Untyped Lambda Calculus
Lecture Notes

Gert Smolka
Saarland University

December 14, 2015

These notes are written for readers familiar with constructive type theory and the proof assistant Coq. We expect that the reader has seen in Coq a formalization of propositional logic with natural deduction.

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. We 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].

We start with a informal presentation of the untyped lambda calculus and discuss important ideas and results. Formal definitions and proofs will appear latter.

2 Terms

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

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

We speak of **variables**, **applications**, and **abstractions**. An abstraction $\lambda x.s$ describes a function taking an argument x. Technically, 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 also called **combinators**. 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** for future reference:

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

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 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 language, two pre-terms represent the same term if and only they are α-equivalent.

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

3 De Bruijn Representation of Terms

De Bruijn [5] came up with a simple inductive representation of terms avoiding bound variables and α-renaming. De Bruijn terms are obtained with the following inductive definition:

$$s, t \ ::= \ n \mid st \mid \lambda s \qquad (n \in \mathbb{N})$$

With De Bruijn's term representation binders do not introduce bound variables. Reference to a binder is established with an **index** n saying how many binders must be skipped on the path to the binder. Here are examples of De Bruijn terms

representing closed terms:

$$\begin{array}{rcl} \lambda f x y.\, f x y & \rightsquigarrow & \lambda\,\lambda\,\lambda\,2\,1\,0 \\ \lambda f.\, f(\lambda x.\, f x(\lambda y.\, f x y)) & \rightsquigarrow & \lambda\,0\,(\lambda\,1\,0\,(\lambda\,2\,1\,0)) \end{array}$$

Here is an example of a De Bruijn term with free variables:

$$(\lambda x y.\, f y x)x \quad \rightsquigarrow \quad (\lambda\,\lambda\,(f{+}2)\,0\,1)\,x$$

An index n that is below d binders refers to one of the binders if $n < d$, and to the free variable $x = n - d$ if $n \geq d$.

Exercise 2 Represent in Coq pre-terms and De Bruijn terms with inductive types. Write a function from pre-terms to De Bruijn terms such that two pre-terms are α-equivalent if and only if they are mapped to the same De Bruijn term.

4 Beta Redexes and Substitution

A **β-redex** is a term of the form $(\lambda x.s)t$. A β-redex $(\lambda x.s)t$ may be simplified

$$(\lambda x.s)t \quad \rightsquigarrow \quad s^x_t$$

to the term s^x_t, where s^x_t is obtained from the term s by substituting the term t for the variable x. The simplification of β-redexes captures the idea that $\lambda x.s$ describes a function and that $(\lambda x.s)t$ describes the application of the function to a term t.

We need to clarify the notion of **substitution** before we say more about the simplification of terms. We say that s^x_t is obtained from s by **substituting t for x**, or by **substituting x with s**. The following examples clarify what we mean by this. We assume that f, x, y, g, and z are distinct variables.

$$\begin{array}{rcl} (f x y)^x_y & = & f y y \\ (f x y x)^x_z & = & f z y z \\ ((\lambda x.x)x)^x_z & = & (\lambda x.x)z \\ (\lambda x y.\, f x y)^f_g & = & \lambda x y.\, g x y \\ (\lambda x y.\, f x y)^f_{g z} & = & \lambda x y.\, g z x y \\ (\lambda x y.\, f x y)^f_{g x} & = & (\lambda z y.\, f z y)^f_{g x} \;=\; \lambda z y.\, g x z y \end{array}$$

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

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

Figure 1: Substitution laws

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.

We postpone a formal definition of substitution based on De Bruijn terms. As it comes to our informal view of terms, we can say that s^x_t is a total operation taking three arguments and satisfying the **substitution laws** shown in Figure 1. The last two substitution laws follow by induction on s from the other laws.

Fact 3 (Free Variables) If z is free s^x_t, then z is free in either s or in t.

5 Beta Equivalence

We capture the idea that $(\lambda x.s)t$ and s^x_t describe the same object with a equivalence relation $s \equiv t$ on terms called **β-equivalence**. We define β-equivalence as an inductive predicate with the rules shown in Figure 2. From the inductive definition it is clear that β-equivalence is the least equivalence relation on terms (last three rules) that respects the **β-law** (first rule) and the term structure (second and third rule). Informally, we can say that β-equivalence is the abstract equality that results from assuming the β-law, where assuming the β-law means that we can rewrite every subterm with the β-law in either direction (known as **contraction** and **expansion**).

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

$$\frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \qquad t \equiv u}{s \equiv u}$$

Figure 2: Inductive definition of β-equivalence

Example 4 We have $SKK \equiv I$. This can be verified with the following derivation.

$$\begin{aligned} SKK &= (\lambda fgx.fx(gx))KK \\ &\equiv \lambda x.Kx(Kx) \\ &= \lambda x.(\lambda xy.x)x(Kx) \\ &\equiv \lambda x.(\lambda y.x)(Kx) \\ &\equiv \lambda x.x \\ &= I \end{aligned}$$

Closed terms are descriptions of functions such that β-equivalent closed terms describe the same function. It turns out that equivalences between open terms are also interesting since all there substitution instances are valid β-equivalence.

Fact 5 (Substitutivity) If $s \equiv t$, then $s^x_u \equiv t^x_u$.

Proof Let $s \equiv t$. Then $s^x_u \equiv (\lambda x.s)u \equiv (\lambda x.t)u \equiv t^x_u$. ∎

The proof tells us that substitutivity is a straightforward consequence of the β-law. We may say that the β-law internalizes substitution.

Recall that we assume that terms satisfy the α-law. There is the possibility to work with pre-terms and to formalize α-equivalence with an inductive predicate. In such a set-up substitution may be defined as a partial operation on pre-terms. Church's original formalization of the λ-calculus is based on pre-terms and an explicit notion of α-equivalence. This style of formalization is also used in the textbook [6]. In contrast, Barendregt [1] and many others choose to work with an informal notion of terms satisfying the α-law (as we have done so far).

Exercise 6 Prove the following equivalences.

1. $SKK \equiv I$
2. $BCC \equiv \lambda fxy.fxy$

Exercise 7 Let $s \equiv t$ and $u \equiv v$. Prove $s^x_u \equiv t^x_v$.

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

Figure 3: Inductive definition of β-reduction

Exercise 8 We have $(\lambda x.s)x \equiv s$. Explain which substitution law is needed to obtain the equivalence.

6 Beta Reduction

The simplification of a β-redex $(\lambda x.s)t$ to s^x_t is known as β-reduction. Figure 3 defines an inductive predicate $s \succ t$ that formally defines β-reduction. The statement $s \succ t$ says that t is obtained from s by simplifying a single β-redex appearing as a subterm of s.

We write $s \succ^n t$ if t can be obtained from s with n β-reductions, and $s \succ^* t$ if $s \succ^n t$ for some $n \geq 0$. Formally, $s \succ^* t$ is the reflexive and transitive closure of $s \succ t$. We have $SKK \succ^2 \lambda x.Kx(Kx) \succ^2 I$.

Fact 9 If $s \succ t$, then $s \succ^* t$ and $s \equiv t$. Moreover, if $s \succ^* t$, then $s \equiv t$.

Fact 10 $s \equiv t$ is the least equivalence relation containing $s \succ t$.

A rigorous proof of Fact 10 requires work. We will study a proof latter.

Fact 11 (Free Variables) If $s \succ t$ and x is free in t, then x is free in s.

Fact 12 (Substitutivity) If $s \succ t$, then $s^x_u \succ t^x_u$. Moreover, if $s \succ^* t$, then $s^x_u \succ^* t^x_u$.

β-reduction is a computational notion. We may consider the process that given a term β-reduces the term as long as this is possible. If the term contains several β-redexes, the process may choose which β-redex it reduces next. There are terms on which β-reduction always terminates:

$$SKK \succ (\lambda gx.Kxg)K \succ (\lambda gx.(\lambda y.x)g)K \succ (\lambda g.I)K \succ I$$

There are also terms on which β-reduction never terminates:

$$\omega\omega \succ \omega\omega \succ \cdots$$

Finally, there are terms on which β-reduction may terminate or may not terminate:

$$KI(\omega\omega) \succ (\lambda y.I)(\omega\omega) \succ I$$
$$KI(\omega\omega) \succ KI(\omega\omega) \succ \cdots$$

$$\frac{}{(\lambda x.s)t \Rightarrow s_t^x} \qquad \frac{s \Rightarrow s'}{\lambda x.s \Rightarrow \lambda x.s'}$$

$$\frac{\text{neutral } s \qquad s \Rightarrow s'}{st \Rightarrow s't} \qquad \frac{\text{neutral } s \qquad \text{normal } s \qquad t \Rightarrow t'}{st \Rightarrow st'}$$

Figure 4: Inductive definition of leftmost-outermost reduction

A term is **normal** if it contains no β-redex. Note that the combinators I, K, S, and ω are normal, and that the combinator Ω is not normal.

Fact 13 A term is normal if and only if it cannot be β-reduced.

Exercise 14 Formalize normality of De Bruijn terms and show that it is decidable.

7 Evaluation

Given two terms s and t, we say that s **evaluates to** t and write $s \Downarrow t$ if $s \succ^* t$ and t is normal. If s evaluates to t, we say that t is a **normal form** of s. Note that the combinator Ω has no normal form. A main result about the lambda calculus says that every term has at most one normal form.

An **interpreter** for the lambda calculus is an algorithm that given a term computes a normal form of the term whenever there is one. From the term $KI\Omega$ we learn that the naive strategy that reduces some beta redex as long as there is one does not suffice for an interpreter (since $KI\Omega \succ KI\Omega$ but also $KI\Omega \succ I$). It is known that the strategy that always reduces the leftmost outermost β-redex finds a normal form whenever there is one.

In Coq we have β-redexes and β-reduction for typed terms. The typing ensures that β-reduction always terminates. Thus every term has a normal form in Coq.

We say that a term is **weakly normalizing** if it has a normal form. A term is **strongly normalizing** if there is no infinite β-reduction chain issuing from it. The term $KI\Omega$ is weakly normalizing but not strongly normalizing. If a term is strongly normalizing, a naive reduction strategy suffices for evaluation. We say that a term is **normalizing** if it is weakly normalizing.

Figure 4 shows the formal definition of leftmost-outermost reduction. A term is **neutral** if it is either a variable or an application.

We state formally that leftmost-outermost reduction is a deterministic and complete reduction strategy.

Fact 15 If $s \Rightarrow t$, then $s \succ t$.

Fact 16 If s is not normal, then there exists exactly one term t such that $s \Rightarrow t$.

Theorem 17 (Curry 1958) $s \Downarrow u$ if and only if $s \Rightarrow^* u$ and u is normal.

8 Church-Rosser Theorem

There is a fundamental theorem relating β-equivalence with β-reduction.

Theorem 18 (Church-Rosser 1936)
If $s \equiv t$, then $s \succ^* u$ and $t \succ^* u$ for some term u.

The proof of the theorem is not straightforward and the original proof by Church and Rosser is very informal. We will see a formal proof of the theorem latter. There are many important consequences of the Church-Rosser theorem. The consequences have straightforward proofs given the theorem.

Corollary 19

1. A term has at most one normal form.
2. If $s \equiv t$ and t is normal, then $s \succ^* t$ and $s \Downarrow t$.
3. If s and t are normal, then $s \equiv t$ iff $s = t$.
4. Different normal forms are not equivalent.

The facts stated by the corollary are of great importance for computational correctness proofs. If we want to argue that a term s evaluates to a term t or that two terms are equivalent as it comes to evaluation, we can use equational reasoning based on β-equivalence. This is easier than arguing about β-reduction directly.

Exercise 20 Prove Corollary 19.

Exercise 21 (Capture is deadly) Suppose $(\lambda y.yx)^x_y = \lambda y.yy$ (i.e., substitution is capturing for the given instance). Show that under this assumption every combinator is equivalent to I. Hint: Exploit substitutivity of β-equivalence. The Church-Rosser theorem is not needed.

9 Fixed Point Combinators

A **fixed point** of a function f is an argument x such that $fx = x$. It turns out that every function of the untyped λ-calculus has a fixed point. This important fact has a straightforward proof.

A **fixed point combinator** is a combinator R such that

$$Rs \equiv s(Rs)$$

for every term s. We can see a fixed point combinator as a function that yields a fixed point for every function of the lambda calculus.

Fact 22 R is a fixed point combinator if and only if $Rx \equiv x(Rx)$ for some variable x.

Proof Follows with substitutivity. ∎

Two well-known fixed point combinators are T and Y:[1]

$$\begin{aligned} A &:= \lambda xf.\ f(xxf) \\ T &:= AA \\ Y &:= \lambda f.\ (\lambda x.f(xx))(\lambda x.f(xx)) \end{aligned}$$

Fact 23 T is a fixed point combinator such that $Tf \succ^2 f(Tf)$.

Fact 24 Y is a fixed point combinator.

Proof We have

$$\begin{aligned} Yf &\succ (\lambda x.f(xx))(\lambda x.f(xx)) \\ Yf &\succ^2 f((\lambda x.f(xx))(\lambda x.f(xx)) \end{aligned}$$

Thus $Yf \equiv f(Yf)$. ∎

Exercise 25 Let X be a set of numbers with two different elements a and b. Give a function from X to X that has no fixed point.

Exercise 26

1. Give a term B such that $B \equiv xB$.
2. Give a term C such that $C \equiv \lambda xy.xCy$.
3. Give a term D such that $Dxyz \equiv xyzD$.

Exercise 27 Prove that there is no normal fixed point combinator.

Exercise 28 (Span) The **span** of a term is the minimal number of variables needed to write a term. It turns out that β-reduction can increase the span of a term. To show this, find a combinator M such that $M \succ^* \lambda x_1 \dots x_n.M(x_1 \dots x_n)$ for every $n \geq 1$. Hint: It suffices to find a combinator M such that $M \succ^* \lambda xy.M(xy)$

[1] T is attributed to Turing (1937) and Y to Curry and Rosembloom; see [6]. T is usually written as Θ.

10 Recursive Functions

We assume that the reader is familiar with the notion of a recursive function, which is realized in most programming languages. We will now see that the notion of a recursive function can be elegantly formalized in the untyped λ-calculus.

A declaration of a recursive function may look like this:

$$f := s$$

The arguments the recursive function f is taking may be expressed on the right hand side with the term s (e.g, we may have $s = \lambda xy.s'$). What makes the declaration recursive is the fact that the name f has at least one free occurrence in the term s. What the declaration is asking for is a term t such that

$$t \equiv s_t^f$$

or, equivalently,

$$t \equiv (\lambda f.s)t$$

Thus the declaration asks for a fixed point of the function $\lambda f.s$. Such a fixed point can be obtained with any fixed point combinator R. Given a fixed point combinator R, the function f declared by

$$f := s$$

can be given explicitly as the term $R(\lambda f.s)$.

This is all we need to say about recursive functions in the untyped lambda calculus. The next section contains several concrete examples for recursive functions in the untyped lambda calculus.

Exercise 29 Given distinct variables $f, x_1, \ldots, x_n$ and a term s, find a term t such that $tx_1 \ldots x_n \equiv s_t^f$. Informally, we may say that t solves the possibly recursive equation $fx_1 \ldots x_n = s$ for f.

11 Scott Representation of Inductive Data Types

Consider the inductive data type for natural numbers in Coq:

$$\mathsf{nat} := O : \mathsf{nat} \mid S : \mathsf{nat} \to \mathsf{nat}$$

The type provides us with two constructors O and S and a match for numbers. We may represent the match for numbers with the function

$$Mnaf := \mathsf{match}\ n \mid O \Rightarrow a \mid S\ n' \Rightarrow fn'$$

The function represents the two rules of the match with the arguments a and f. We call these arguments **continuations**.

In untyped lambda calculus, we now represent a number n as the function Mn. This gives us the following representation for numbers:

$$\begin{aligned}\hat{0} &:= \lambda af.a\\ \widehat{Sn} &:= \lambda af.f\,\hat{n}\end{aligned}$$

We speak of **Scott numerals** since this representation was invented by Dana Scott. Successor and predecessor functions for Scott numerals may be written as follows:

$$\begin{aligned}\mathsf{succ} &:= \lambda x.\,\lambda af.fx\\ \mathsf{pred} &:= \lambda x.\,x\,\hat{0}\,I\end{aligned}$$

Verification of the correctness of succ and pred is straightforward:

$$\begin{aligned}\mathsf{succ}\;\hat{n} &\equiv \widehat{Sn}\\ \mathsf{pred}\;\hat{0} &\equiv \hat{0}\\ \mathsf{pred}\;\widehat{Sn} &\equiv \hat{n}\end{aligned}$$

We now come to addition of Scott numerals. We start with a recursive addition function:

$$\mathsf{add}\;m\;n \;:=\; \mathsf{match}\;m \mid 0 \Rightarrow n \mid Sm' \Rightarrow S(\mathsf{add}\;m'\;n)$$

Except for the recursion, the translation of the function to λ-calculus is straightforward:

$$\mathsf{add}\;x\;y \;:=\; x\,y\,(\lambda z.\,\mathsf{succ}(\mathsf{add}\;z\;y))$$

As discussed in the previous section, the recursive function add can be expressed with a fixed point combinator R.

$$\begin{aligned}\mathsf{Add} &:= \lambda fxy.\,xy(\lambda z.\,\mathsf{succ}(fzy))\\ \mathsf{add} &:= R\,\mathsf{Add}\end{aligned}$$

Note that the encoding has an additional argument f for the recursive reference.

It is not difficult to verify the correctness of the encoding. First we prove that add satisfies the Dedekind equations:

$$\begin{aligned}\mathsf{add}\;\hat{0}\;y &\equiv y\\ \mathsf{add}\;\widehat{Sn}\;y &\equiv \mathsf{succ}\;(\mathsf{add}\;\hat{n}\;y)\end{aligned}$$

No induction is needed for the proof of the equivalences. Here is the proof of the second equivalence:

$$\begin{aligned}
\mathsf{add}\ \widehat{Sn}\ y &\equiv \mathsf{Add}\ \mathsf{add}\ \widehat{Sn}\ y && \text{fixed point property of } R\\
&\succ^3 \widehat{Sn}\ y\ (\lambda z.\,\mathsf{succ}\,(\mathsf{add}\ z\ y))\\
&\succ^2 (\lambda z.\,\mathsf{succ}\,(\mathsf{add}\ z\ y))\,\hat{n} && \text{definition of } \widehat{Sn}\\
&\succ \mathsf{succ}\ (\mathsf{add}\ \hat{n}\ y)
\end{aligned}$$

The correctness of add

$$\mathsf{add}\ \hat{m}\ \hat{n} \equiv \widehat{m+n}$$

now follows by natural induction on m using the Dedekind equivalences for add and the correctness equivalence for succ.

We now see that we can follow common functional programming techniques in the untyped λ-calculus and that the correctness of recursive functions can be established with standard techniques based on β-equivalence.

It is not difficult to write and verify functions in the untyped λ-calculus providing multiplication, subtraction, and primitive recursion for Scott numerals.

Moreover, Scott's encoding works for inductive data types in general, not just numbers. Here are the encodings for booleans, pairs, and lists:

$$\begin{aligned}
\mathsf{true} &:= \lambda ab.a & \mathsf{bool} &:= \mathsf{true} : \mathsf{bool} \mid \mathsf{false} : \mathsf{bool}\\
\mathsf{false} &:= \lambda ab.b\\
\mathsf{pair}\ x\ y &:= \lambda f.fxy & \mathsf{Pair} &:= \mathsf{pair} : X \to Y \to \mathsf{Pair}\\
\mathsf{nil} &:= \lambda af.a & \mathsf{list} &:= \mathsf{nil} : \mathsf{list} \mid \mathsf{cons} : X \to \mathsf{list} \to \mathsf{list}\\
\mathsf{cons}\ x\ y &:= \lambda af.fxy
\end{aligned}$$

Exercise 30 Define a recursive function computing factorials and prove its correctness.

Exercise 31 Define recursive functions that append and reverse lists in the lambda calculus and prove their correctness.

Exercise 32 (Mutual Recursion) Given terms s and t, find terms u and v such that $u \equiv suv$ and $v \equiv tuv$. Hint: Use T and pairs.

12 Declarations

We define a language on top of the untyped lambda calculus that offers declarations for possibly recursive functions:

$$\begin{array}{rcll} D & ::= & f x_1 \ldots x_n := s & \text{declaration} \\ P & ::= & s \mid D; P & \text{program} \end{array}$$

Note that a program is a sequence of declarations followed by a term. A program is a description of a term that can be obtained from the programm with the following compilation function C (R is some fixed point combinator).

$$\begin{array}{rcll} C\ s & := & s & \\ C\ (f x_1 \ldots x_n := s;\ P) & := & (C\ P)^f_{\lambda x_1 \ldots x_n . s} & \text{if } f \text{ not free in } s \\ C\ (f x_1 \ldots x_n := s;\ P) & := & (C\ P)^f_{R(\lambda f x_1 \ldots x_n . s)} & \text{if } f \text{ free in } s \end{array}$$

Note that the compilation function C defines the meaning of programs and declarations. We may speak of a definitional compiler.

13 Church Numerals and Primitive Recursion

Church encoded the natural numbers as iterator functions. This way operations like addition and multiplication can be expressed as non-recursive functions. Even a primitive recursion combinator can be expressed without recursion.

Following Church, we encode a number n as a normal combinator that iterates a given function n-times:

$$\begin{array}{rcl} \overline{n} & := & \lambda f a . f^n a \\ f^0 s & := & s \\ f^{n+1} s & := & f(f^n s) \end{array}$$

We call the term $\overline{n}$ the **Church numeral for n**. Here are the first four Church numerals.

$$\begin{array}{rcl} \overline{0} & = & \lambda f a . a \\ \overline{1} & = & \lambda f a . f a \\ \overline{2} & = & \lambda f a . f(f a) \\ \overline{3} & = & \lambda f a . f(f(f a)) \end{array}$$

Note that the Church numerals are normal combinators. This ensures that numerals for different numbers are not β-equivalent.

Sometimes it is helpful to think of a numeral $\overline{n}$ as an operator $\lambda f.f^n$ that applied to a function f yields the function f^n.

Exercise 33 Write a function in Coq mapping numbers to Church numerals represented as pre-terms. Prove that the function is injective and yield normal pre-terms.

13.1 Successor Function

We express the successor function as the following normal combinator.

$$\mathsf{succ} \;:=\; \lambda xfa.f(xfa)$$

The proof of the correctness statement

$$\mathsf{succ}\;\overline{n} \;\equiv\; \overline{n+1}$$

is straightforward:

$$\mathsf{succ}\;\overline{n} \equiv \lambda fa.\; f(\overline{n}fa) \equiv \lambda fa.\; f(f^n a) = \lambda fa.\; f^{n+1}a = \overline{n+1}$$

Exercise 34 Prove $\mathsf{succ}\, f\, g \;\equiv\; g \circ fg$.

13.2 Addition, Multiplication, Exponentiation

The so-called **Dedekind equations** fully characterize addition and multiplication of natural numbers based on the constructors 0 and S.

$$\begin{aligned} 0 + n &= n & \qquad 0 \cdot m &= 0 \\ Sm + n &= S(m+n) & \qquad Sm \cdot n &= n + m \cdot n \end{aligned}$$

From the equations we see that $m + n$ can be obtained by iterating S m-times on n, and that $m \cdot n$ can be obtained by iterating $+n$ m-times on 0. This leads to the following combinators for addition and multiplication:

$$\mathsf{add} \;:=\; \lambda x.x\,\mathsf{succ} \qquad\qquad \mathsf{mul} \;:=\; \lambda xy.x\,(\mathsf{add}\,y)\,\overline{0}$$

We first show that the combinators satisfy the Dedekind equations:

$$\begin{aligned} \mathsf{add}\;\overline{0}\;\overline{n} &\equiv \overline{n} & \qquad \mathsf{mul}\;\overline{0}\;\overline{n} &\equiv \overline{0} \\ \mathsf{add}\;\overline{Sm}\;\overline{n} &\equiv \mathsf{succ}\,(\mathsf{add}\;\overline{m}\;\overline{n}) & \qquad \mathsf{mul}\;\overline{Sm}\;\overline{n} &\equiv \mathsf{add}\;\overline{n}\,(\mathsf{mul}\;\overline{m}\;\overline{n}) \end{aligned}$$

Then we show by natural induction on m that the combinators provide addition and multiplication for Church numerals:

$$\mathsf{add}\;\overline{m}\;\overline{n} \;\equiv\; \overline{m+n} \qquad\qquad \mathsf{mul}\;\overline{m}\;\overline{n} \;\equiv\; \overline{m \cdot n}$$

Since the right hand sides of the equivalences are normal, the Church-Rosse theorem gives us the following correctness properties for evaluation:

$$\mathsf{add}\;\overline{m}\;\overline{n} \;\Downarrow\; \overline{m+n} \qquad\qquad \mathsf{mul}\;\overline{m}\;\overline{n} \;\Downarrow\; \overline{m \cdot n}$$

Exercise 35 Find a combinator providing exponentiation for Church numerals. Prove that your combinator exp satisfies $\mathsf{exp}\ \overline{m}\ \overline{n} \equiv \overline{m^n}$. Do the proof in detail.

Exercise 36 Prove the following equivalences:

a) $\mathsf{succ}^m\ \overline{n} \equiv \overline{m+n}$

b) $\overline{m+n} \equiv \lambda f.\ \overline{m}\ f \circ \overline{n}\ f$

c) $\overline{m \cdot n} \equiv \overline{m} \circ \overline{n}$

d) $\overline{m^n} \equiv \overline{n}\ (B\ \overline{m})\ \overline{1}$

13.3 Predecessor

At first, writing a predecessor function for Church numerals seems difficult. The trick is to iterate on pairs.[2] We start from the pair $(0,0)$ and iterate n-times to obtain the pair $(n, n-1)$.

$$(0,0) \rightsquigarrow (1,0) \rightsquigarrow (2,1) \rightsquigarrow \cdots \rightsquigarrow (n, n-1)$$

To do so, we need to represent pairs as functions.

13.4 Pairs

We encode pairs as follows.

$$\begin{aligned} \mathsf{pair} &:= \lambda xyf.fxy \\ \mathsf{fst} &:= \lambda p.p(\lambda xy.x) \\ \mathsf{snd} &:= \lambda p.p(\lambda xy.y) \end{aligned}$$

The following equivalences are easy to prove:

$$\begin{aligned} \mathsf{fst}\ (\mathsf{pair}\, x\, y) &\equiv x \\ \mathsf{snd}\ (\mathsf{pair}\, x\, y) &\equiv y \end{aligned}$$

13.5 Primitive Recursion

Primitive recursion is a definition scheme for functions on the natural numbers introduced by Peano in 1889 as a compagnon to natural induction. We will define a primitive recursion combinator prec satisfying the equivalences

$$\begin{aligned} \mathsf{prec}\ x\ f\ \overline{0} &\equiv x \\ \mathsf{prec}\ x\ f\ \overline{Sn} &\equiv f\,\overline{n}\,(\mathsf{prec}\ x\ f\ \overline{n}) \end{aligned}$$

[2] It is reported that Kleene, then a student of Church, came up with the idea during dental treatment.

As with predecessor, we realize primitive recursion with iteration on pairs:

$$(\overline{0}, x) \rightsquigarrow (\overline{1}, f\,\overline{0}\,x) \rightsquigarrow (\overline{2}, f\,\overline{1}\,(t\,\overline{0}\,x)) \rightsquigarrow \cdots$$

This leads to the following definition.

$$\begin{aligned} \mathsf{a} &:= \lambda x.\ \mathsf{pair}\ \overline{0}\ x \\ \mathsf{step} &:= \lambda f p.\ \mathsf{pair}\ (\mathsf{succ}\ (\mathsf{fst}\ p))\ (f\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \\ \mathsf{prec} &:= \lambda x f n.\ \mathsf{snd}\ (n\ (\mathsf{step}\ f)\ (a x)) \end{aligned}$$

Showing the first correctness equivalence for prec is easy. For the second correctness equivalence we need the following lemma.

$$\overline{Sn}\ (\mathsf{step}\ f)\ (a x) \equiv \mathsf{pair}\ \overline{Sn}\ (f\ \overline{n}\ (\mathsf{snd}\ (\overline{n}\ (\mathsf{step}\ f)\ (a x))))$$

The lemma follows by induction on n.

Exercise 37 Prove the correctness of prec.

Exercise 38 The predecessor operation can be expressed with primitive recursion.

$$\mathsf{pred} := \mathsf{prec}\ \overline{0}\ (\lambda x y.x)$$

Show the correctness of pred by proving the following equivalences.

$$\begin{aligned} \mathsf{pred}\ \overline{0} &\equiv \overline{0} \\ \mathsf{pred}\ \overline{Sn} &\equiv \overline{n} \end{aligned}$$

14 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 39 (Böhm 1968) Let s and t be different $\beta\eta$-normal combinators. Then there exist combinators $u_1, \ldots, u_n$ such that

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

for all variables x and y.

Corollary 40 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.

Exercise 41 Prove $BCC \equiv I$ under $\beta\eta$-equivalence.

Exercise 42 Prove that the η-law follows from $\lambda x.fx \equiv f$ provided $f \neq x$.

Exercise 43 Find combinators $u_1, \ldots, u_n$ that separate $\overline{0}$ and $\overline{1}$ as described by Böhm's theorem. Do the same for $\overline{0}$ and $\overline{2}$.

Exercise 44 Prove by induction on n that $\overline{m^n} \equiv \overline{n}\ \overline{m}$ under $\beta\eta$-equivalence. Use the equivalence $\overline{m \cdot n} \equiv \overline{m} \circ \overline{n}$ from Exercise 36.

15 Church Numerals in Coq

We will now represent Church numerals and their operations in Coq and prove the correctness of the operations. This will deepen our understanding of Church numerals and raise some interesting issues about Coq.

Since Coq is typed, we must represent Church numerals as typed functions. We represent Church numerals as members of the type

```
Definition Nat : Prop := ∀ X : Prop, (X → X) → X → X.
```

It is crucial that the variable X ranges over propositions rather than general types. This will be explained later.

We define a function N that maps a number n to the numeral $\overline{n}$.

```
Definition zero : Nat := fun X f x ⇒ x.
Definition succ : Nat → Nat := fun n X f x ⇒ f (n X f x).
Definition N : nat → Nat := fun n ⇒ nat_iter n succ zero.
```

Use the command Compute N 7 to see the Church numeral for 7. Following the Dedekind equations, we express addition, multiplication, and exponentiation as follows.

```
Definition add : Nat → Nat → Nat := fun m ⇒ m Nat succ.
Definition mul : Nat → Nat → Nat := fun m n ⇒ m Nat (add n) (N 0).
Definition exp : Nat → Nat → Nat := fun m n ⇒ n Nat (mul m) (N 1).
```

We can now prove the following correctness statements.

- succ (N n) = N (S n)
- add (N m) (N n) = N (m + n).
- mul (N m) (N n) = N (m * n).
- exp (N m) (N n) = N (pow m n).

All proofs are straightforward and are based on the characteristic equations for the operations, which hold by conversion. The proofs for addition and multiplication are by induction on m, and the proof for exponentiation is by induction on n, as one would expect from the definitions. The correctness proof for addition looks as follows.

```
Lemma add_correct m n :
  add (N m) (N n) = N (m + n).
Proof.
  induction m; simpl.
  – reflexivity.
  – change (add (N (S m)) (N n)) with (succ (add (N m) (N n))).
    now rewrite IHm.
Qed.
```

Exercise 45 (Primitive Recursion) Find a function

$$\text{prec} \;:\; \forall X : \text{Prop}.\; X \to (\text{Nat} \to X \to X) \to \text{Nat} \to X$$

such that

$$\begin{aligned} \text{prec}\; x\; f\; (N0) &= x \\ \text{prec}\; x\; f\; (N(Sn)) &= f\; (Nn)\; (\text{prec}\; x\; f\; (Nn)) \end{aligned}$$

15.1 Predicativity of Coq's Universe Type

We now explain why the definition

```
Definition Nat : Type := ∀ X : Type, (X → X) → X → X.
```

does not work. The reason is that Type is a **predicative universe**. This means that the functions of a type $A := \forall X : \mathsf{Type}.\ s$ can only be applied to types that are *smaller than* A. In particular, if f is a function of type A, then f cannot be applied to A. As a consequence, the definition

```
Definition add : Nat → Nat → Nat := fun m ⇒ m Nat succ.
```

will not type check since m:Nat is applied to Nat.

In contrast, Prop is an **impredicative universe** where a size restriction on types does not exist. The universe Type can not be made impredicative since this would result in an inconsistent system where False is provable. This is a basic fact of logic that cannot be massaged away.

Since Nat is a proposition, it follows that we express numerals and operations on numerals as proofs. So our representation of Church numerals shows that Coq's proof language has considerable computational power. Since numerals are proofs, we cannot show in Coq that the embedding function $N : \mathsf{nat} \to \mathsf{Nat}$ is injective (because of the elim restriction). Nevertheless, we can observe from the outside that N yields different numerals for different numbers.

15.2 Church Exponentiation

Since Coq has η-conversion, we can prove the equation $\overline{m^n} = \overline{mn}$ and use it to obtain an exponentiation function. We speak of **Church exponentiation**. For the proof to go through, we need suitable equations for multiplication and addition. The following works.

```
Definition add : Nat → Nat → Nat := fun m n X f x ⇒ m X f (n X f x).
Definition mul : Nat → Nat → Nat := fun m n X f ⇒ m X (n X f).
Definition exp : Nat → Nat → Nat := fun m n X ⇒ n (X → X) (m X).
Lemma add_correct m n :  N (m + n) = add (N m) (N n).
Lemma mul_correct m n :  N (m * n) = mul (N m) (N n).
Lemma exp_correct m n :  N (pow m n) = exp (N m) (N n).
```

Interestingly, the above encodings of addition, multiplication, and exponentiation will type check if Nat is defined with Type rather than Prop since they do not require an application of a numeral to the type Nat. However, once we encode the predecessor operation, the typing problem will reoccur and cannot be avoided.

16 SK-Terms and Weak Reduction

SK-terms are defined inductively:

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

Theorem 46 Every term is equivalent to an SK-term.

The proof of the theorem is based on an **abstraction operator** that for a variable x and an SK-term s yields an SK-term ${}^x s$ such that ${}^x s \equiv \lambda x.s$. The abstraction operator is defined by structural recursion on SK-terms:

$$\begin{array}{rcll} {}^x x & := & SKK & \\ {}^x s & := & Ks & \text{if } x \text{ not free in } s \\ {}^x st & := & S\ {}^x s\ {}^x t & \text{otherwise} \end{array}$$

Fact 47 Let s be an SK-term. Then:

1. ${}^x s \succ^* \lambda x.s$.
2. y free in ${}^x s$ if and only if $y \neq x$ and y free in s.

Proof By induction on s. ∎

Next we define a **translation operator** $[s]$ translating every term into an equivalent SK-term. The definition is by structural recursion on terms.

$$\begin{array}{rcl} [x] & := & x \\ [st] & := & [s][t] \\ [\lambda x.s] & := & {}^x[s] \end{array}$$

Fact 48 $[s]$ is an SK-term such that $[s] \succ^* s$.

Proof By induction on s using Fact 47. ∎

Weak reduction is a restricted form of β-reduction that reduces only redexes of the form $Sstu$ and Kst. Weak reduction is arranged so that it reduces SK-terms to SK-terms. We define **weak reduction** $s \succ_w t$ with an inductive predicate defined on all terms:

$$\frac{}{Kst \succ_w s} \qquad \frac{}{Sstu \succ_w su(tu)} \qquad \frac{s \succ_w s'}{st \succ_w s't} \qquad \frac{t \succ_w t'}{st \succ_w st'}$$

Here are facts about weak reduction.

$$s,t \;::=\; x \mid K \mid S \mid st \qquad (x : \mathbf{N})$$

$$\frac{}{Kst \succ s} \qquad \frac{}{Sstu \succ su(tu)} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{st \succ st'}$$

$$\frac{}{Kst \equiv s} \qquad \frac{}{Sstu \equiv su(tu)} \qquad \frac{s \equiv s' \qquad t \equiv t'}{st \equiv s't'} \qquad \frac{}{s \equiv s} \qquad \frac{s \equiv t}{t \equiv s} \qquad \frac{s \equiv t \qquad t \equiv u}{s \equiv u}$$

Figure 5: Definition of SK

- Weak reduction does not involve substitution.
- Weak reduction does not happen below lambda (i.e., inside abstractions).
- Weak reduction applies only to β-redexes of the form Kst or $Sstu$.
- SK-terms are closed under weak reduction.
- If $s \succ_w t$, then $s \succ^2 t$ or $s \succ^3 t$.

Fact 49 Let s and t be SK-terms. Then:

1. $({}^x s)t \succ_w^* s_t^x$.
2. $({}^x s)_t^y = {}^x(s_t^y)$ provided $x \neq y$ and x is not free in t.

Proof By induction on s. ∎

The system consisting of SK-terms and weak reduction is known as **combinatory logic** (see [6] for a full development). In combinatory logic, S and K are accommodated as constants since their representation as λ-terms is not relevant. We will refer to this system as **SK**. A formal definition of SK is shown in Figure 5. SK satisfies the Church-Rosser property.

Similar to the λ-calculus, SK is a Turing-complete programming language. There is a fixed point combinator and inductive data types can be expressed following Scott's encoding. Programming is tedious in SK since abstractions must be translated away. On the other hand, SK has a straightforward formal definition.

Given the translation from terms to SK-terms, we may ask whether weak reduction can fully simulate β-reduction. This is not the case since weak reduction can not simulate **deep β-reductions** (i.e., β-reductions that happen within abstractions). See Exercise 57.

We may see SK as a subsystem of lambda calculus. This view becomes more natural if we generalize the lambda calculus to the so-called multivariate λ-calculus [9] where an abstraction may bind $n \geq 1$ arguments and can only β-reduce with that many arguments.

Exercise 50 Prove that every closed term is equivalent to a term that can be written with at most three bound variables.

Exercise 51 Which condition must a term s satisfy so that $\lambda x.s \equiv Ks$?

Exercise 52 Prove Facts 47, 48, and 49.

Exercise 53 Give the following SK-terms: $[I]$, $[\omega]$, $[\Omega]$, $[\lambda x.\, S(K(xx))]$.

Exercise 54 (Divergence) Prove $[\Omega] \succ_w^n [\Omega]$ for some $n > 0$.

Exercise 55 (Fixed point combinator) Prove $[T]s \succ_w^* s([T]s)$.

Exercise 56 (Church numerals in SK) We define some SK-terms (overwriting definitions done before in λ-calculus):

$$I := SKK \qquad \overline{0} := KI \qquad \text{succ} := S(K(SB))I$$
$$B := S(KS)K \qquad \overline{n+1} := SB\,\overline{n}$$

Prove the following:

a) $Bstu \succ_w^* s(tu)$

b) $\text{succ}\ s \succ_w^* SBs$

c) $\overline{n}\,st \succ_w^* s^n t$

Exercise 57 Weak reduction cannot simulate deep β-reduction. For instance, $\lambda x.Ix \succ I$, but not $[\lambda x.Ix] \succ_w^* [I]$. Determine $[\lambda x.Ix]$ and $[I]$ and prove the claim.

Exercise 58 (Shallow λ-calculus) Show that the shallow lambda calculus does not satisfy the Church Rosser property. Shallow means that an occurrence of a β-redex can only be reduced if it is not below a binder. Formally, shallow equivalence and shallow reduction may be defined by omitting the compatibility rules for abstractions in Figure 2 and Figure 3. Hint: Consider $K(II)$ and note that shallow reduction may bury redexes. Note that weak reduction does not suffer from this problem.

Exercise 59 (Idempotent translation) Hindley and Seldin [6] give a translation $[s]^*$ from terms to SK-terms satisfying the following properties:

1. *Soundness:* $[s]^*$ is an SK-term.
2. *Idempotence:* $[s]^* = s$ if s is an SK-term.
3. *η-property:* $[\lambda x.sx]^* = [s]^*$ if x is not free in s.
4. *β-property:* $[\lambda x.s]^* t \succ_w^* ([s]^*)_t^x$.

Give the translation and prove the properties.

17 Weak Call-By-Value Lambda Calculus

We have seen three variants of the untyped λ-calculus so far: $\lambda\beta$ (λ-calculus with β-reduction), $\lambda\beta\eta$ (λ-calculus with β-and η-reduction), and **SK** (SK-terms with weak reduction). We see $\lambda\beta$ as a subsystem of $\lambda\beta\eta$, and SK as a subsystem of $\lambda\beta$. We now introduce a further subsystem of $\lambda\beta$ we call **L**. Using phrases from the literature, we may characterise L as a weak call-by-value λ-calculus.

L is obtained from the λ-calculus by imposing three restrictions on β-reduction:

1. A β-redex st can only be reduced if t is an abstraction. We speak of **call-by-value β-reduction**.
2. An occurrence of a β-redex can only be reduced if it does not appear within an abstraction. We speak of **shallow β-reduction**.
3. A β-redex can only be reduced if it is closed.

Thus II is reducible in L, but $\lambda x.II$ and Ix are not reducible in L. Moreover, the term $K\Omega$ does not have a normal form in L.

The restricted form of reduction employed by L agrees with what is realized in call-by-value functional programming languages such as ML. The call-by-value requirement ensures that a β-redex st can only be reduced after the argument term t has been fully evaluated.

As it comes to programming, L retains most of the nice properties of $\lambda\beta$. Moreover, L enjoys a strengthened Church-Rosser theorem ensuring that every normalizing term is strongly normalizing. In fact, all reductions of a term to its normal form do have the same length.

The requirement that only closed β-redexes be reduced allows for a much simplified substitution operation not taking care of free variables.

Figure 6 shows the formal definition of L using De Bruijn terms. For technical simplicity, the requirement that only closed β-redexes are reduced is not enforced. However, the definition of L uses a simplified substitution operation that is only correct if there are no free variables (see Exercise 71). If $s \succ_{\mathrm{L}} t$ and s is closed, then $s \succ t$ in $\lambda\beta$ and $\lambda\eta$.

Fact 60 If $s \succ_{\mathrm{L}} t$, then $s \equiv_{\mathrm{L}} t$.

Fact 61 (Uniform Confluence) Suppose $s \succ_{\mathrm{L}} t_1$ and $s \succ_{\mathrm{L}} t_2$. Then either $t_1 = t_2$ or there exists a term u such that $t_1 \succ_{\mathrm{L}} u$ and $t_1 \succ_{\mathrm{L}} u$.

Proof Observe that redexes cannot be nested and that reduction of redexes is deterministic. The claim now follows by induction on $s \succ_{\mathrm{L}} t_1$. ∎

Uniform confluence implies the Church-Rosser property.

$$s,t ::= n \mid st \mid \lambda s \qquad (n : \mathbf{N})$$

$$\begin{aligned} n_u^k &:= \text{if } n = k \text{ then } u \text{ else } n \\ (st)_u^k &:= s_u^k\, t_u^k \\ (\lambda s)_u^k &:= \lambda(s_u^{k+1}) \end{aligned}$$

$$\frac{}{(\lambda s)(\lambda t) \succ_{\mathrm{L}} s^0_{\lambda t}} \qquad \frac{s \succ_{\mathrm{L}} s'}{st \succ_{\mathrm{L}} s't} \qquad \frac{t \succ_{\mathrm{L}} t'}{st \succ_{\mathrm{L}} st'}$$

$$\frac{}{(\lambda s)(\lambda t) \equiv_{\mathrm{L}} s^0_{\lambda t}} \qquad \frac{s \equiv_{\mathrm{L}} s' \qquad t \equiv_{\mathrm{L}} t'}{st \equiv_{\mathrm{L}} s't'} \qquad \frac{}{s \equiv_{\mathrm{L}} s} \qquad \frac{s \equiv_{\mathrm{L}} t}{t \equiv_{\mathrm{L}} s} \qquad \frac{s \equiv_{\mathrm{L}} t \qquad t \equiv_{\mathrm{L}} u}{s \equiv_{\mathrm{L}} u}$$

Figure 6: Definition of L

Fact 62 (Church-Rosser Property)
If $s \equiv_{\mathrm{L}} t$, then $s \succ_{\mathrm{L}}^* u$ and $t \succ_{\mathrm{L}}^* u$ for some term u.

Uniform confluence gives L another useful property the other three systems ($\lambda\beta\eta$, $\lambda\beta$, SK) do not have.

Fact 63 (Uniform Normalisation) Let $s \succ_{\mathrm{L}}^m u$ und $s \succ_{\mathrm{L}}^n v$. If u is irreducible, then either $m = n$ and $u = v$ or $n < m$ and $v \succ_{\mathrm{L}}^{m-n} u$.

Fact 64 (Fixed Point Combinator) $Ts \succ_{\mathrm{L}}^2 s(Ts)$ provided s is an abstraction.

We call closed abstractions **procedures** and take the view that L computes with procedures.

In order that we can program with recursive functions in L, it is essential that recursive functions can be expressed as abstractions. Note that Ts is not even normalizing. The problem can be solved by putting an η-expansion into the definition of A (see §9). We define a function ρ from terms to abstractions:

$$\begin{aligned} D &:= \lambda zfx.\, f(zzf)x \\ \rho s &:= \lambda x.\, s(DDs)x \end{aligned}$$

We refer to ρ as **recursion operator**.

Fact 65 If s is a procedure, then ρs is a procedure.

Fact 66 (Recursive procedures) $(\rho s)t \succ_{\mathrm{L}}^3 s(\rho s)t$ provided s and t are procedures.

Proof We have $(\rho s)t \succ_{\mathrm{L}} s(DDs)t$ and $DDs \succ_{\mathrm{L}}^{2} \rho s$. The claim follows. ∎

Exercise 67 (Primitive Recursion) We represent numbers using Scott's encoding:

$$\overline{0} := \lambda af.a$$
$$\overline{Sn} := \lambda af.f\,\overline{n}$$

a) Give a procedure succ and prove $\mathsf{succ}\ \overline{n} \equiv_L \overline{n+1}$.

b) Give a procedure add and prove $\mathsf{add}\ \overline{m}\ \overline{n} \equiv_L \overline{m+n}$.

c) Give a procedure prec and prove, for procedures s and t,

$$\mathsf{prec}\ \overline{0}\ s\ t \equiv_L s$$
$$\mathsf{prec}\ \overline{n+1}\ s\ t \equiv_L s\ \overline{n}\ (\mathsf{prec}\ \overline{n}\ s\ t)$$

Exercise 68 (Church Numerals in L) Find a function N from numbers to terms and procedures succ and add such that:

a) Nn is a procedure.

b) Nn is a strongly normalizing term in $\lambda\beta$.

c) $(Nn)st \succ_{\mathrm{L}}^{*} s^n t$ provided s and t are procedures.

d) $\mathsf{succ}\,(Nn) \succ_{\mathrm{L}}^{*} N(n+1)$.

e) $\mathsf{add}\,(Nm)(Nn) \succ_{\mathrm{L}}^{*} N(m+n)$.

Verify the properties.

Exercise 69 Show that none of the systems $\lambda\beta\eta$, $\lambda\beta$, and SK satisfies uniform normalisation.

Exercise 70 Show that L-reduction is not substitutive.

Exercise 71 (Disagreement of L-Reduction) An L-reduction $(\lambda s)(\lambda t) \succ_{\mathrm{L}} s^0_{\lambda t}$ is a β-reduction in $\lambda\beta$ if $(\lambda s)(\lambda t)$ is closed. If $(\lambda s)(\lambda t)$ is open, $(\lambda s)(\lambda t) \succ_{\mathrm{L}} s^0_{\lambda t}$ is defined but $(\lambda s)(\lambda t) \succ s^0_{\lambda t}$ may not hold. Consider, for instance, the following β-reduction in $\lambda\beta$:

$$(\lambda y.\, x(\lambda z.y))(\lambda y.x) \succ x(\lambda zy.x)$$

In De Bruijn notation, we have

$$(\lambda(x{+}1)(\lambda 1))(\lambda(x+1)) \succ x(\lambda\lambda(x{+}2))$$

However, with L-reduction we have something different:

$$(\lambda(x{+}1)(\lambda 1))(\lambda(x+1)) \succ_{\mathrm{L}} (x{+}1)(\lambda\lambda(x+1))$$

Exercise 72 (Span) We define the span of a De Bruijn term as the largest index occurring in the term. Show that L-reduction does not increase the span of a term. Compare the situation with reduction in $\lambda\beta$, see Exercise 28.

18 Summary: Untyped Lambda Calculus as Programming Language

We have seen that the untyped lambda calculus provides us with an idealized functional programming language with recursive functions and inductive data types. The combination of inductive datatypes and recursive functions gives us a Turing-complete language (i.e., all computable functions can be expressed).

What is spectacular about the untyped λ-calculus is its simplicity and elegance, both as it comes to programming and program verification. There are only three syntactic constructs and only one semantic principle (the β-law). Values of inductive data types are expressed as normal combinators. We can reason about the correctness of procedures just based on β-equivalence.

19 Historical Remarks

Church and his coworkers had tremendous difficulties in coming up with a rigorous formulation of the λ-calculus, as witnessed, for instance, by Church's presentation [2] from 1941. Much early work on the λ-calculus was done by Curry and Feys [3]. Both Church [2] and Curry and Feys [3] work with pre-terms and treat α-equivalence explicitly. Barendregt [1] pioneered the informal and abstract view of terms based on the α-assumption. Barendregt [1] (Appendix C) also suggests that a term should be understood as a De Bruijn term. Hindley and Seldin [6] do not follow Barendregt's lead and works with pre-terms and explicit α-equivalence.

According to Barendregt [1], the precise nature of variables and the difference between free and bound variables was first clarified by Frege and Peirce. The idea to express bound variables with S and K was first formulated by Schönfinkel [10].

Much information about SK and related systems of combinatory logic can be found in Hindley and Seldin [6].

Interest in the weak call-by-value calculus is more recent. Dal Lago and Martini [7] show that the weak call-by-value calculus and Turing machine can simulate each other with polynomial overhead.

Scott's encoding of inductive data types appeared first for numbers [4, 11]. Mogensen [8] used Scott's encoding for terms and constructed a self-interpreter. Dal Lago and Martini [7] use Scott's encoding for the weak call-by-value calculus.

References

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

[2] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.

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

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

[5] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34:381–392, 1972.

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

[7] Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008.

[8] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[9] Garrel Pottinger. A tour of the multivariate lambda calculus. In J. Michael Dunn and Anil Gupta, editors, *Truth or Consequences*, pages 209–229. Springer Netherlands, 1990.

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

[11] Christopher Wadsworth. Some unusual λ-calculus numeral systems. In Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin, editors, *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 215–230. Academic Press, 1980.