calculus
  - A very simple language
  - Very expressive
    - Can compute any recursive function
    - Same expressive power as Turing machine
  - Useful to study
    - Programs as expression
    - Computation as reduction
    - Meaning as normal form