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 $\varphi$, 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 a formula $\varphi$, 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: \( \tau \)
- A node with two trees as children: \( \text{Node}(t_1, t_2) \)

leaves defined recursively as:
- \( \text{leaves}(\tau) = 0 \)
- \( \text{leaves}(\text{Node}(\tau, \tau)) = 1 \)
- \( \text{leaves}(\text{Node}(t_1, t_2)) = \text{leaves}(t_1) + \text{leaves}(t_2) \)

internal defined recursively as:
- \( \text{internal}(\tau) = -1 \)
- \( \text{internal}(\text{Node}(\tau, \tau)) = 0 \)
- \( \text{internal}(\text{Node}(t_1, t_2)) = 1 + \text{internal}(t_1) + \text{internal}(t_2) \)
Examples

**Example**

leaves and internal functions from Assignment 1.

In Haskell:

\[
\text{Tree} = \text{Empty} \mid \text{Node Tree Tree}
\]

\[
\text{leaves Empty} = 0 \\
\text{leaves (Node Empty Empty)} = 1 \\
\text{leaves (Node t1 t2)} = \text{leaves t1} + \text{leaves t2}
\]

\[
\text{internal Empty} = -1 \\
\text{internal (Node Empty Empty)} = 0 \\
\text{internal (Node t1 t2)} = 1 + \text{internal t1} + \text{internal t2}
\]
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 $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ given by $f(x, y) = x + 2y$.
- For every $x \in \mathbb{N}$ let $g_x : \mathbb{N} \rightarrow \mathbb{N}$ be given by $g_x(y) = x + 2y$.
- Now consider $h : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ given by $h(x) = g_x$. We have:
  $$h(x)(y) = f(x, y)$$

In general:

$$(A \times B \rightarrow C) \cong (A \rightarrow (B \rightarrow C))$$
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 $\lambda$-term is defined recursively as follows:
- $x$ is a $\lambda$-term for any variable $x$
- (Application) If $M$ and $N$ are $\lambda$-terms then $MN$ is a $\lambda$-term
- (Abstraction) If $M$ is a $\lambda$-term then $\lambda x.M$ is a $\lambda$-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 \( \lambda \)-term is defined recursively as follows:

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

The following are $\lambda$-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.

$\alpha$-reductions correspond to variable refactoring:

- Rename bound variables, e.g.:

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

$\beta$-reductions correspond to function evaluation (i.e. computation):

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

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

- For example:

\[(\lambda x. xx)(\lambda y. y) \xrightarrow{\beta} (\lambda y. y)(\lambda y. y) \xrightarrow{\beta} (\lambda y. y)\]
Reductions

\(\beta\)-reductions correspond to function evaluation (i.e. computation):

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

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

- For example:

\[
(\lambda x. xx)(\lambda y. y) \xrightarrow{\beta} (\lambda y. y)(\lambda y. y) \xrightarrow{\beta} (\lambda y. y)
\]
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:

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

- For example:

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

$$(AX)Y \rightarrow (((\lambda p.(\lambda q.(pq)p))X)Y$$

$\beta \rightarrow (\lambda q.(Xq)X))Y$

$\beta \rightarrow (XY)X$

$\beta \rightarrow ((\lambda x.(\lambda y.x))Y)X$

$\beta \rightarrow (\lambda y.Y)X$

$\beta \rightarrow 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:

$$(AX)Y \beta\rightarrow ((\lambda p.(\lambda q.(pq)p))X)Y$$

$$(\lambda q.(Xq)X))Y \beta\rightarrow (XY)X$$

$$(\lambda y.(\lambda y.x))Y \beta\rightarrow (\lambda y.Y)X$$

$$(\lambda y.Y)X \beta\rightarrow Y$$
Example

Consider the following \( \lambda \)-terms:

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

We have:

\[
(AX)Y = (((\lambda p.(\lambda q.(pq)p))X)Y)
\]

\[
\beta \rightarrow (\lambda q.(Xq)X))Y
\]

\[
\beta \rightarrow (XY)X
\]

\[
\beta \rightarrow (((\lambda x.(\lambda y.x))Y)X)
\]

\[
\beta \rightarrow (\lambda y.Y)X
\]

\[
\beta \rightarrow Y
\]
Lambda calculus: Examples

Example

Consider the following \( \lambda \)-terms:

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

We have:

\[
(AX)Y = (((\lambda p.(\lambda q.(pq)p))X)Y) \\
\beta \rightarrow (\lambda q.(Xq)X))Y \\
\beta \rightarrow (XY)X \\
\beta \rightarrow (((\lambda x.(\lambda y.x))Y)X) \\
\beta \rightarrow (\lambda y.Y)X \\
\beta \rightarrow Y
\]
Example

Consider the following $\lambda$-terms:

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

We have:

$$(AX)Y = (\lambda p.(\lambda q.(pq)p))X)Y$$

$\beta$-reduce:

$$\rightarrow (\lambda q.(Xq)X))Y$$
$$\rightarrow (XY)X$$
$$\rightarrow ((\lambda x.(\lambda y.x))Y)X$$
$$\rightarrow (\lambda y.Y)X$$
$$\rightarrow Y$$
Example

Consider the following \( \lambda \)-terms:

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

We have:

\[
(AX)Y = ((\lambda p. (\lambda q. (pq)p))X)Y
\]

\[
\beta \rightarrow (\lambda q. (Xq)X))Y
\]

\[
\beta \rightarrow (XY)X
\]

\[
\beta \rightarrow ((\lambda x. (\lambda y. x))Y)X
\]

\[
\beta \rightarrow (\lambda y. Y)X
\]

\[
\beta \rightarrow Y
\]
Example

Consider the following \( \lambda \)-terms:

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

We have:

\[(AX)Y \rightarrow^* Y\]

Similarly we can show:

\[(AY)X \rightarrow^* Y \quad (AY)Y \rightarrow^* Y \quad (AX)X \rightarrow^* X\]

So \( A \) behaves like \( \land \) if we view \( X \) as true and \( Y \) as false.
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

\[
\begin{align*}
SY & \rightarrow^* \lambda x.(\lambda y.xy) \\
S(SY) & \rightarrow^* \lambda x.(\lambda y.x(xy)) \\
S(S(SY)) & \rightarrow^* \lambda x.(\lambda y.x(x(xy))) \\
\vdots & \quad \vdots & \quad \vdots
\end{align*}
\]
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