Lecture 4: Lambda Calculus

Models of Computation

https://clegra.github.io/moc/moc.html

Clemens Grabmayer

Ph.D. Program, Advanced Courses Period
Gran Sasso Science Institute
L'Aquila, Italy

July 10, 2025

Course overview

Monday, July 7 10.30 – 12.30	Tuesday, July 8 10.30 – 12.30	Wednesday, July 9 10.30 – 12.30	Thursday, July 10 10.30 – 12.30	Friday, July 11
intro	*classic models*			*additional models*
Introduction to Computability	**Machine Models**	**Recursive Functions**	**Lambda Calculus**	
computation and decision problems, from logic to computability, overview of models of computation relevance of MoCs	Post Machines, typical features, Turing's analysis of human computers, Turing machines, basic recursion theory	primitive recursive functions, Gödel–Herbrand recursive functions, partial recursive funct's, partial recursive = = Turing-computable, Church's Thesis	λ-terms, β-reduction, λ-definable functions, partial recursive = λ-definable = Turing computable	
	imperative programming	*algebraic programming*	*functional programming*	
				14.30 – 16.30
				Three more Models of Computation
				Post's Correspondence Problem, Interaction-Nets, Fractran comparing computational power

Today

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules

Today

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions

Today

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions
- primitive recursive functions are λ-definable

Today

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions
- primitive recursive functions are λ-definable
- μ-recursive/partial recursive functions are λ-definable

Today

Lambda calculus

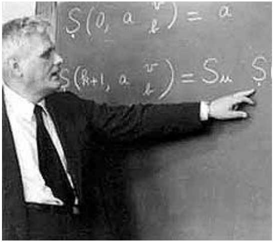
- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions
- primitive recursive functions are λ-definable
- μ-recursive/partial recursive functions are λ-definable
- λ-definable functions are Turing computable

Today

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions
- primitive recursive functions are λ-definable
- μ-recursive/partial recursive functions are λ-definable
- λ-definable functions are Turing computable
- Hence: λ-definable = partial recursive = Turing-computable

Church's Thesis



Alonzo Church (1903 –1995)

Thesis (Church, 1936)

- *Every total effectively calculable function is recursive.*
- *Every effectively calculable partial function is partial-recursive.*

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

Notation conventions:

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

Notation conventions:

- omit outermost brackets
  - x short for (x), and $\lambda x.x$ short for $(\lambda x.x)$

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

Notation conventions:

- omit outermost brackets
  - x short for (x), and $\lambda x.x$ short for $(\lambda x.x)$
- application associates to the left
  - $MNPQ$ is short for $((MN)P)Q$

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

Notation conventions:

- omit outermost brackets
  - x short for (x), and $\lambda x.x$ short for $(\lambda x.x)$
- application associates to the left
  - $MNPQ$ is short for $((MN)P)Q$
- abstraction associates to the right
  - $\lambda xy.M$ is short for $\lambda x.(\lambda y.M)$

λ-terms

Definition

- variables: $x, y, z, x_1, y_1, z_1, \ldots \in \mathbf{\Lambda}$
- λ-abstraction: x a variable, $M \in \mathbf{\Lambda} \implies (\lambda x.M \in \mathbf{\Lambda})$
- application: $M, N \in \mathbf{\Lambda} \implies (MN) \in \mathbf{\Lambda}$

Notation conventions:

- omit outermost brackets
  - x short for (x), and $\lambda x.x$ short for $(\lambda x.x)$
- application associates to the left
  - $MNPQ$ is short for $((MN)P)Q$
- abstraction associates to the right
  - $\lambda xy.M$ is short for $\lambda x.(\lambda y.M)$
- scope of $\lambda(\cdot)$ is as big as possible
  - $\lambda x.yx$ is short for $\lambda x.(yx)$
  - note: $(\lambda x.y)x$ is different from $\lambda x.yx$

β-reduction

Definition

- One-step β-reduction $\to_\beta$ is defined as the application of the rule:

$$(\lambda x.M)N \quad \to_\beta \quad M\{x := N\}$$

in λ-terms $C[(\lambda x.M)N]$ formed by arbitrary λ-term contexts $C[\,]$, where is $\lambda x.MN$ called a redex, and furthermore:

$M\{x := N\}$:= substitution of N for free occurrences of x in M
(using α-conversion to avoid variable capture)

β-reduction

Definition

- One-step β-reduction $\to_\beta$ is defined as the application of the rule:

  $$(\lambda x.M)N \quad \to_\beta \quad M\{x := N\}$$

  in λ-terms $C[(\lambda x.M)N]$ formed by arbitrary λ-term contexts $C[\,]$, where is $\lambda x.MN$ called a redex, and furthermore:

  $M\{x := N\}$:= substitution of N for free occurrences of x in M (using α-conversion to avoid variable capture)

- Many-step β-reduction $\to_\beta^*$ is defined as the concatenation of zero, one, or more $\to_\beta$-steps.

β-reduction

Definition

- One-step β-reduction $\rightarrow_\beta$ is defined as the application of the rule:

  $$(\lambda x.M)N \;\; \rightarrow_\beta \;\; M\{x := N\}$$

  in λ-terms $C[(\lambda x.M)N]$ formed by arbitrary λ-term contexts $C[\,]$, where is $\lambda x.MN$ called a redex, and furthermore:

  $M\{x := N\}$:= substitution of N for free occurrences of x in M (using α-conversion to avoid variable capture)

- Many-step β-reduction $\rightarrow_\beta^*$ is defined as the concatenation of zero, one, or more $\rightarrow_\beta$-steps.
- A λ-term M is a normal form if it does not contain a redex.

Church numerals

Definition

For every $n \in \mathbb{N}$, the Church numeral $\ulcorner n \urcorner$ for n is defined by:

$$\ulcorner n \urcorner := \lambda f x . f^n x$$

Church numerals

Definition

For every $n \in \mathbb{N}$, the Church numeral $\ulcorner n \urcorner$ for n is defined by:

$$\begin{aligned} \ulcorner n \urcorner &:= \lambda f x. f^n x \\ &= \lambda f x. \underbrace{f(f(\ldots(f}_{n} x)\ldots)) \end{aligned}$$

Church numerals

Definition

For every $n \in \mathbb{N}$, the Church numeral $\ulcorner n \urcorner$ for n is defined by:

$$\begin{aligned} \ulcorner n \urcorner &:= \lambda f x. f^n x \\ &= \lambda f x. \underbrace{f(f(\ldots(f}_{n}\, x)\ldots)) \end{aligned}$$

Examples.

$$\begin{aligned} \ulcorner 0 \urcorner &= \lambda f x. x \\ \ulcorner 1 \urcorner &= \lambda f x. f x \\ \ulcorner 2 \urcorner &= \lambda f x. f(f x) \\ &\ldots \end{aligned}$$

□

Turing-computable (total) functions

Definition

A total function $f : \mathbb{N}^k \to \mathbb{N}$ is Turing-computable if there exists a Turing machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \not{b}, F \rangle$ and a calculable coding function $\langle\cdot\rangle : \mathbb{N} \to \Sigma^*$ such that:

- for all $n_1, \ldots, n_k \in \mathbb{N}$ there exists $q \in F$ such that:

$$q_0 \langle n_1 \rangle \not{b} \langle n_2 \rangle \not{b} \ldots \not{b} \langle n_k \rangle \vdash_M^* q \langle f(n_1, \ldots, n_k) \rangle$$

λ-definable functions

Definition

- Let $f : \mathbb{N}^n \to \mathbb{N}$ be total.
  A λ-term M_f represents f if for all $m_1, \ldots, m_n \in \mathbb{N}$:

$$M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner \ \to_\beta^* \ \ulcorner f(m_1, \ldots, m_n) \urcorner$$

  f is λ-definable if there exists a λ-term that represents f.

λ-definable functions

Definition

- Let $f : \mathbb{N}^n \to \mathbb{N}$ be total.
  A λ-term M_f represents f if for all $m_1, \ldots, m_n \in \mathbb{N}$:

  $$M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner \ \to_\beta^* \ \ulcorner f(m_1, \ldots, m_n) \urcorner$$

  f is λ-definable if there exists a λ-term that represents f.

- Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function.
  A λ-term M_f represents f if for all $m_1, \ldots, m_n \in \mathbb{N}$:

  $$f(m_1, \ldots, m_n)\downarrow \implies M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner \ \to_\beta^* \ \ulcorner f(m_1, \ldots, m_n) \urcorner$$

  $$f(m_1, \ldots, m_n)\uparrow \implies M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner \ \text{has no normal form}$$

  f is λ-definable if there exists a λ-term that represents f.

λ-definable

Examples.

- successor: $M_{\mathsf{succ}} := \lambda nfx.f(nfx)$
- addition: $M_{+} := \lambda mnfx.mf(nfx)$
- multiplication: $M_{\times} := \lambda mnfx.m(nf)x$
- exponentiation: $M_{\mathsf{E}} := \lambda mnfx.mnfx$
- unary constant zero function: $M_{\mathsf{C}_0^1} = \lambda m.\ulcorner 0 \urcorner$
- projection function: $M_{\pi_i^k} = \lambda n_1 \ldots n_k.n_i$

Pairs in λ-calculus

Definition

For all $M, N \in \mathbf{\Lambda}$ we define the pair $\langle M, N \rangle$ consisting of M and N:

$$\langle M, N \rangle := \lambda x.xMN$$

and the unpairing projections ρ_1 and ρ_2:

$$\rho_1 := \lambda p.p(\lambda xy.x)$$
$$\rho_2 := \lambda p.p(\lambda xy.y)$$

Proposition

For all $M_1, M_2 \in \mathbf{\Lambda}$ and $i = 1, 2$:

$$\rho_i \langle M_1, M_2 \rangle \rightarrow^*_\beta M_i$$

True, false, if-then-else, **zero?** in λ-calculus

Definition

$$\mathbf{true} := \lambda xy.x$$

$$\mathbf{false} := \lambda xy.y$$

$$\textbf{if } P \textbf{ then } Q \textbf{ else } R := PQR$$

$$\textbf{zero?} := \lambda x.x(\lambda y.\mathbf{false})\mathbf{true}$$

Proposition

$$\textbf{if } \mathbf{true} \textbf{ then } Q \textbf{ else } R \to_\beta^* Q$$

$$\textbf{if } \mathbf{false} \textbf{ then } Q \textbf{ else } R \to_\beta^* R$$

$$\textbf{zero?}\ \ulcorner 0 \urcorner \to_\beta^* \mathbf{true}$$

$$\textbf{zero?}\ \ulcorner n+1 \urcorner \to_\beta^* \mathbf{false}$$

Typical features of 'computationally complete' MoC's

- storage (unbounded)

Typical features of ‘computationally complete’ MoC’s

- storage (unbounded)
- control (finite, given)

Typical features of ‘computationally complete’ MoC’s

- storage (unbounded)
- control (finite, given)
- modification
  - of (immediately accessible) stored data
  - of control state

Typical features of 'computationally complete' MoC's

- storage (unbounded)
- control (finite, given)
- modification
  - of (immediately accessible) stored data
  - of control state
- conditionals

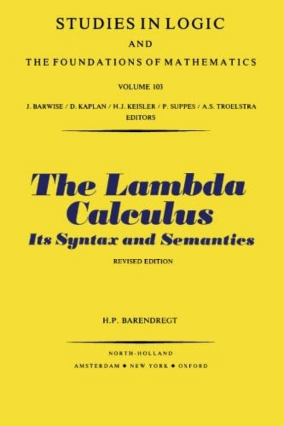
Typical features of ‘computationally complete’ MoC’s

- storage (unbounded)
- control (finite, given)
- modification
  - of (immediately accessible) stored data
  - of control state
- conditionals
- loop

Typical features of ‘computationally complete’ MoC’s

- storage (unbounded)
- control (finite, given)
- modification
  - of (immediately accessible) stored data
  - of control state
- conditionals
- loop
- stopping condition

The Book



(reference [1]) Hendrik Pieter (Henk) Barendregt

Exercises

(1) Describe all possible ways to reduce $(\lambda xy.x)((\lambda x.xx)(\lambda x.xx))$ to normal form.

(2) Find two distinct λ-terms representing the successor function on Church-numerals (hint: think of $n + 1$ and $1 + n$). Prove that your λ-terms are not-β-equivalent.

(3) Try computing the normal form of the Y-combinator, i.e. of AA where $A = \lambda am.m(aam)$, e.g. by each time selecting the leftmost redex (reducible expression, i.e. subexpression of the shape $(\lambda x.M)N$).

Primitive recursive functions ($\mathbb{N}^n \cup \mathbb{N}^0 \to \mathbb{N}$)

Base functions:

- $\mathcal{O} : \mathbb{N}^0 = \{\varnothing\} \to \mathbb{N}\,,\ \varnothing \mapsto 0$ (0-ary constant-0 function)
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}\,,\ x \mapsto x + 1$ (successor function)
- $\pi^n_i : \mathbb{N}^n \to \mathbb{N}\,,\ \vec{x} = \langle x_1, \ldots, x_n \rangle \mapsto x_i$ (projection function)

Closed under operations:

- composition: if $f : \mathbb{N}^k \to \mathbb{N}$, and $g_i : \mathbb{N}^n \to \mathbb{N}$ are prim. rec., then so is $h = f \circ (g_1 \times \ldots \times g_k) : \mathbb{N}^n \to \mathbb{N}$:
$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x}))$$
- primitive recursion: if $f : \mathbb{N}^n \to \mathbb{N}$, $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are prim. rec., then so is $h = \mathrm{pr}(f; g) : \mathbb{N}^{n+1} \to \mathbb{N}$:
$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y + 1) = g(\vec{x}, h(\vec{x}, y), y)$$

Primitive recursive functions are λ-definable

Proposition

Every primitive recursive function is λ-definable.

Primitive recursive functions are λ-definable

Proposition

Every primitive recursive function is λ-definable.

Proof (The case of primitive recursion).

Let $h := \mathrm{pr}(f; g) : \mathbb{N}^{n+1} \to \mathbb{N}$ for prim.rec. $f : \mathbb{N}^n \to \mathbb{N}$, $g : \mathbb{N}^{n+2} \to \mathbb{N}$:

$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y+1) = g(\vec{x}, h(\vec{x}, y), y)$$

Suppose that f and g are represented by $M_f, M_g \in \mathbf{\Lambda}$, respectively.

Primitive recursive functions are λ-definable

Proposition

Every primitive recursive function is λ-definable.

Proof (The case of primitive recursion).

Let $h := \mathrm{pr}(f;g) : \mathbb{N}^{n+1} \to \mathbb{N}$ for prim.rec. $f : \mathbb{N}^n \to \mathbb{N}$, $g : \mathbb{N}^{n+2} \to \mathbb{N}$:

$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y+1) = g(\vec{x}, h(\vec{x}, y), y)$$

Suppose that f and g are represented by $M_f, M_g \in \mathbf{\Lambda}$, respectively.

$$\text{Init} := \langle \ulcorner 0 \urcorner, M_f\, x_1 \ldots x_n \rangle$$
$$\text{Step} := \lambda p. \langle M_{\mathsf{succ}}(\rho_1 p), M_g x_1 \ldots x_n (\rho_2 p)(\rho_1 p) \rangle$$

Then the following λ-term M_h represents h:

$$M_h := \lambda x_1 \ldots x_n x. \rho_2(x\, \text{Step Init}\,)$$

$\square$

μ-recursion, and partial recursive functions

Definition

A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is called partial recursive if it can be specified from base functions ($\mathcal{O}$, succ, π^n_i) by successive applications of composition, primitive recursion, and unbounded minimisation.

A partial recursive function is called (total) recursive if it is total.

μ-recursion, and partial recursive functions

Definition

A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is called partial recursive if it can be specified from base functions ($\mathcal{O}$, succ, π_i^n) by successive applications of composition, primitive recursion, and unbounded minimisation.

A partial recursive function is called (total) recursive if it is total.

Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ total. Then the partial function defined by:

$$\mu(f) : \mathbb{N}^n \rightharpoonup \mathbb{N}$$
$$\vec{x} \mapsto \begin{cases} \text{min}(\{y \mid f(\vec{x}, y) = 0\}) & \dots\ \exists y\,(f(\vec{x}, y) = 0) \\ \uparrow & \dots\ \text{else} \end{cases}$$

is called the unbounded minimisation of f.

μ-recursion, and partial recursive functions

Definition

A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is called partial recursive if it can be specified from base functions ($\mathcal{O}$, succ, π_i^n) by successive applications of composition, primitive recursion, and unbounded minimisation.

A partial recursive function is called (total) recursive if it is total.

Let $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ partial. Then the partial function $\mu(f)$:

$$\mu(f) : \mathbb{N}^n \rightharpoonup \mathbb{N}$$

$$\vec{x} \mapsto \begin{cases} \uparrow & \dots \neg\exists y \left(\wedge\, f(\vec{x}, y) = 0 \forall z \left(0 \le z < y \to (f(\vec{x}, z)\downarrow)\right)\right) \\ z & \dots \wedge\, f(\vec{x}, z) = 0 \forall y\, 0 \le y < z \to \left(f(\vec{x}, y)\downarrow \neq 0\right) \end{cases}$$

is called the unbounded minimisation of f.

Reminder: Kleene's normal form theorem

Theorem

For every partial recursive function $h : \mathbb{N}^n \to \mathbb{N}$ there exist primitive recursive functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N}^{n+1} \to \mathbb{N}$ such that:

$$h(x_1, \ldots, x_n) = (f \circ \mu(g))(x_1, \ldots, x_n)$$

μ-recursive/partial recursive $\Rightarrow$ λ-definable

Theorem

Every μ-recursive/partial recursive function is λ-definable.

Proof.

Let $h : \mathbb{N}^{n+1} \to \mathbb{N}$ be partial recursive.

μ-recursive/partial recursive $\Rightarrow$ λ-definable

Theorem

Every μ-recursive/partial recursive function is λ-definable.

Proof.

Let $h : \mathbb{N}^{n+1} \to \mathbb{N}$ be partial recursive.
Then by Kleene's normal form theorem there exist $g : \mathbb{N}^{n+1} \to \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$ such that:

$$h(\vec{x}) = f \circ \mu(g)(\vec{x}) = f(\mu z.[g(\vec{x}, z) = 0])$$

μ-recursive/partial recursive $\Rightarrow$ λ-definable

Theorem

Every μ-recursive/partial recursive function is λ-definable.

Proof.

Let $h : \mathbb{N}^{n+1} \to \mathbb{N}$ be partial recursive.
Then by Kleene's normal form theorem there exist $g : \mathbb{N}^{n+1} \to \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$ such that:

$$h(\vec{x}) = f \circ \mu(g)(\vec{x}) = f(\mu z.[g(\vec{x}, z) = 0])$$

Let M_f and M_g be λ-terms representing f and g, respectively. Let:

$$W := \lambda y.\textbf{if } (\textbf{zero? } M_g x_1 \ldots x_n y) \textbf{ then } (\lambda w.M_f y) \textbf{ else } (\lambda w.w(M_{\mathsf{succ}} y)w)$$

Then the following λ-term M_h represents h:

$$M_h := \lambda x_1 \ldots x_n.W \ulcorner 0 \urcorner W$$

□

A normalizing reduction strategy

Normal order reduction strategy $\xrightarrow{\text{n}}$:
only perform $\rightarrow_\beta$-steps in left-most positions.

A normalizing reduction strategy

Normal order reduction strategy $\xrightarrow{\mathrm{n}}$:
only perform $\rightarrow_{\beta}$-steps in left-most positions.

Theorem

The normal order reduction strategy in is normalizing in λ-calculus, that is:

$$M \rightarrow_{\beta}^{*} N \wedge N \text{ is a normal form } \implies M \xrightarrow{\mathrm{n}}{}^{*} N$$

λ-definable $\Rightarrow$ Turing-computable

Theorem

Every λ-definable function is Turing computable.

λ-definable $\Rightarrow$ Turing-computable

Theorem

Every λ-definable function is Turing computable.

Idea of the Proof.

Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function that is λ-definable. Then there exists a λ-term M_f that represents f.

λ-definable $\Rightarrow$ Turing-computable

Theorem

Every λ-definable function is Turing computable.

Idea of the Proof.

Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function that is λ-definable. Then there exists a λ-term M_f that represents f.

To compute f, one can build a Turing machine M that, for given $m_1, \ldots, m_n \in \mathbb{N}$:

- simulates a normal order rewrite sequence on $M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner$

λ-definable $\Rightarrow$ Turing-computable

Theorem

Every λ-definable function is Turing computable.

Idea of the Proof.

Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function that is λ-definable. Then there exists a λ-term M_f that represents f.

To compute f, one can build a Turing machine M that, for given $m_1, \ldots, m_n \in \mathbb{N}$:

- simulates a normal order rewrite sequence on $M_f \ulcorner m_1 \urcorner \ldots \ulcorner m_n \urcorner$

to obtain the normal form $\ulcorner f(m_1, \ldots, m_n) \urcorner$ □

Summary

Lambda calculus

- λ-calculus
  - syntax
  - reduction rules
- λ-definable functions
- primitive recursive functions are λ-definable
- μ-recursive/partial recursive functions are λ-definable
- λ-definable functions are Turing computable
- Hence: λ-definable = partial recursive = Turing-computable

Suggested reading

- Interaction-Based Models of Computation:
  Chapter 7, The Lambda Calculus of the book:
  - Maribel Fernández [2]: *Models of Computation (An Introduction to Computability Theory)*, Springer-Verlag London, 2009.

Suggested reading

- Interaction-Based Models of Computation:
  Chapter 7, The Lambda Calculus of the book:
  - Maribel Fernández [2]: *Models of Computation (An Introduction to Computability Theory)*, Springer-Verlag London, 2009.
- Post's Correspondence Problem
  - see paper link webpage
- Fractran
  - see paper and video link webpage

Course overview

Monday, July 7 10.30 – 12.30	Tuesday, July 8 10.30 – 12.30	Wednesday, July 9 10.30 – 12.30	Thursday, July 10 10.30 – 12.30	Friday, July 11
intro	*classic models*			*additional models*
Introduction to Computability	**Machine Models**	**Recursive Functions**	**Lambda Calculus**	
computation and decision problems, from logic to computability, overview of models of computation relevance of MoCs	Post Machines, typical features, Turing's analysis of human computers, Turing machines, basic recursion theory	primitive recursive functions, Gödel–Herbrand recursive functions, partial recursive funct's, partial recursive = = Turing-computable, Church's Thesis	λ-terms, β-reduction, λ-definable functions, partial recursive = λ-definable = Turing computable	
	imperative programming	*algebraic programming*	*functional programming*	
				14.30 – 16.30
				Three more Models of Computation
				Post's Correspondence Problem, Interaction-Nets, Fractran comparing computational power

References

Henk Pieter Barendregt.
The Lambda Calculus (Its Syntax and Semantics), volume 103 of *Studies in Logic and the Foundations of Mathematics*.
Elsevier, 1984.

Maribel Fernández.
Models of Computation (An Introduction to Computability Theory).
Springer, Dordrecht Heidelberg London New York, 2009.