Type-Based Synthesis and the Inhabitation Problem

Lecture 1: Typed λ-Calculus and Combinatory Logic

Jakob Rehof
TU Dortmund University, Germany

Warsaw University Open Lectures for PhD Students in Computer Science (PhD open)

University of Warsaw, December 1-2 2017

λ-Calculus

Untyped, pure λ-calculus

- H.P. Barendregt: *The Lambda Calculus. Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, 2nd Edition. Elsevier Science Publishers 1984. [Bar84]
- Hindley and Seldin: *Lambda-Calculus and Combinators, an Introduction*, Cambridge University Press 2008. [HS08]

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.
- An abstraction $\lambda x.M$ can be read as: "the function of x returning $M(x)$".

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.
- An abstraction $\lambda x.M$ can be read as: "the function of x returning $M(x)$".
- Functional expressions are *anonymous*. Mathematicians sometimes use notation like $x \mapsto x + 1$, and that corresponds to $\lambda x.x + 1$.

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.
- An abstraction $\lambda x.M$ can be read as: "the function of x returning $M(x)$".
- Functional expressions are *anonymous*. Mathematicians sometimes use notation like $x \mapsto x + 1$, and that corresponds to $\lambda x.x + 1$.
- In addition to the operation of abstraction there is a (dual) operation of *application*: If M and N are λ-expressions, then the application of M to N, written $(M\ N)$, is also a λ-expression.

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.
- An abstraction $\lambda x.M$ can be read as: "the function of x returning $M(x)$".
- Functional expressions are *anonymous*. Mathematicians sometimes use notation like $x \mapsto x + 1$, and that corresponds to $\lambda x.x + 1$.
- In addition to the operation of abstraction there is a (dual) operation of *application*: If M and N are λ-expressions, then the application of M to N, written $(M\ N)$, is also a λ-expression.
- Computation arises when abstractions are applied to arguments, for example $((\lambda x.x + 1)\ 2)$. An expression of the form $((\lambda x.M)\ N)$ is called a *redex*.

λ-Calculus in One Slide

- λ-calculus is a minimalistic formalism for defining functional expressions and computations with such (a minimalistic functional programming language):
- Assuming computational expressions such as $x + 1$ the operation of λ-*abstraction* constructs the expression $\lambda x.x + 1$.
- An abstraction $\lambda x.M$ can be read as: "the function of x returning $M(x)$".
- Functional expressions are *anonymous*. Mathematicians sometimes use notation like $x \mapsto x + 1$, and that corresponds to $\lambda x.x + 1$.
- In addition to the operation of abstraction there is a (dual) operation of *application*: If M and N are λ-expressions, then the application of M to N, written $(M\ N)$, is also a λ-expression.
- Computation arises when abstractions are applied to arguments, for example $((\lambda x.x + 1)\ 2)$. An expression of the form $((\lambda x.M)\ N)$ is called a *redex*.
- Computation means *substitution* of arguments for the λ-abstracted variable of the function operator in a redex. For example:

$$((\lambda x.x + 1)\ 2) \rightarrow_\beta (x + 1)[x := 2] \equiv 2 + 1$$

λ-terms

Let $\mathcal{V}$ denote a denumerable set of λ-*variables*, $v_1, v_2, v_3, \ldots$.

Let $x, y, z, \ldots$ denote a denumerable set of *metavariables* ranging over $\mathcal{V}$.

Definition 1

The set Λ of λ-*terms*, ranged over by $M, N, P, Q, \ldots$, is defined inductively by:

- $x \in \mathcal{V} \Rightarrow x \in \Lambda$ (variables)
- $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$ (application)
- $M \in \Lambda, x \in \mathcal{V} \Rightarrow (\lambda x M) \in \Lambda$ (abstraction)

λ-terms

Notation 1

We may abbreviate $(((FM_1)M_2)\cdots M_n)$ by

$$FM_1M_2\cdots M_n$$

for $n \geq 0$.

We may abbreviate $(\lambda x_1(\lambda x_2(\cdots(\lambda x_n M)\cdots)))$ by

$$\lambda x_1.\lambda x_2.\cdots\lambda x_n.M$$

for $n \geq 0$.

And further, $\lambda x_1.\lambda x_2.\cdots\lambda x_n.M$ by $\lambda x_1x_2.\cdots x_n.M$.

Definition 2 (Free and bound variables)

The set of *free variables* of M, denoted $\mathsf{FV}(M)$, is defined by induction on M:

$$\begin{array}{lcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(MN) & = & \mathsf{FV}(M) \cup \mathsf{FV}(N) \\ \mathsf{FV}(\lambda x.M) & = & \mathsf{FV}(M) \setminus \{x\} \end{array}$$

A variable $x \in \mathsf{FV}(M)$ is called *bound* in $\lambda x.M$ (by that λ). A term $M \in \Lambda$ is called *closed*, if $\mathsf{FV}(M) = \emptyset$.

Definition 3 (Syntactic identity)

We write $M \equiv N$ to denote that M and N are the same term under renaming of bound variables (α-*conversion*). For example,

$$\begin{array}{lcll} (\lambda x.x)z & \equiv & (\lambda x.x)z & \\ (\lambda x.x)z & \equiv & (\lambda y.y)z & \\ (\lambda x.x)x & \equiv & (\lambda y.y)x & \\ (\lambda x.x)z & \not\equiv & (\lambda x.y)z & \text{(provided } x \not\equiv y\text{)} \end{array}$$

Note that $x \equiv y$ is possible, unless we have stipulated that $x \not\equiv y$.

Definition 4 (Substitution)

Let $M[x := N]$ denote the result of substituting N for the free occurrences of x in M, defined by induction on M:

$$
\begin{array}{lcl}
x[x := N] & \equiv & N \\
y[x := N] & \equiv & y, \quad \text{if } x \not\equiv y \\
(PQ)[x := N] & \equiv & (P[x := N]Q[x := N]) \\
(\lambda y.P)[x := N] & \equiv & \lambda y.P[x := N], \quad \text{if } x \not\equiv y \\
(\lambda x.P)[x := N] & \equiv & \lambda x.P
\end{array}
$$

Lemma 5 (Substitution lemma)

Assume $x \not\equiv y$ *and* $x \notin \mathsf{FV}(P)$. *Then*

$$M[x := N][y := P] \equiv M[y := P][x := N[y := P]]$$

Exercise 1

Prove Lemma 5 by induction on M.

Notion of reduction

Definition 6

A *notion of reduction* on a set $\mathbf{D}$ is a binary relation $\rhd \subseteq \mathbf{D} \times \mathbf{D}$.

Definition 7 (Notion of β-reduction)

The notion of β-*reduction* is the relation $\rhd_\beta \subseteq \Lambda \times \Lambda$ defined by

$$(\lambda x.M)N \ \rhd_\beta \ M[x := N]$$

for all $M, N \in \Lambda$. An expression of the form $(\lambda x.M)N$ is called a *redex*.

Contexts

Contexts C over Λ are expressions of the form

$$C ::= [] \mid (CM) \mid (MC) \mid \lambda x.C$$

We think of $[]$ as a hole and of C as a term with a hole in it.

If C is a context and M a term then $C[M]$ is the term which arises from exchanging $[]$ with M ("hole-filling").

For example, if $C \equiv \lambda x.\lambda y.y[]$, then

$$C[\lambda z.(xz)] \equiv \lambda x.\lambda y.y(\lambda z.(xz))$$

Note that variable capture can happen through hole-filling.

β-reduction

Definition 8

A notion of reduction $\rhd \subseteq \Lambda \times \Lambda$ satisfying

$$\forall C.\ M \ \rhd \ N \Rightarrow C[M] \ \rhd \ C[N]$$

is called a *reduction relation*. The smallest reduction relation containing $\rhd$ is called the *reduction relation induced by* $\rhd$ (sometimes also called the compatible closure of $\rhd$).

Definition 9 (β-reduction)

The relation $\to_\beta$ (β-*reduction*) is the reduction relation induced by $\rhd_\beta$.

If $M \to_\beta N$, then N is called a *reduct* of M.

A *normal form* is a term with no redex.

β-reduction

Definition 10 (Multi-step β-reduction)

The relation $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$.

Definition 11 (β-conversion)

The relation $=_\beta$ (β-*conversion*) is the reflexive symmetric transitive closure of $\rightarrow_\beta$.

If $M =_\beta N$, we say that M and N are β-equivalent.

λ-calculus combinators

A λ-calculus *combinator* is a closed λ-term.

Some interesting combinators are:

$\mathbf{I} \equiv \lambda x.x$, $\mathbf{K} \equiv \lambda xy.x$, $\mathbf{K}^* \equiv \lambda xy.y$, $\mathbf{S} \equiv \lambda xyz.(xz)(yz)$

$\omega \equiv \lambda x.xx$, $\Omega \equiv \omega\omega$

$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, $\Theta \equiv (\lambda x.\lambda y.(y(xxy)))(\lambda x.\lambda y.(y(xxy)))$

Exercise 2

Show that, for all terms F, we have $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$ and $\Theta F \twoheadrightarrow_\beta F(\Theta F)$.

Confluence

Theorem 12 (Church-Rosser, confluence)

For every $M, P_1, P_2 \in \Lambda$, if $M \twoheadrightarrow_\beta P_1$ and $M \twoheadrightarrow_\beta P_2$, then there exists $Q \in \Lambda$ such that $P_1 \twoheadrightarrow_\beta Q$ and $P_2 \twoheadrightarrow_\beta Q$.

Corollary 13 (Uniqueness of normal forms)

If N_1 and N_2 are normal forms such that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then $N_1 \equiv N_2$. If $P =_\beta Q$, then P and Q have the same normal form if any.

Exercise 3

Prove Corollary 13, by using Theorem 12.

Church numerals

For $P, M \in \Lambda$, define $P^n(M)$ by induction on $n \geq 0$:

- $P^0(M) \equiv M$
- $P^{n+1}(M) \equiv P(P^n(M))$

The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by $\mathbf{c}_n \equiv \lambda f.\lambda x.f^n(x)$.

Define $\mathbf{A}_+ \equiv \lambda xypq.xp(ypq)$, $\mathbf{A}_\times \equiv \lambda xyz.x(yz)$, $\mathbf{A}_e \equiv \lambda xy.yx$. Then one can prove:

$\mathbf{A}_+\mathbf{c}_n\mathbf{c}_m =_\beta \mathbf{c}_{n+m}$,

$\mathbf{A}_\times\mathbf{c}_n\mathbf{c}_m =_\beta \mathbf{c}_{n\times m}$,

$\mathbf{A}_e\mathbf{c}_n\mathbf{c}_m =_\beta \mathbf{c}_{n^m}$ $(m > 0)$.

Turing completeness

A numerical function $f : \mathbb{N}^k \to \mathbb{N}$ is called λ-*definable* if there exists a combinator P_f such that

$$P_f \mathbf{c}_{n_1} \dots \mathbf{c}_{n_k} =_\beta \mathbf{c}_{f(n_1,\dots,n_k)}$$

Theorem 14 (Kleene)

Every recursive function is λ-definable.

Typed λ-Calculus

Some general references:

- Barendregt: *Lambda Calculus with Types*, Handbook of Logic in Computer Science, vol. 2, Oxford University Press 1993. [Bar93]
- Hindley and Seldin: *Lambda-Calculus and Combinators, an Introduction*, Cambridge University Press 2008. [HS08]
- Barendregt, Dekkers, Statman: *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press 2013. [BDS13]
- Sørensen and Urzyczyn: *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics, 149, Elsevier 2006. [SU06]
- Nederpelt and Geuvers: *Type Theory and Formal Proof. An Introduction*, Cambridge University Press 2014. [NG14] (higher, dependent and inductive, types).

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.
- We write $\Gamma \vdash M : \tau$ to express that λ-term M has type τ under the type assumptions for the free variables in M given by Γ

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.
- We write $\Gamma \vdash M : \tau$ to express that λ-term M has type τ under the type assumptions for the free variables in M given by Γ
- In many programming languages we write down types for functions like this:

```
int strlen(string x){ M }
```

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.
- We write $\Gamma \vdash M : \tau$ to express that λ-term M has type τ under the type assumptions for the free variables in M given by Γ
- In many programming languages we write down types for functions like this:

```
int strlen(string x){ M }
```

- In type theory we write

$$\vdash M : \texttt{string} \to \texttt{int}$$

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.
- We write $\Gamma \vdash M : \tau$ to express that λ-term M has type τ under the type assumptions for the free variables in M given by Γ
- In many programming languages we write down types for functions like this:

```
int strlen(string x){ M }
```

- In type theory we write

$$\vdash M : \texttt{string} \to \texttt{int}$$

- The type system is a deductive system for deriving ($\vdash$) well-typings of such a form.

Simple types in one slide

- There are atomic types (type constants such as `int`, `bool`, `real` etc.) and type variables α, β, γ etc.
- There are function types $\tau \to \sigma$, whenever τ and σ are types.
- We write $\Gamma \vdash M : \tau$ to express that λ-term M has type τ under the type assumptions for the free variables in M given by Γ
- In many programming languages we write down types for functions like this:

```
int strlen(string x){ M }
```

- In type theory we write

$$\vdash M : \texttt{string} \to \texttt{int}$$

- The type system is a deductive system for deriving ($\vdash$) well-typings of such a form.
- Type assumptions are needed:

$$\{\texttt{strlen} : \texttt{string} \to \texttt{int}, s : \texttt{string}\} \vdash \texttt{strlen(s)} : \texttt{int}$$

Simple types

Definition 15 (Simple types)

Let $\mathbb{V}$ denote a denumerable set of *type variables*, ranged over by metavariables $\alpha, \beta, \gamma, \ldots$, and let b range over a set $\mathbb{B}$ of *type constants*. The set $\mathbb{T}$ of *simple types*, ranged over by $\tau, \sigma, \rho, \ldots$, is defined inductively by:

- $\alpha \in \mathbb{V} \Rightarrow \alpha \in \mathbb{T}$
- $b \in \mathbb{B} \Rightarrow b \in \mathbb{T}$
- $\tau \in \mathbb{T}, \sigma \in \mathbb{T} \Rightarrow \tau \to \sigma \in \mathbb{T}$

Type variables and type constants are referred to as *atoms*. We write $\tau \to \sigma \to \rho$ for $\tau \to (\sigma \to \rho)$.

A *type environment* is a finite set Γ of *type assumptions* of the form $\Gamma = \{(x_1 : \tau_1), \ldots, (x_n : \tau_n)\}$ with $x_i \not\equiv x_j$ for $i \neq j$. We let $\mathsf{Dm}(\Gamma) = \{x \in \mathcal{V} \mid \exists \tau \in \mathbb{T}.\ (x : \tau) \in \Gamma\}$. We write $\Gamma, x : \tau$ for $\Gamma \cup \{(x : \tau)\}$.

Simple typed λ-calculus, $\lambda^{\rightarrow}$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}(\mathsf{var})$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \rightarrow \sigma}(\rightarrow\mathsf{I})$$

$$\frac{\Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}(\rightarrow\mathsf{E})$$

Simple typed λ-calculus, $\lambda^{\rightarrow}$

$$\dfrac{\dfrac{}{\{x:\tau\} \vdash x:\tau}(\mathsf{var})}{\emptyset \vdash \lambda x.x : \tau \rightarrow \tau}(\rightarrow\mathsf{I})$$

Simple typed λ-calculus, $\lambda^{\rightarrow}$

$$\dfrac{\dfrac{}{\{x:\tau\} \vdash x:\tau}(\text{var})}{\emptyset \vdash \lambda x.x : \tau \rightarrow \tau}(\rightarrow \text{I})$$

Since such a derivation can be constructed for arbitrary τ we can see that

$$\emptyset \vdash \lambda x.x : \tau \rightarrow \tau, \quad \text{for all } \tau$$

is a meta-theorem of the system.

Explicit typing

$$\frac{}{\Gamma, x : \tau \vdash^* x : \tau}(\mathsf{var})$$

$$\frac{\Gamma, x : \tau \vdash^* M : \sigma}{\Gamma \vdash^* \lambda x{:}\,\tau.M : \tau \to \sigma}(\to\mathsf{I})$$

$$\frac{\Gamma \vdash^* M : \tau \to \sigma \quad \Gamma \vdash^* N : \tau}{\Gamma \vdash^* MN : \sigma}(\to\mathsf{E})$$

Exercise 4

Show by induction on M: if $\Gamma \vdash^* M : \sigma$ and $\Gamma \vdash^* M : \tau$, then $\sigma \equiv \tau$.

Basis lemma

Let $\Gamma \downarrow_V = \{(x : \tau) \in \Gamma \mid x \in V\}$.

Lemma 16

- *If* $\Gamma \subseteq \Gamma'$ *then* $\Gamma \vdash M : \tau$ *implies* $\Gamma' \vdash M : \tau$,
- *If* $\Gamma \vdash M : \tau$, *then* $\mathsf{FV}(M) \subseteq \mathsf{Dm}(\Gamma)$,
- *If* $\Gamma \vdash M : \tau$, *then* $\Gamma \downarrow_{\mathsf{FV}(M)} \vdash M : \tau$.

Exercise 5

Prove Lemma 16.

Inversion

Lemma 17 (Generation lemma)

Suppose that $\Gamma \vdash M : \tau$.

1. *If* $M \equiv x$, *then* $\Gamma = \Gamma', x : \tau$,
2. *If* $M \equiv \lambda x.N$, *then* $\tau \equiv \rho \to \sigma$ *and* $\Gamma, x : \rho \vdash N : \sigma$,
3. *If* $M \equiv PQ$, *then* $\Gamma \vdash P : \sigma \to \tau$ *and* $\Gamma \vdash Q : \sigma$ *for some* σ.

Proof.

Immediate by inspection of the last rule used in the derivation of $\Gamma \vdash M : \tau$. □

Exercise 6

Call a term M *typable, if* $\Gamma \vdash M : \tau$ *for some* Γ *and* τ.

Show that every subterm of a typable term is typable.

Substitutivity

A *type substitution* is a map $S : \mathbb{V} \to \mathbb{T}$, and it is lifted homomorphically to a map $S : \mathbb{T} \to \mathbb{T}$. Let $S(\Gamma) = \{(x : S(\tau)) \mid (x : \tau) \in \Gamma\}$.

Lemma 18 (Substitution)

1. *If* $\Gamma, x : \tau \vdash M : \sigma$ *and* $\Gamma \vdash N : \tau$, *then* $\Gamma \vdash M[x := N] : \sigma$.
2. *If* $\Gamma \vdash M : \tau$, *then* $S(\Gamma) \vdash M : S(\tau)$, *for any type substitution* S.

Exercise 7

Prove Lemma 18 by induction on derivations.

Subject reduction

Theorem 19 (Subject Reduction)

If $\Gamma \vdash M : \tau$ *and* $M \twoheadrightarrow_\beta N$, *then* $\Gamma \vdash N : \tau$.

Proof.

(Sketch) First prove the statement for the case when M is a redex and N its reduct, using the substitution lemma. Then prove the statement when $M \to_\beta N$ by reduction of a redex R in M, by induction on C where $M \equiv C[R]$. Then prove the statement by induction in the length of the reduction $M \twoheadrightarrow_\beta N$. □

Exercise 8

Complete the proof sketch of Theorem 19.

Strong normalization

Definition 20

A λ-term M is called *strongly normalizing*, if every β-reduction sequence starting from M is finite (i.e., leading to a normal form), and *weakly normalizing*, if there exists some reduction sequence to normal form starting from M.

Theorem 21

Every typable term in $\lambda^{\rightarrow}$ is strongly normalizing.

Proof.

See lecture notes. □

Combinators

Let $\mathcal{C}$ be a set of *combinator symbols*, ranged over by X, Y, Z. The set $\Xi_{\mathcal{C}}$ of *combinatory expressions*, ranged over by F, G, H are defined inductively by:

- $X \in \mathcal{C} \Rightarrow X \in \Xi_{\mathcal{C}}$,
- $x \in \mathcal{V} \Rightarrow x \in \Xi_{\mathcal{C}}$,
- $F, G \in \Xi_{\mathcal{C}} \Rightarrow (FG) \in \Xi_{\mathcal{C}}$

Consider the set $\mathsf{SKI} = \{\mathbf{S}, \mathbf{K}, \mathbf{I}\}$ of combinator symbols (referred to as a combinatory *base*) and define the notion of *weak reduction* $\rhd_w$ on Ξ_{SKI} by setting, for all $X, Y, Z \in \Xi_{\mathsf{SKI}}$:

$$\begin{array}{lll} \mathbf{I}F & \rhd_w & F \\ \mathbf{K}FG & \rhd_w & F \\ \mathbf{S}FGH & \rhd_w & (FH)(GH) \end{array}$$

Let $\rightarrow_w$ and $\twoheadrightarrow_w$ be the reduction relations on Ξ_{SKI} induced by $\rhd_w$, by closing $\rhd_w$ under contexts $C ::= [] \mid (CF) \mid (FC), \ \ (F \in \Xi_{\mathsf{SKI}})$.

Combinatory bases

By choosing different sets $\mathfrak{B} \subseteq \mathcal{C}$ of combinators (such as $\mathfrak{B} = \mathsf{SKI} = \{\mathbf{S}, \mathbf{K}, \mathbf{I}\}$) we can study different combinatory calculi, since in each case we can consider a $\mathfrak{B}$-calculus generated from the combinators in $\mathfrak{B}$. In such cases, we refer to the set $\mathfrak{B}$ as a *combinatory base*.

Exercise 9

Show that the combinator $\mathbf{I}$ *can be coded in terms of* $\mathbf{S}$ *and* $\mathbf{K}$*. Hint: Notice that* $(\mathbf{K}F)(\mathbf{K}F) \twoheadrightarrow_w F$*.*

In other words, the base $SKI = \{\mathbf{S}, \mathbf{K}, \mathbf{I}\}$ *is redundant. Or, in yet other words, the base* $SK = \{\mathbf{S}, \mathbf{K}\}$ *is complete with respect to SKI-calculus. For this reason, one also talks about SK-calculus.*

Combinatory bases

Schönfinkel used, in addition to $\mathbf{S}$ and $\mathbf{K}$, the combinators $\mathbf{B}$ and $\mathbf{C}$ with the definitions

$$\begin{array}{lll} \mathbf{B}FGH & \rhd_B & FGH \\ \mathbf{C}FGH & \rhd_C & FHG \end{array}$$

But they are not strictly needed, for we can take

$$\begin{array}{lll} \mathbf{B} & \equiv & \mathbf{S}(\mathbf{KS})\mathbf{K} \\ \mathbf{C} & \equiv & \mathbf{S}(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KS})\mathbf{K}))\mathbf{S})(\mathbf{KK}) \end{array}$$

Combinatory bases

Exercise 10 (One point basis)

Define the combinator $\mathbf{X}$ *by the rule*

$$(\mathbf{X}F) \rhd_X ((F\mathbf{S})\mathbf{K})$$

Show that $(\mathbf{XX}) \twoheadrightarrow_X \mathbf{SK}(\mathbf{KK})$.

Show that $\mathbf{X}(\mathbf{X}(\mathbf{XX})) \twoheadrightarrow_X \mathbf{K}$.

Show that $\mathbf{X}(\mathbf{X}(\mathbf{X}(\mathbf{XX}))) \twoheadrightarrow_X \mathbf{S}$.

Conclude that $\{\mathbf{X}\}$ *is complete with respect to SKI-calculus.*

Define the map $(\_)_\Lambda : \Xi_{\mathsf{SKI}} \to \Lambda$ by induction on expressions in Ξ_{SKI}:

$$\begin{array}{lcl} (x)_\Lambda & \equiv & x, \text{ for } x \in \mathcal{V} \\ (\mathbf{I})_\Lambda & \equiv & \lambda x.x \\ (\mathbf{K})_\Lambda & \equiv & \lambda xy.x \\ (\mathbf{S})_\Lambda & \equiv & \lambda xyz.(xz)(yz) \\ (FG)_\Lambda & \equiv & (F)_\Lambda (G)_\Lambda \end{array}$$

Proposition 1

If $F \twoheadrightarrow_w G$, then $(F)_\Lambda \twoheadrightarrow_\beta (G)_\Lambda$.

Exercise 11

Prove Proposition 1 by induction on the length of $F \twoheadrightarrow_w G$.

Define, for each $x \in \mathcal{V}$, the "bracket abstraction" map $[x] : \Xi_{\mathsf{SKI}} \to \Xi_{\mathsf{SKI}}$ by induction on expressions in Ξ_{SKI}:

$$\begin{array}{lcl} [x]x & \equiv & \mathbf{I} \\ [x]F & \equiv & \mathbf{K}F, \text{ if } x \notin \mathsf{FV}(F) \\ [x](FG) & \equiv & \mathbf{S}([x]F)([x]G), \text{ otherwise} \end{array}$$

Proposition 2 (Combinatory completeness)

1. $\forall F \in \Xi_{\mathsf{SKI}}.\forall G \in \Xi_{\mathsf{SKI}}.\ ([x]F)G \twoheadrightarrow_w F[x := G]$
2. $\forall F \in \Xi_{\mathsf{SKI}}.\ ([x]F)_\Lambda \twoheadrightarrow_\beta \lambda x.(F)_\Lambda$
3. $\forall x \in \mathcal{V}.\ \forall F \in \Xi_{\mathsf{SKI}}.\ \exists H \in \Xi_{\mathsf{SKI}}.\ \forall G \in \Xi_{\mathsf{SKI}}.\ HG \twoheadrightarrow_w F[x := G]$

Exercise 12

Prove Proposition 2.

Define the map $(\_)_\Xi : \Lambda \to \Xi_{\mathsf{SKI}}$ by induction on λ-terms:

$$\begin{array}{lcl} (x)_\Xi & \equiv & x, \text{ for } x \in \mathcal{V} \\ (MN)_\Xi & \equiv & (M)_\Xi (N)_\Xi \\ (\lambda x.M)_\Xi & \equiv & [x](M)_\Xi \end{array}$$

Proposition 3

For all $M \in \Lambda$, one has $((M)_\Xi)_\Lambda \twoheadrightarrow_\beta M$.

Exercise 13

Prove Proposition 3.

Combinatory logic SKI

$$\frac{}{\Gamma, x : \tau \vdash_{\mathrm{SKI}} x : \tau}(\mathsf{var})$$

$$\frac{}{\Gamma \vdash_{\mathrm{SKI}} \mathbf{I} : \tau \to \tau}(\mathbf{I})$$

$$\frac{}{\Gamma \vdash_{\mathrm{SKI}} \mathbf{K} : \tau \to \sigma \to \tau}(\mathbf{K})$$

$$\frac{}{\Gamma \vdash_{\mathrm{SKI}} \mathbf{S} : (\tau \to \sigma \to \rho) \to (\tau \to \sigma) \to \tau \to \rho}(\mathbf{S})$$

$$\frac{\Gamma \vdash_{\mathrm{SKI}} F : \tau \to \sigma \quad \Gamma \vdash_{\mathrm{SKI}} G : \tau}{\Gamma \vdash_{\mathrm{SKI}} (FG) : \sigma}(\to\mathsf{E})$$

Notice that variables x have fixed, *monomorphic types*, whereas combinators $\mathbf{S}, \mathbf{K}, \mathbf{I}$ have infinitely many types (their types are *schematic* or *polymorphic*). We shall return to this important point in Lecture 5.

Combinatory logic SKI

Lemma 22 (Deduction theorem for SKI)

If $\Gamma, x : \sigma \vdash_{\mathrm{SKI}} F : \tau$, *then* $\Gamma \vdash_{\mathrm{SKI}} [x]F : \sigma \to \tau$.

Proposition 4

1. *If* $\Gamma \vdash_{\mathrm{SKI}} F : \tau$, *then* $\Gamma \vdash_{\Lambda} (F)_{\Lambda} : \tau$.
2. *If* $\Gamma \vdash_{\Lambda} M : \tau$, *then* $\Gamma \vdash_{\mathrm{SKI}} (M)_{\Xi} : \tau$.

Exercise 14

Prove Proposition 4. Hint: The first statement is proven by induction on the derivation of $\Gamma \vdash_{\mathrm{SKI}} F : \tau$. *The second statement is proven by induction on the derivation of* $\Gamma \vdash_{\Lambda} M : \tau$ *using Lemma 22.*

Decision problems

- *Type checking* ($\Gamma \vdash M : \tau$?). Given Γ, M and τ, does $\Gamma \vdash M : \tau$ hold?
- *Typability* ($? \vdash M :?$). Given M, are there Γ and τ such that $\Gamma \vdash M : \tau$?
- *Inhabitation* ($\Gamma \vdash ? : \tau$). Given Γ and τ, does there exist a term ("inhabitant") M such that $\Gamma \vdash M : \tau$?

Type checking and typability for $\lambda^{\rightarrow}$ are linear time solvable.

Inhabitation for $\lambda^{\rightarrow}$ is PSPACE-complete. See Lecture 2.

Inhabitation can be seen as a program synthesis problem:

- Construct *program* M satisfying *specification* τ.


H. P. Barendregt.
The Lambda Calculus. Its Syntax and Semantics.
Studies in Logic and the Foundations of Mathematics, 2nd Edition.
Elsevier Science Publishers, 1984.


H. P. Barendregt.
Lambda Calculus with Types, volume 2.
Oxford University Press, 1993.


H. P. Barendregt, W. Dekkers, and R. Statman.
Lambda Calculus with Types.
Perspectives in Logic, Cambridge University Press, 2013.


J. R. Hindley and J. P. Seldin.
Lambda-calculus and Combinators, an Introduction.
Cambridge University Press, 2008.


R. Nederpelt and H. Geuvers.
Type Theory and Formal Proof. An Introduction.
Cambridge University Press, 2014.


M.H. Sørensen and P. Urzyczyn.
Lectures on the Curry-Howard Isomorphism, volume 149 of *Studies in Logic and the Foundations of Mathematics*.
Elsevier, 2006.