Introduction to Lambda Calculus

Gert Smolka, Saarland University

November 29, 2019

We give an informal introduction to untyped λ-calculus assuming that the reader is familiar with abstract reduction systems. We show how λ-calculus can encode inductive data types and functional recursion. We also cover simplified versions of the λ-calculus known as call-by-value calculus and the SK-Calculus.

1 Introduction

Untyped lambda calculus is a basic logical system everyone should know. It provides syntactic descriptions of computable functions and can express all computable functions. Untyped lambda calculus is a minimal system with only three primitives: variables, application, and functional abstraction. Untyped lambda calculus was invented by Alonzo Church in the 1930s.

Untyped lambda calculus covers basic aspects of programming languages and logical languages with bound variables. There are various refinements of untyped lambda calculus, many of them involving types.

Untyped lambda calculus models a class of functions operating on functions. There are no other objects but functions. Functions take functions as arguments and return functions as results. No other objects are needed since data objects like numbers and pairs can be represented as functions in the untyped lambda calculus.

Untyped lambda calculus is a model of computation that appeared before Turing machines. Turing himself showed that computability formalized by Turing machines agrees with computability formalized by untyped lambda calculus.

Coq encompasses a typed version of the lambda calculus. Familiarity with Coq is very helpful in understanding the untyped lambda calculus. W e assume that the reader is familiar with Coq.

Untyped lambda calculus is the basic theoretical model for the study of syntactic expressions with bound variables and substitution. Expressions with bound variables and substitution are part of every programming language and are used informally in all of mathematics; for instance, $\{\, n \in \mathbf{N} \mid n^2 < 100n \,\}$ is an expression describing a set using a bound variable n. As it turns out, the formalization of syntactic expressions with bound variables and substitution takes considerable effort.

A textbook introducing untyped lambda calculus is Hindley and Seldin [6]. An advanced presentation of untyped lambda calculus is Barendregt [1].

2 Terms

Untyped lambda calculus comes with syntactic objects called **terms**. Terms can be described with the following grammar:

$$s,t,u,v \;::=\; x \mid st \mid \lambda x.s \qquad (x \in \mathbb{N})$$

We speak of **variables**, **applications**, and **abstractions**. An abstraction $\lambda x.s$ describes a function taking an argument x. Within $\lambda x.s$, x is a **bound variable** providing a reference to the argument of the function described. For instance, the term $\lambda x.x$ describes the identity function that simply returns its argument.

Bound variables may be understood as local variables that are only visible within the abstraction in which they are introduced. We speak of the **scope** of a bound variable and say that λ is a **variable binder**. A variable x is called **free in a term s** if it appears in s at a position that is not in the scope of a binder λx. A term is **open** if some variable occurs free in it, and **closed** if no variable occurs free in it. For instance, $\lambda x.x$ is a closed term and $\lambda x.(fx)y$ is an open term with two free variables f and y.

Closed terms are called **combinators**, and closed abstractions are called **procedures**. Combinators are self-contained descriptions of functions. Open terms are partial descriptions of functions. Partial terms can be refined into closed terms by replacing their free variables with closed terms.

We adopt common notational conventions for terms:

$$\begin{aligned} stu &\rightsquigarrow (st)u \\ \lambda xy.s &\rightsquigarrow \lambda x.\lambda y.s \\ \lambda x.st &\rightsquigarrow \lambda x.(st) \end{aligned}$$

Here is a list of **prominent combinators** we will use in the following:

$$\begin{aligned} I &:= \lambda x.x & \omega &:= \lambda x.xx \\ K &:= \lambda xy.x & \Omega &:= \omega\omega \\ S &:= \lambda fgx.fx(gx) & B &:= \lambda fgx.f(gx) \\ & & C &:= \lambda fxy.fyx \end{aligned}$$

The combinator B can be seen as a composition operator. We may write $s \circ t$ for the term Bst.

Technically, the functions of the lambda calculus take only one argument. Since the result of a function application is again a function, we can write arbitrarily long application chains $st_1 \dots t_n$. Informally, it is often convenient to speak of functions with several arguments. For instance, we may speak of the combinator S as a function with three arguments.

We assume that two terms are identical if their presentations are equal up to consistent renaming of bound variables. We refer to this assumption as **α-assumption**. For instance, $\lambda x.x$ and $\lambda y.y$ are assumed to be identical terms even if x and y are different variables. Consistent renaming of bound variables is known as **α-renaming**.

We distinguish between **pre-terms** and proper terms, where pre-terms are not affected by the α-assumption. Pre-terms can be formalized with an inductive type realizing the grammar given above, and proper terms may be understood as equivalence classes of pre-terms. The equivalence relation relating pre-terms that are equal up to α-renaming is known as **α-equivalence**. Given this setup, two pre-terms represent the same term if and only if they are α-equivalent.

In this demo we will not give a formal definition of terms satisfying the α-assumption. Giving such a definition and establishing its basic properties takes considerable effort. Interestingly, the higher-level properties of λ-calculus can be argued informally without knowing the formal definition of terms.

Exercise 1 Formalize pre-terms with an inductive type in Coq. Moreover, formalize the notion of free variables of pre-terms with an inductive predicate *free* x s.

3 Reduction and Equivalence

There is a single reduction rule for terms

$$(\lambda x.s)\,t \succ s^x_t$$

known as **β-reduction**. The notation s^x_t stands for the term that is obtained from s by replacing all free occurrences of the variable x with the term t. One speaks of a **substitution**. Terms of the form $(\lambda x.s)t$ are called **β-redexes**. Here are examples for reductions:

$$
\begin{gathered}
I\omega \succ \omega \\
KI\omega \succ (\lambda x.\, I)\,\omega \succ I \\
\omega\omega \succ \omega\omega \\
KK(\omega\omega) \succ (\lambda x.\, K)\,(\omega\omega) \succ K \\
K(\omega\omega) \succ \lambda x.\omega\omega \succ \lambda x.\omega\omega
\end{gathered}
$$

Note the following:

- The terms $\omega\omega$ and $\lambda x.\omega\omega$ have no normal form.
- The term $I\omega$ is strongly normalizing.
- The term $KK(\omega\omega)$ is normalizing but not strongly normalizing.

The **reduction relation $s \succ t$** is obtained from β-reduction by allowing β-reduction at every subterm position. Thus no term containing a subterm that is not strongly normalizing is strongly normalizing. An inductive definition of the reduction relation $s \succ t$ looks as follows:

$$
\frac{}{(\lambda x.s)t \succ s^x_t} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'} \qquad \frac{s \succ s'}{\lambda x.s \succ \lambda x.s'}
$$

We write $s \equiv t$ for the equivalence closure of the reduction relation $s \succ t$. A main result about the λ-calculus says that the reduction relation is confluent.[1] Thus we can use equivalence $s \equiv t$ to reason about evaluation and more generally about computation in the λ-calculus.

A binary relation R on terms is **compatible** (with the term structure) if it satisfies the following rules:

$$
\frac{Rss'}{R(st)(s't)} \qquad \frac{Rtt'}{R(st)(st')} \qquad \frac{Rss'}{R(\lambda x.s)(\lambda x.s')}
$$

The reduction relation $s \succ t$ is compatible by definition (it is defined as the compatibility closure of top-level β-reduction). Moreover, the equivalence relation $s \equiv t$ is compatible since it is defined as $\succ^{\leftrightarrow *}$ and the closure operators $\leftrightarrow$ and $*$ both preserve compatibility.

Fact 2 Reduction $s \succ t$ is confluent. Thus:

1. $s \equiv t \to \text{normal } t \to s \rhd t$.
2. $s \equiv t \to \text{normal } s \to \text{normal } t \to s = t$.

[1] The confluence result is often referred to as *Church-Rosser theorem*, a name honoring Church and Rosser who proved it first.

3. $s \equiv t \to s \rhd u \leftrightarrow t \rhd u$.

Proof Claims 1-3 are abstract consequences of the Church-Rosser property, and the Church-Rosser property is an abstract consequence of confluence. We do not give a confluence proof here. ∎

We now formulate our method of choice for proving that two terms are not equivalent.

Corollary 3 (Disequivalence) $s \not\equiv t$ provided there are terms $u_1, \dots, u_n$ and normal terms $v \neq w$ such that $su_1 \dots u_n \rhd v$ and $tu_1 \dots u_n \rhd w$. We refer to $u_1, \dots, u_n$ as **separating arguments** for s and t..

Proof Suppose $s \equiv t$. Then $su_1 \dots u_n \equiv tu_1 \dots u_n$ and thus $v \equiv w$. Since v and w are normal, we have $v = w$. Contradiction. ∎

Fact 4 If $s \succ t$ and s is closed, then t is closed.

We remark that reduction and equivalence are stable under substitution. We say that a binary relation R on terms is **stable under substitution** if $Rst \to R(s^x_u)(t^x_u)$ for all s, t, x, and u.

Fact 5 $s \succ t$, $s \succ^* t$, and $s \equiv t$ are stable under substitution.

Proof Stability of $s \succ^* t$ and $s \equiv t$ follow from stability of $s \succ t$. We do not give a stability proof for $s \succ t$ here. ∎

Exercise 6 Show the following:

a) $K \not\equiv I$.

b) $s \equiv t \to sts \equiv tst$.

c) Give a non-normalizing term D where each reduction step increases the size, that is, if $D \succ^* s \succ t$, then the term t is larger than the term s.

4 Substitution Laws

The following examples explain the substitution operation s^x_t. We assume that f, x, y, g, and z are distinct variables.

$$\begin{aligned}
(fxy)^x_y &= fyy \\
(fxyx)^x_z &= fzyz \\
((\lambda x.x)x)^x_z &= (\lambda x.x)z \\
(\lambda xy.fxy)^f_g &= \lambda xy.gxy \\
(\lambda xy.fxy)^f_{gz} &= \lambda xy.gzxy \\
(\lambda xy.fxy)^f_{gx} &= (\lambda zy.fzy)^f_{gx} = \lambda zy.gxzy
\end{aligned}$$

Note that only free occurrences of x are affected by a substitution s^x_t. Also note the last example, which shows that for a substitution s^x_t we may have to rename bound variables of s to avoid **capturing** of free variables of t. Capturing cannot occur if t is closed. If a free occurrence of x in s is in the scope of a bound variable z and z occurs free in t, the bound variable z has to be renamed in s.

Here are laws that must be satisfied by the substitution operation:

$$\begin{array}{rcll}
x^y_u & = & x & \text{if } x \neq y \\
y^y_u & = & u & \\
st^y_u & = & s^y_u\, t^y_u & \\
(\lambda x.s)^y_u & = & \lambda x.s^y_u & \text{if } x \neq y \text{ and } x \text{ not free in } u \\
(\lambda y.s)^y_u & = & \lambda y.s & \\
\lambda x.s & = & \lambda y.\, s^x_y & \text{if } y \text{ not free in } s \\
s^y_u & = & s & \text{if } y \text{ not free in } s \\
s^x_x & = & s &
\end{array}$$

We will refer to these laws as **substitution laws**.

5 Scott Encoding of Numbers

To encode numbers, we need encodings of the constructors 0 and S and of match on numbers. The basic idea of the Scott encoding is to encode numbers as functions that act as matches for the numbers they encode. It in fact suffices to find combinators satisfying the equivalences

$$\mathsf{zero}\ uv \equiv u$$
$$\mathsf{succ}\ suv \equiv vs$$

for all terms s, u, and v. The two equivalence are enough to show that the constructors zero and succ are disjoint and that succ is injective. The equivalences suggest the following definitions of zero and succ:

$$\begin{array}{rcl}
\mathsf{zero} & := & \lambda ab.a \\
\mathsf{succ} & := & \lambda xab.bx
\end{array}$$

We define **iterated application** $s^n t$ for terms as follows:

$$s^0 t := t \qquad s^{\mathsf{S}n} t := s(s^n t)$$

We take the combinators $\mathsf{succ}^n\,\mathsf{zero}$ as representations of numbers. Since these combinators are strongly normalizing, we can represent every number n uniquely with a normal combinator $\overline{n}$. We have:

$$\begin{array}{rcll} \overline{0} & = & \lambda ab.a & = \mathsf{zero} \\ \overline{1} & = & \lambda ab.\, b\,\overline{0} & \equiv \mathsf{succ}\ \mathsf{zero} \\ \overline{2} & = & \lambda ab.\, b\,\overline{1} & \equiv \mathsf{succ}^2\ \mathsf{zero} \\ \overline{Sn} & = & \lambda ab.\, b\,\overline{n} & \equiv \mathsf{succ}^{Sn}\ \mathsf{zero} \end{array}$$

Exercise 7 Let zero and succ be combinators satisfying the two characteristic equivalences given above. Show disjointness and injectivity of zero and succ:

a) $\forall s.\ \neg(\mathsf{zero} \equiv \mathsf{succ}\ s)$.

b) $\forall st.\ \ \mathsf{succ}\ s \equiv \mathsf{succ}\ t\ \ \rightarrow\ \ s \equiv t$.

Exercise 8 Represent pairs with a normal combinator pair. Give the characteristic equivalence for pair and verify that your definition satisfies it. Define the projections π_1 and π_2 and show their correctness.

Exercise 9 Represent the booleans with two normal combinators true and false. Give the characteristic equivalences for true and false and verify that your definitions satisfies them.

6 Recursive Functions as Fixed Points

For addition we need a combinator add satisfying the following equivalences for all terms s and t:

$$\begin{aligned} \mathsf{add}\ \mathsf{zero}\ t &\equiv t \\ \mathsf{add}\,(\mathsf{succ}\ s)\ t &\equiv \mathsf{succ}\,(\mathsf{add}\ st) \end{aligned}$$

The definition of add can be carried out following a fixed scheme using a **fixed point combinator** R satisfying the equivalence

$$R\,s \equiv s\,(R\,s)$$

for all terms s. Note that the equivalence says that R maps every function s to a fixed point of s.[2] We postpone the definition of R and first give the definition of add:

$$\begin{array}{rcl} \mathsf{Add} & := & \lambda fxy.\ xy(\lambda x'.\ \mathsf{succ}\,(fx'y)) \\ \mathsf{add} & := & R\ \mathsf{Add} \end{array}$$

[2] A fixed point of a function $f : X \to X$ is an x such that $fx = x$.

Verifying that the combinator add satisfies the equivalences given above is straightforward. From the addition example we can see that the fixed point combinator R can express arbitrary recursion. We refer to Add as *unfolding function* for add.[3] Note that the fixed point combinator maps the unfolding function to a fixed point of the unfolding function.

For the definition of R so-called *self-application* is essential. Here is our definition of R:

$$\begin{aligned} C &:= \lambda f g.\, g(f f g) \\ R &:= C C \end{aligned}$$

Verifying $R\,s \succ^2 s(R\,s)$ for all terms s is straightforward. Think of C as *copy function* and of f as argument variable for the copy function. The argument variable g represents the unfolding function.

Note that the combinator R is not normalising. We have $R \succ \lambda g.g(Rg)$ and this the only reduction that applies to R.

Hindley and Seldin [6] say that the fixed point combinator R was invented by Alan Turing in 1937.

Exercise 10 (Correctness of add) Prove

$$\mathsf{add}\,(\mathsf{succ}^m\,\mathsf{zero})\,(\mathsf{succ}^n\,\mathsf{zero}) \;\equiv\; (\mathsf{succ}^{m+n}\,\mathsf{zero})$$

using the equivalences for add, succ, and zero. Note that the proof is purely equational and does not involve substitution of terms.

Exercise 11 (Killer) Give a combinator K such that $Ks \succ^* K$.

Exercise 12 (Procrastinator) Give a combinator P such that $Pst \succ^* Pts$.

Exercise 13 (Lists) Represent lists with two normal combinators nil and cons. Give the characteristic equivalences for nil and cons and verify that they are satisfied with your definition. Give a combinator append that appends two lists. Give the characteristic equivalences for append and verify that they are satisfied by your definition.

7 Call-By-Value Lambda Calculus

As a computational system, full λ-calculus is an overkill. In particular, reduction within abstractions is not needed for functional computation. If we just

[3] Unfolding functions are also called functionals in the literature.

disallow reduction below abstractions, we obtain a non-confluent system. For instance, $K(II) \succ^* \lambda x.I$ and $K(II) \succ^* \lambda x.II$. The problem goes away if we restrict β-reduction such that the argument term must be an abstraction.

We define **call-by-value reduction $s \succ_v t$** as follows:

$$\frac{\text{abstraction } t}{(\lambda x.s)t \succ_v s^x_t} \qquad \frac{s \succ_v s'}{st \succ_v s't} \qquad \frac{t \succ_v t'}{st \succ_v st'}$$

Thus β-redexes $(\lambda x.s)t$ can only be reduced if t is an abstraction or the reduction takes place within t. Furthermore, reduction is not possible within abstractions $\lambda x.s$. Note that our definitions ensure that call-by-value reduction is subsumed by ordinary reduction (i.e., $\succ_v \subseteq \succ$).

Call-by-value reduction is confluent. In fact it satisfies a stronger confluence property known as uniform confluence. Uniform confluence ensures that every normalizing term is also strongly normalizing, and that all reductions $s \succ^* t$ where t is normal have the same length.

Call-by-value equivalence $\equiv_v$ is defined as before as

$$s \equiv_v t \;:=\; (s \overset{\leftrightarrow}{\succ}{}_v^{*}\; t)$$

Since call-by-value reduction is confluent, the Church-Rosser property and the resulting properties (as stated by Fact 2) hold for call-by-value equivalence.

As it comes to computation, it turns out that call-by-value reduction of closed terms suffices. The representation of inductive datatypes stays unchanged. The encoding of recursion needs to be changed, however. For an abstraction s we still have $Rs \succ_v^2 s(Rs)$, but this means that no term $Rst_1 \dots t_n$ is call-by-value normalizing. The problem can be fixed with a function ρ from terms to terms satisfying

$$(\rho s)t \succ_v^3 s(\rho s)t$$
$$\rho s \text{ is a procedure}$$

for all procedures s and t. Such a **recursion operator ρ** can be defined as

$$\begin{aligned} C &:= \lambda fg.g(\lambda x.ffgx) \\ \rho s &:= \lambda x.CCsx \end{aligned}$$

The verification of the reduction $(\rho s)t \succ_v^3 s(\rho s)t$ for procedures s and t proceeds as follows:

$$(\rho s)t \;\succ_v\; CCst \;\succ_v\; (\lambda g.g(\lambda x.CCgx))st \;\succ_v\; s(\lambda x.CCsx)t \;=\; s(\rho s)t$$

Using ρ, the addition function can be defined as

$$\mathsf{add} \;:=\; \rho(\mathsf{Add})$$

where the unfolding function Add remains unchanged. The characteristic equivalences for addition also hold for call-by-value equivalence.

As a computational system, the call-by-value calculus has the advantage that abstractions are always irreducible. This way the addition function can be represented as a normal term, which is not the case in the full lambda calculus.

All computable functions on inductive data types (e.g., numbers) can be defined in the call-by-value λ-calculus. In fact, the call-by-value λ-calculus can serve as a model of computation for which the usual undecidability results can be shown [4].

8 Translation to SK-Terms

We return to the full λ-calculus. It turns out that every term can be transformed into an equivalent term using λ-abstractions only within the combinators

$$\begin{aligned} S &:= \lambda xyz.xz(yz) \\ K &:= \lambda xy.x \end{aligned}$$

We call such terms SK-terms. Formally, we define **SK-terms** inductively:

1. Variables are SK-terms.
2. K and S are SK-terms.
3. st is an SK-term if s and t are SK-terms.

We will make use of the SK-term

$$I := SKK$$

The reuse of the name I is motivated by the equivalence $SKK \equiv \lambda x.x$.

We show how general terms can be transformed into SK-terms. For this we need a method that translates a term $\lambda x.s$ where s is an SK-term into an SK-term. Given this method, we can eliminate all abstractions not appearing within the combinators S and K following a bottom up strategy. A single elimination step can be performed by rewriting with the following equivalences from left to right:

$$\begin{aligned} \lambda x.x &\equiv I \\ \lambda x.s &\equiv Ks && \text{if } x \text{ not free in } s \\ \lambda x.st &\equiv S(\lambda x.s)(\lambda x.t) \end{aligned}$$

The equivalences are enough to translate a term $\lambda x.s$ where s is an SK-term to an equivalent SK-term. Here are examples:

$$\begin{aligned} \lambda x.xx &\equiv S(\lambda x.x)(\lambda x.x) \equiv SII \\ \lambda xy.xx &\equiv \lambda x.K(xx) \equiv S(KK)(\lambda x.xx) \equiv S(KK)(SII) \end{aligned}$$

Theorem 14 Every term s can be transformed into an equivalent SK-term t such that every free variable of t is a free variable of s.

Exercise 15 (Single Combinator Base) We define $X := \lambda x.xSK$ and $I := \lambda x.x$. Verify the following:

a) $XX \succ^* I$.

b) $X(XX) \succ^* SK$.

c) $X(X(XX)) \succ^* K$.

d) $X(X(X(XX))) \succ^* S$.

e) Every combinator is equivalent to a term just obtained with X and application.

One says that X is a single combinator base for the lambda calculus. See Goldberg [5] for more about single combinator bases.

9 Weak Reduction

We have

$$\begin{aligned} Kst &\succ^* s \\ Sstu &\succ^* su(tu) \end{aligned}$$

The motivates the definition of two **weak reduction rules**

$$\begin{aligned} Kst &\succ_w s \\ Sstu &\succ_w su(tu) \end{aligned}$$

We have

$$Is = SKKs \succ_w Ks(Ks) \succ_w s$$

We also have that full reduction subsumes weak reduction:

Fact 16 $s \succ_w^* t \rightarrow s \succ^* t$.

It turns out that Scott encodings and recursion can be modelled with just SK-terms and weak reduction. In fact, there are SK-terms zero, succ, and copy satisfying the following weak reductions:

$$\begin{aligned} \mathsf{zero}\,uv &\succ_w^* u \\ \mathsf{succ}\,suv &\succ_w^* vs \\ \mathsf{copy}\,st &\succ_w^* t(sst) \end{aligned}$$

For zero we use

$$\textsf{zero} := K$$

For succ and copy we rely on the outlined translation method. To end up with readable SK-terms, we fix two further SK-terms

$$\begin{aligned} B &:= S(KS)K \\ C &:= S(S(KB)S)(KK) \end{aligned}$$

which satisfy

$$\begin{aligned} Bstu &\succ_w^* s(tu) \\ Cstu &\succ_w^* sut \end{aligned}$$

From the reductions we can see that B and C act as special versions of S. We can now optimize the translation to SK-terms with the following equivalences:

$$\begin{aligned} \lambda x.st &\equiv Bs(\lambda x.t) && \text{if } x \text{ not free in } s \\ \lambda x.st &\equiv C(\lambda x.s)t && \text{if } x \text{ not free in } t \end{aligned}$$

Using these additional equivalences, we translate the terms for succ and copy in the full λ-calculus as follows:

$$\begin{aligned} \lambda xyz.zx &\equiv \lambda xy.CIx \equiv \lambda x.K(CIx) \equiv BK(B(CI)I) \\ \lambda fg.g(ffg) &\equiv \lambda f.SI(B(ff)I) \equiv B(SI)(C(\lambda f.B(ff))I) \equiv B(SI)(C(BB(SII))I) \end{aligned}$$

Following this translation, we define

$$\begin{aligned} \textsf{succ} &:= BK(B(CI)I) \\ \textsf{copy} &:= B(SI)(C(BB(SII))I) \end{aligned}$$

We can now verify the required weak reductions

$$\begin{aligned} \textsf{succ}\, suv &\succ_w^* K(CIs)uv \succ_w^* CIsv \succ_w^* Ivs \succ_w^* vs \\ \textsf{copy}\, st &\succ_w^* SI(B(ss)I)t \succ_w^* t(sst) \end{aligned}$$

We have now discovered a computational system known as **SK-calculus** that is simpler than the λ-calculus but still Turing complete. The terms of the SK-calculus

$$s, t, u, v ::= S \mid K \mid st$$

are obtained with two constants S and K and with binary application. There are no variables and no abstractions. Reduction in the SK-calculus is defined with two reduction rules

$$\begin{aligned} Kst &\succ_w s \\ Sstu &\succ_w su(tu) \end{aligned}$$

One can show that reduction in the SK-calculus is confluent. As it comes to the structure of reduction SK-calculus is similar to λ calculus. One may consider a call-by-value version of SK-calculus, which is similar to call-by-value λ-calculus, and in particular is uniformly confluent.

SK-calculus is much simpler than λ-calculus since it comes without variables and does not require substitution. Programming in the SK-calculus is feasible provided one first programs in the λ-calculus and then compiles down to the SK-calculus.

The theory of SK-like systems is know as combinatory logic [6]. The initial ideas for S and K are from Schönfinkel [7]. Combinatory logic and λ-calculus were studied extensively by Haskell Curry and his students [2, 3].

Exercise 17 Give terms of the SK-calculus as follows:

a) A term that doesn't have a normal form.

b) A normalising term that is not strongly normalising.

c) A term K such that $Ks \succ_w^* K$.

d) A term P such that $Pst \succ_w^* Pts$.

Exercise 18 Show that SK-calculus is not uniformly confluent.

Exercise 19 Let zero and succ be the terms of the SK-calculus defined above. Show disjointness and injectivity of zero and succ in the SK-calculus:

a) $\forall s.\ \neg(\mathsf{zero} \equiv \mathsf{succ}\ s)$.

b) $\forall st.\ \mathsf{succ}\ s \equiv \mathsf{succ}\ t\ \rightarrow\ s \equiv t$.

10 Eta Law

We may take the view that a closed term s describes the same function as the term $\lambda x.sx$. We can formalize this view by defining an equivalence relation on terms that realizes the **η-law**

$$\lambda x.sx \equiv s \qquad \text{if } x \text{ is not free in } s$$

in addition to the β-law. We speak of **$\beta\eta$-equivalence** and write $s \equiv_{\beta\eta} t$. We can also have **η-reduction**

$$\lambda x.sx \succ s \qquad \text{if } x \text{ is not free in } s$$

in addition β-reduction. We speak of **$\beta\eta$-reduction** and write $s \succ_{\beta\eta} t$. The Church-Rosser theorem remains true for untyped λ-calculus with $\beta\eta$-equivalence and $\beta\eta$-reduction.

Coq's convertibility relation accommodates both β and η. In fact, in Coq two terms are definitionally equal if they are $\beta\eta$-equivalent. While β-reduction is explicit in Coq, η-equivalence is implicit (in the same way α-equivalence is implicit). Coq's type discipline ensures that β-reduction always terminates. In particular, Coq's type discipline does not admit self-application of functions as in $\omega = \lambda x.xx$.

Adding the η-law has the consequence that $\overline{1} \equiv I$ since $\overline{1} = \lambda fa.fa \succ_\eta \lambda f.f = I$. Thus $\overline{1}$ is not $\beta\eta$-normal. However, all other Church numerals are $\beta\eta$-normal and different from the the $\beta\eta$-normal form of $\overline{1}$ (which is I). Thus the Church numerals for two different numbers are not $\beta\eta$-equivalent.

It turns out that $\beta\eta$-equivalence is the coarsest equivalence we can have in untyped λ-calculus. This result is a consequence of Böhm's theorem.

Theorem 20 (Böhm 1968) Let s and t be different $\beta\eta$-normal combinators. Then there exist combinators $u_1, \dots, u_n$ such that

$$\begin{aligned} su_1 \dots u_n xy &\equiv_{\beta\eta} x \\ tu_1 \dots u_n xy &\equiv_{\beta\eta} y \end{aligned}$$

for all variables x and y.

Corollary 21 Let $\approx$ be a nontrivial equivalence relation on terms sucht that $s \approx t$ whenever $s \equiv_{\beta\eta} t$. Then $\approx$ and $\equiv_{\beta\eta}$ agree on $\beta\eta$-normal terms.

More about Böhm's theorem can be found in Hindley and Seldin [6].

Exercise 22 Prove $BCC \equiv I$ under $\beta\eta$-equivalence.

Exercise 23 Prove that the η-law follows from $\lambda x.fx \equiv f$ provided $f \neq x$.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984. Revised Edition.

[2] Haskell B. Curry and Robert Feys. *Combinatory Logic, Volume I.* North-Holland Publishing Company, 1958.

[3] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II.* North-Holland Publishing Company, 1972.

[4] Yannick Forster and Gert Smolka. Call-by-value lambda calculus as a model of computation in Coq. *Automated Reasoning*, 63(2):393–413, 2019.

[5] Mayer Goldberg. A construction of one-point bases in extended lambda calculi. *Information processing letters*, 89(6):281–286, 2004.

[6] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an Introduction.* Cambridge University Press, 2008.

[7] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.