COMP2111 Week 9
Term 1, 2019
Introduction to Lambda Calculus

Summary

- History
- Functional programming
- Lambda calculus

Summary

- History
- Functional programming
- Lambda calculus

Entscheidungsproblem

1928: David Hilbert asks if there is a "mechanical procedure" that, given a finite set of first-order formulas T, and and formula φ, decides if

$$T \models \varphi$$

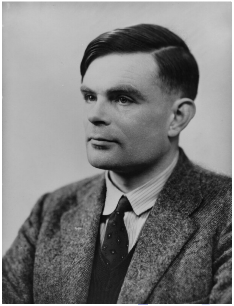
1936: Alonzo Church and Alan Turing independently show there isn't

Entscheidungsproblem

1928: David Hilbert asks if there is a "mechanical procedure" that, given a finite set of first-order formulas T, and and formula φ, decides if

$$T \models \varphi$$

1936: Alonzo Church and Alan Turing independently show there isn't



Formally defining an algorithm

Turing: Mechanical process (Turing Machines)

Church: Logical process (Lambda calculus)

Gödel: General recursive functions

All approaches are equivalent!

Church-Turing thesis

Every effectively calculable function is equivalent to one computed by a Turing Machine.

Formally defining an algorithm

Turing: Mechanical process (Turing Machines)

Church: Logical process (Lambda calculus)

Gödel: General recursive functions

All approaches are equivalent!

Church-Turing thesis

Every effectively calculable function is equivalent to one computed by a Turing Machine.

Lambda calculus in a nutshell

- Everything is a function:
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting
- Lead to the concept of **Functional programming**

Lambda calculus in a nutshell

- **Everything** is a function:
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting
- Lead to the concept of **Functional programming**

Lambda calculus in a nutshell

- **Everything** is a function:
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting
- Lead to the concept of **Functional programming**

Lambda calculus in a nutshell

- **Everything** is a function:
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting
- Lead to the concept of **Functional programming**

Summary

- History
- Functional programming
- Lambda calculus

Functional programming

What is Functional Programming?

- Programming paradigm distinct from Imperative programming, Object Oriented programming
- Extensively used in academia. Can be found in industry (e.g. Jane Street)
- Covered in COMP3161 (others?)
- Languages: Haskell, ML, OCaml, Scala

Examples

Example

`leaves` and `internal` functions from Assignment 1.

A tree is either:

- Empty: τ
- A node with two trees as children: $\text{Node}(t_1, t_2)$

`leaves` defined recursively as:

- $\texttt{leaves}(\tau) = 0$
- $\texttt{leaves}(\text{Node}(\tau, \tau)) = 1$
- $\texttt{leaves}(\text{Node}(t_1, t_2)) = \texttt{leaves}(t_1) + \texttt{leaves}(t_2)$

`internal` defined recursively as:

- $\texttt{internal}(\tau) = -1$
- $\texttt{internal}(\text{Node}(\tau, \tau)) = 0$
- $\texttt{internal}(\text{Node}(t_1, t_2)) = 1 + \texttt{internal}(t_1) + \texttt{internal}(t_2)$

Examples

Example

`leaves` and `internal` functions from Assignment 1.

In Haskell:

```
Tree = Empty | Node Tree Tree

leaves Empty  = 0
leaves (Node Empty Empty) = 1
leaves (Node t1 t2) = leaves t1 + leaves t2

internal Empty = -1
internal (Node Empty Empty) = 0
internal (Node t1 t2) = 1 + internal t1
                          + internal t2
```

Functional programming

Guiding principles:

- Everything is a function (more-or-less)
- Programs are pure (no side-effects)

Pros/cons:

- Easy to prove properties: theoretically well-behaved
- Interactivity is complicated: I/O, Error handling

Functional programming

Guiding principles:

- Everything is a function (more-or-less)
- Programs are pure (no side-effects)

Pros/cons:

- Easy to prove properties: theoretically well-behaved
- Interactivity is complicated: I/O, Error handling

Currying

A function of n variables can be viewed as a function of 1 variable that returns a function of $n-1$ variables

Example

- Consider $f : \mathbb{N}^2 \to \mathbb{N}$ given by $f(x, y) = x + 2y$.
- For every $x \in \mathbb{N}$ let $g_x : \mathbb{N} \to \mathbb{N}$ be given by $g_x(y) = x + 2y$
- Now consider $h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ given by $h(x) = g_x$. We have:

$$h(x)(y) = f(x, y)$$

In general:

$$(A \times B \to C) \cong (A \to (B \to C))$$

Currying

A function of n variables can be viewed as a function of 1 variable that returns a function of $n-1$ variables

Example

- Consider $f : \mathbb{N}^2 \to \mathbb{N}$ given by $f(x,y) = x + 2y$.
- For every $x \in \mathbb{N}$ let $g_x : \mathbb{N} \to \mathbb{N}$ be given by $g_x(y) = x + 2y$
- Now consider $h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ given by $h(x) = g_x$. We have:

$$h(x)(y) = f(x,y)$$

In general:

$$(A \times B \to C) \cong (A \to (B \to C))$$

Summary

- History
- Functional programming
- Lambda calculus

Lambda calculus in a nutshell

- Everything is a function of one variable
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting

Lambda calculus in a nutshell

- Everything is a function of one variable
  - Booleans, numbers, ...
- "Computation" is captured with function application and rewriting

Lambda calculus: formally

SYNTAX: A λ-term is defined recursively as follows:

- x is a λ-term for any variable x
- (Application) If M and N are λ-terms then MN is a λ-term
- (Abstraction) If M is a λ-term then $\lambda x.M$ is a λ-term

SEMANTICS: Intuitively:

- MN corresponds to the result of passing N as the argument to the function M (applying M to N)
- $\lambda x.M$ is the definition of a new function that binds x to be the (independent) variable of the function (e.g. anonymous functions)

Lambda calculus: formally

SYNTAX: A λ-term is defined recursively as follows:

- x is a λ-term for any variable x
- (Application) If M and N are λ-terms then MN is a λ-term
- (Abstraction) If M is a λ-term then $\lambda x.M$ is a λ-term

SEMANTICS: Intuitively:

- MN corresponds to the result of passing N as the argument to the function M (applying M to N)
- $\lambda x.M$ is the definition of a new function that binds x to be the (independent) variable of the function (e.g. anonymous functions)

Lambda calculus: Examples

Example

The following are λ-terms:

- $\lambda x.(\lambda y.y)$
- $\lambda x.(\lambda y.x)$
- $\lambda x.(\lambda y.xy)$
- $\lambda n.\lambda f.\lambda x.f(nfx)$
- $\lambda p.(\lambda q.(pq)p)$

Reductions

Reductions are rewrite rules.

α-**reductions** correspond to variable refactoring:

- Rename bound variables, e.g.:

$$\lambda x.(\lambda y.x) \quad \xrightarrow{\alpha} \quad \lambda z.(\lambda y.z)$$

Reductions

β-**reductions** correspond to function evaluation (i.e. computation):

- Only applies to λ-terms of the form $M'N$ where M' is of the form $\lambda x.M$
- Substitute occurrences of x with N, that is:

$$(\lambda x.M)N \quad \xrightarrow{\beta} \quad M[N/x]$$

- For example:

$$(\lambda x.xx)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)$$

Reductions

β-**reductions** correspond to function evaluation (i.e. computation):

- Only applies to λ-terms of the form $M'N$ where M' is of the form $\lambda x.M$
- Substitute occurrences of x with N, that is:

$$(\lambda x.M)N \quad \xrightarrow{\beta} \quad M[N/x]$$

- For example:

$$(\lambda x.xx)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)$$

Reductions

β-**reductions** correspond to function evaluation (i.e. computation):

- Only applies to λ-terms of the form $M'N$ where M' is of the form $\lambda x.M$
- Substitute occurrences of x with N, that is:

$$(\lambda x.M)N \quad \xrightarrow{\beta} \quad M[N/x]$$

- For example:

$$(\lambda x.xx)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)(\lambda y.y) \quad \xrightarrow{\beta} \quad (\lambda y.y)$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$\begin{aligned}
(AX)Y &= ((\lambda p.(\lambda q.(pq)p))X)Y \\
&\xrightarrow{\beta} (\lambda q.(Xq)X))Y \\
&\xrightarrow{\beta} (XY)X \\
&\xrightarrow{\beta} ((\lambda x.(\lambda y.x))Y)X \\
&\xrightarrow{\beta} (\lambda y.Y)X \\
&\xrightarrow{\beta} Y
\end{aligned}$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$\begin{aligned}
(AX)Y &= ((\lambda p.(\lambda q.(pq)p))X)Y \\
&\xrightarrow{\beta} (\lambda q.(Xq)X))Y \\
&\xrightarrow{\beta} (XY)X \\
&\xrightarrow{\beta} ((\lambda x.(\lambda y.x))Y)X \\
&\xrightarrow{\beta} (\lambda y.Y)X \\
&\xrightarrow{\beta} Y
\end{aligned}$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$
\begin{aligned}
(AX)Y &= ((\lambda p.(\lambda q.(pq)p))X)Y \\
&\xrightarrow{\beta} (\lambda q.(Xq)X))Y \\
&\xrightarrow{\beta} (XY)X \\
&\xrightarrow{\beta} ((\lambda x.(\lambda y.x))Y)X \\
&\xrightarrow{\beta} (\lambda y.Y)X \\
&\xrightarrow{\beta} Y
\end{aligned}
$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$
\begin{aligned}
(AX)Y &= ((\lambda p.(\lambda q.(pq)p))X)Y \\
&\xrightarrow{\beta} (\lambda q.(Xq)X))Y \\
&\xrightarrow{\beta} (XY)X \\
&\xrightarrow{\beta} ((\lambda x.(\lambda y.x))Y)X \\
&\xrightarrow{\beta} (\lambda y.Y)X \\
&\xrightarrow{\beta} Y
\end{aligned}
$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$\begin{array}{rcl} (AX)Y & = & ((\lambda p.(\lambda q.(pq)p))X)Y \\ & \xrightarrow{\beta} & (\lambda q.(Xq)X))Y \\ & \xrightarrow{\beta} & (XY)X \\ & \xrightarrow{\beta} & ((\lambda x.(\lambda y.x))Y)X \\ & \xrightarrow{\beta} & (\lambda y.Y)X \\ & \xrightarrow{\beta} & Y \end{array}$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$\begin{array}{rcl} (AX)Y & = & ((\lambda p.(\lambda q.(pq)p))X)Y \\ & \xrightarrow{\beta} & (\lambda q.(Xq)X))Y \\ & \xrightarrow{\beta} & (XY)X \\ & \xrightarrow{\beta} & ((\lambda x.(\lambda y.x))Y)X \\ & \xrightarrow{\beta} & (\lambda y.Y)X \\ & \xrightarrow{\beta} & Y \end{array}$$

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $X = \lambda x.(\lambda y.x)$
- $A = \lambda p.(\lambda q.(pq)p)$

We have:

$$(AX)Y \to^* Y$$

Similarly we can show:

$$(AY)X \to^* Y \qquad (AY)Y \to^* Y \qquad (AX)X \to^* X$$

So A behaves like $\wedge$ if we view X as `true` and Y as `false`

Lambda calculus: Examples

Example

Consider the following λ-terms:

- $Y = \lambda x.(\lambda y.y)$
- $S = \lambda n.\lambda f.\lambda x.f(nfx)$

It is possible to show

$$\begin{array}{rcl} SY & \to^* & \lambda x.(\lambda y.xy) \\ S(SY) & \to^* & \lambda x.(\lambda y.x(xy)) \\ S(S(SY)) & \to^* & \lambda x.(\lambda y.x(x(xy))) \\ \vdots & \vdots & \vdots \end{array}$$

Lambda calculus: Examples

Example

What happens if we try to reduce:

$$(\lambda x.xx)(\lambda x.xx)?$$

Lambda calculus: Further topics

- Normal forms
- Typing
- Combinators (e.g. defining recursion)
- Combinatory logic