Lambda Calculus - Step by Step

Helmut Brandl

(firstname dot lastname at gmx dot net)

Abstract

This little text gives a step by step introduction into untyped lambda calculus. All needed theory is explained and no special know how is assumed. Although elementary, all important theorems about untyped lambda calculus including some undecidability theorems are given and proved within this text.

It has been tried to use a notation which is easy to understand with a lot of graphic notation to support a good intuition about the presented material.

Contents

1 Motivation

Why study lambda calculus?

Let us put the question in some historical context. At the beginning of the 20th century the famous mathematician David Hilbert challenged the mathematical community by the statement that mathematical problems must be decidable. At the 1930 annual meeting of the *Society of German Scientists and Physicians* he made his famous quote "We must know, we will know".

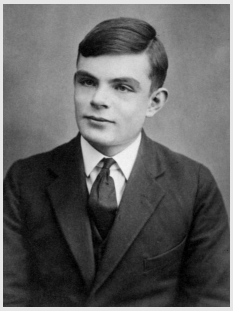
David Hilbert

Challenge →

"Entscheidungsproblem" (Decision Problem). Mathematics must be decidable. "We must know, we will know!"

Kurt Gödel (1931): Incompleteness Theorems

Alonzo Church (1936): Lambda Calculus



Alan Turing (1936): Turing Machine

The young mathematician Kurt Gödel attended the meeting and expressed some doubts to his collegues about the general decidability of mathematical statements. One year later in 1931 he published his famous incompleteness theorems [3]. He proved that for all consistent formal systems which are capable of expressing logic and doing simple arithmetics there are certain statements which are not provable withing the system but true. These incompleteness theorems are considered as the first serious blow of Hilbert's program.

Five years later Alonzo Church [2] and Alan Turing [1] independently proved that the *decision problem* cannot be solved. Alonzo

Church invented the lambda calculus and Alan Turing his automatic machine (today called Turing machine) which are both equivalent in expressiveness.

Although Church's lambda calculus has been published slightly before Alan Turing published his paper on automatic machines usually Turing machines are used define computability and decidability. Turing machines resemble more the structure of modern computers than lambda calculus. A programming language is called *Turing complete* if all possible algorithms can be coded within the language. Nobody talks about *lambda complete*.

However lambda calculus is a quite fascinating model of computation. The lambda calculus invented by Alonzo Church is remarkably simple. It consists just of variables, function applications and lambda abstractions. But the calculus is sufficiently powerful to express all computable functions and decision procedures.

Beside its expressive power lambda calculus is used as the theoretical base of functional languages like Haskell, ML, F#.

In this paper we explain the lambda calculus in its purest form as untyped lambda calculus.

2 Inductive Sets and Relations

2.1 Inductive Sets

Set Notation A set is an unordered collection of objects. If the object a is an element of the set A we write $a \in A$.

A set A is a subset of set B if all elements of the set A are also elements of the set B. The symbol $\subseteq$ is used to express the subset relation. The operator $:=$ is used to express that something is valid by definition. Therefore the subset relation is defined symbolically by $A \subseteq B := \forall a \,.\, a \in A \Rightarrow a \in B$. The statement $A \subseteq B$ can always be replaced by its definition $\forall a \,.\, a \in A \Rightarrow a \in B$ and vice versa.

The double arrow $\Rightarrow$ is used to express implication. $p \Rightarrow q$ states the assertion that having a proof of p we can conclude q. The assertion $p \Rightarrow q$ is proved by assuming p and deriving the validity of q.

Rule Notation We have often several premises which are needed reach a conclusion. E.g. we might have the assertion $p_1 \wedge p_2 \wedge \ldots \Rightarrow c$. Then we use the rule notation $\dfrac{p_1, p_2, \ldots}{c}$ or $\dfrac{\begin{array}{c} p_1 \\ p_2 \\ \vdots \end{array}}{c}$ to express the same fact. Evidently the order of the premises is not important.

Variable in rules are universally quantified. Therefore we can state the subset definition $\forall a \,.\, a \in A \Rightarrow a \in B$. in rule notation more compactly and better readable as $\dfrac{a \in A}{a \in B}$. It should be clear from the context which symbols denote variables.

Inductive Definition of Sets Rules can be used to define sets inductively. The set of even numbers E can be defined by the two rules $0 \in E$ and $\dfrac{n \in E}{n + 2 \in E}$. A set defined by rules is the least set which satisfies the rules. The set E of even number must contain the number 0 and with all numbers n it contains also the number $n + 2$.

The fact that some object is an element of an inductively defined set must be established by an arbitrarily long but finite sequence of applications of the rules which define the set. A proof of $4 \in E$ consists of a proof of $0 \in E$ by application of the first rule and then two applications of the second rule to reach $2 \in E$ and $4 \in E$.

Rule Induction If we have the fact that an object is an element of some inductively defined set then we can be sure that it is in the set because of one of the rules which define the set.

This can be used to prove facts by rule induction. Suppose we want to prove that some property p is shared by all even numbers. We can express this statement by the rule $\frac{n \in E}{p(n)}$. Note that variables in rules are implicitly universally quantified, i.e. the rule expresses the statement $\forall n \,.\, n \in E \Rightarrow p(n)$.

It is possible to prove this statement by induction on $n \in E$. Such a proof consists of a proof of the statement for each rule. In the case of even numbers there are two rules.

For the first rule we have to prove the the number 0 satisfies the property $p(0)$.

For the second rule we assume that $n \in E$ because there is some other number m already in the set of even numbers E and $n = m + 2$. I.e. we have to prove the goal $p(m + 2)$ under the premise $m \in E$. Because of the premise $m \in E$ we can assume the induction hypothesis $p(m)$. I.e. we can assume $m \in E$ and $p(m)$ and derive the validity of $p(m + 2)$.

If the proof succeeds for both rules we are allowed to conclude that the property p is satisfied by all even numbers.

Natural Number Induction It is not difficult to see that the usual law of induction on natural numbers is just a special case of rule induction. We can define the set of natural numbers inductively $\mathbb{N}$ by the rules

1. $0 \in \mathbb{N}$
2. $\frac{n \in \mathbb{N}}{n' \in \mathbb{N}}$

where n' denotes the successor of n i.e. 3 is just a shorthand for $0'''$.

The usual induction law of natural numbers allows to prove a property p for all natural number by a proof of $p(0)$ and a proof of $\forall n \,.\, p(n) \Rightarrow p(n')$ which are exactly the requirements of a proof by rule induction.

Grammar Notation In some cases it is convenient to define a set by a grammar. E.g. we can define the set of natural numbers by all

terms n generated by the grammar

$$n ::= 0 \mid n'$$

i.e. we can use the corresponding induction law to prove that all terms generated by the grammar satisfy a certain property.

This definition is just a special form of the definition of the set of natural numbers $\mathbb{N}$ by the rules

1. $0 \in \mathbb{N}$
2. $\dfrac{n \in \mathbb{N}}{n' \in \mathbb{N}}$

2.2 Inductive Relations

Relations n-ary relations are just sets of n-tuples. A binary relation r over the sets A and B is a subset of the cartesian product

$$r \subseteq A \times B.$$

We use the notations $(a, b) \in r$, $r(a, b)$ and $a \to_r b$ to denote the fact that the pair (a, b) with $a \in A$ and $b \in B$ figure in the relation r.

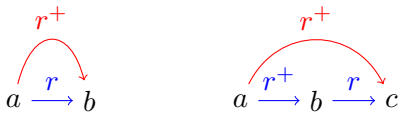
In this paper we need only endorelations i.e. binary relations where the domain A and the range B of the relation are the same set.

Relation Closure As with sets, relations can be defined inductively.

Definition 2.1. The *transitive closure* r^+ of a relation r is defined by the rules

1. $\dfrac{r(a, b)}{r^+(a, b)}$
2. $\dfrac{r^+(a, b),\ r(b, c)}{r^+(a, c)}$

The rules can be displayed graphically. The premises are marked blue and the conclusion is marked red.



We called r^+ the transitive closure of r, but the fact that r^+ is transitive needs a proof.

Theorem 2.2. The transitive closure r^+ of the relation r is transitive i.e. $\dfrac{a \to_r^+ b,\ b \to_r^+ c}{a \to_r^+ c}$ is valid.

Proof. Assume $a \to_r^+ b$ and prove the goal $a \to_r^+ c$ by induction on $b \to_r^+ c$.

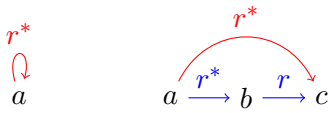
1. Goal $a \to_r^+ c$ assuming $b \to_r^+ c$ and that $b \to_r^+ c$ is valid by rule 1 of the transitive closure.
   Premise $b \to_r c$.
   The goal is valid by the assumption $a \to_r^+ b$, the premise $b \to_r c$ and rule 2 of the transitive closure.
2. Goal $a \to_r^+ d$ assuming $b \to_r^+ d$ and that $b \to_r^+ d$ is valid by rule 2 of the transitive closure.
   Premises $b \to_r^+ c$ and $c \to_r d$.
   Induction hypothesis $a \to_r^+ c$.
   The goal is valid by the induction hypothesis $a \to_r^+ c$, the premise $c \to_r d$ and rule 2 of the transitive closure.

□

Definition 2.3. The *reflexive transitive closure* r^* of a relation r is defined by the rules

1. $r^*(a, a)$
2. $\dfrac{r^*(a, b),\ r(b, c)}{r^*(a, c)}$

Graphical representation of the rules:



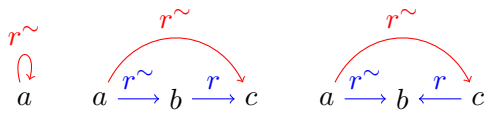
Theorem 2.4. The reflexive transitive closure r^* of the relation r is transitive i.e. $\dfrac{a \to_r^* b,\ b \to_r^* c}{a \to_r^* c}$ is valid. Proof similar to the proof of theorem 2.2.

Definition 2.5. The *equivalence closure* $r^\sim$ of a relation r is defined by the rules

1. $r^\sim(a, a)$
2. $\dfrac{r^\sim(a, b),\ r(b, c)}{r^\sim(a, c)}$

3. $\dfrac{r^{\sim}(a,b),\ r(c,b)}{r^{\sim}(a,c)}$

Again a graphical representation of the rules:



Theorem 2.6. The equivalence closure is transitive. Proof similar to the proof of theorem 2.2.

Theorem 2.7. The equivalence closure is symmetric i.e. $\dfrac{a \ \to_r^{\sim} b}{b \to_r^{\sim} a}$.

Proof. We proof this theorem in 3 steps. First we proof two lemmas and then the theorem by induction.

- Lemma 1: $\dfrac{a \to_r b}{a \to_r^{\sim} b}$. Proof. Assume $a \to_r b$. We get $a \to_r^{\sim} a$ by rule 1 and then $a \to_r^{\sim} b$ by the assumption and rule 2.
- Lemma 2: $\dfrac{a \to_r b}{b \to_r^{\sim} a}$. Proof. Assume $a \to_r b$. We get $b \to_r^{\sim} b$ by rule 1 and then $b \to_r^{\sim} a$ by the assumption and rule 3.
- $\dfrac{a \ \to_r^{\sim} b}{b \to_r^{\sim} a}$ by induction on $a \ \to_r^{\sim} b$.
  1. Goal $a \to_r^{\sim} a$. Trivial by reflexivity.
  2. Goal $c \to_r^{\sim} a$ assuming that $a \to_r^{\sim} c$ is valid by rule 2 of the equivalence closure.
     Premises $a \to_r^{\sim} b$ and $b \to_r c$.
     Induction hypothesis $b \to_r^{\sim} a$.
     We get $c \to_r^{\sim} b$ by the second premise and lemma 2 and then $c \to_r^{\sim} a$ by the induction hypothesis and transitivity of the equivalence closure 2.6.
  3. Goal $c \to_r^{\sim} a$ assuming that $a \to_r^{\sim} c$ is valid by rule 3 of the equivalence closure.
     Premises $a \to_r^{\sim} b$ and $c \to_r b$.
     Induction hypothesis $b \to_r^{\sim} a$.
     We get $c \to_r^{\sim} b$ by the second premise and lemma 1 and then $c \to_r^{\sim} a$ by the induction hypothesis and transitivity of the equivalence closure 2.6.

□

Theorem 2.8. All closures are increasing $r \subseteq r^c$, monotonic $r \subseteq s \Rightarrow r^c \subseteq s^c$ and idempotent $r^{cc} = r^c$ (where the superscript c stands for $+$, $*$ or $\sim$).

Proof. We give a proof for the reflexive transitive closure. The proofs for the other closures are similar.

- Increasing: Goal $r(a,b) \Rightarrow r^*(a,b)$. By rule 1 we get $r^*(a,a)$. The assumption $r(a,b)$ and rule 2 imply $r^*(a,b)$.
- Monotonic: Goal $\dfrac{r \subseteq s,\ r^*(a,b)}{s^*(a,b)}$. Prove by induction on $r^*(a,b)$.
  1. Case $a = b$. Goal $s^*(a,a)$. Trivial by reflexivity of s^*.
  2. Goal $s^*(a,c)$ assuming $r \subseteq s$ and $r^*(a,c)$ is valid because of rule 2. Premises $r^*(a,b)$ and $r(b,c)$. Induction hypothesis $s^*(a,b)$.
     $$\begin{array}{ccccc} a & \to_{r^*} & b & \to_r & c \\ & \Downarrow_1 & & \Downarrow_2 & . \\ a & \to_{s^*} & b & \to_s & c \end{array}$$
     $\Downarrow_1$ is valid by the induction hypothesis. $\Downarrow_2$ is valid by $r \subseteq s$. From the last line and the rule 2 of the reflexive transitive closure we can conclude $s^*(a,c)$.
- Idempotent: The equality of the relations $r^{**} = r^*$ needs a proof of $r^{**} \subseteq r^*$ and a proof of $r^* \subseteq r^{**}$.
  - $r^* \subseteq r^{**}$ is valid because the closure is increasing.
  - Goal $\dfrac{r^{**}(a,b)}{r^*(a,b)}$. Proof by induction on $r^{**}(a,b)$.
    1. Case $a = b$. Goal $r^*(a,a)$. Trivial by reflexivity.
    2. Goal $r^*(a,c)$ assuming $r^{**}(a,c)$ is valid because of rule 2. Premises $r^{**}(a,b)$ and $r^*(b,c)$. Induction hypothesis $r^*(a,b)$.
       $$\begin{array}{ccccc} a & \to_{r^{**}} & b & \to_{r^*} & c \\ & \Downarrow_1 & & \Downarrow_2 & . \\ a & \to_{r^*} & b & \to_{r^*} & c \end{array}$$
       $\Downarrow_1$ is valid by the induction hypothesis. $\Downarrow_2$ is trivial. r^* is transitive. Therefore the last line implies $r^*(a,c)$.

□

Theorem 2.9. A relation s which satisfies $r \subseteq s \subseteq r^c$ has the same closure as r i.e. $r^c = s^c$. Proof:

- $r^c \subseteq s^c$ by monotonicity.

- $s^c \subseteq r^c$: $s^c \subseteq r^{cc}$ by monotonicity and then use idempotence to conclude $s^c \subseteq r^c$.

Terminal Elements

Definition 2.10. The set of *terminal elements* T_r of the relation r is defined by the rule

$$\frac{\left[\dfrac{a \to b}{\bot}\right]}{a \in T_r}$$

where $\bot$ is used to denote a contradiction and the square brackets [and] around the rule above the line indicate that the variables not used outside the bracketed rule are universally quantified in the inner rule. I.e. in order to establish $a \in T_r$ we have to prove $\forall b \,.\, \neg\,(a \to_r b)$ or $\forall b \,.\, a \to_r b \Rightarrow \bot$. Note that the scope of the universal quantification of the variable a spans the whole rule while the scope of the universal quantification of the variable b is just the premise of the rule (i.e. the part above the line).

Theorem 2.11. A terminal element a of a relation r has only trivial outgoing paths: $\dfrac{a \in T_r,\ a \to_r^* b}{a = b}$.

Proof. By induction on $a \to_r^* b$.

1. Goal $a = a$. Trivial.
2. Goal $a = c$ assuming that $a \to_r^* c$ is valid by rule 2. Premises $r^*(a, b)$ and $r(b, c)$. Induction hypothesis $a = b$. Therefore the second premise states $r(a, c)$ which contradicts the assumption $a \in T_r$.

□

Weakly Terminating Elements

Definition 2.12. a is a *(weakly) terminating element* of the relation r if there is a path to a terminal element b i.e. $a \to_r^* b$. The set of *weakly terminating elements* WT_r of the relation r is defined by the rule $\dfrac{a \to_r^* b,\ b \in T_r}{a \in WT_r}$.

Strongly Terminating Elements

Definition 2.13. An object a is *strongly terminating* with respect to the relation r if all paths from a end at some terminal element of r. We define the set of strongly terminating elements ST_r of the relation r by the rule

$$\frac{\left[\dfrac{a \to_r b}{b \in ST_r}\right]}{a \in ST_r}$$

This definition might need some explanation to be understood correctly. Since the premise of the rule is within brackets, all variables not occuring outside the brackets are universally quantified within the brackets (here the variable b).

The rule says that all objects a where all successors b with respect to the relation are strongly terminating are strongly terminating as well. The rule is trivially satisfied by all objects which have no successors i.e. all terminal objects. If the relation r has no terminal objects then there are no initial objects which are strongly terminating.

If there are terminal elements then step by step strongly terminating objects can be constructed by the rule that all successors of them must be strongly terminating (or already terminal). For each constructed strongly terminating object it is guaranteed that all paths starting from it must end within a finite number of steps at some terminal object of the relation.

An object a without successors with respect to the relation r i.e. if there are no b with $a \to_r b$ satisfy the rule, because the premise is satified vacuously. An object a without successors is a terminal element by definition. I.e. all terminal elements are strongly terminating.

2.3 Diamonds and Confluence

In this section we define diamonds and confluent relations.

A relation is called confluent if starting from some object following the relation on different paths of arbitrary length there is always some other object where the two paths meet. This intuitive definition is made precise in the following.

A diamond relation is a kind of a confluent relation where one step different paths can join withing one step.

It turns out that confluence is a rather strong property of a relation. It guarantees that

- all equivalent elements meet at some point
- all paths to terminal elements end up at the same terminal element (i.e. terminal elements are unique)

A diamond relation is a superset of a confluent relation which has already the essential part of confluence. It turns out that a diamond relation is confluent.

First we define formally the diamond property of a relation. The diamond property is intuitively a one step confluence.

Definition 2.14. A relation r is a *diamond* if for all a, b and c there exists a d such that $\begin{array}{ccc} a & \to_r & b \\ \downarrow_r & & \downarrow_r \\ c & \to_r & \exists d \end{array}$ holds.

Note that we use the picture

$$\begin{array}{ccc} a & \to_r & b \\ \downarrow_r & & \downarrow_r \\ c & \to_r & \exists d \end{array}$$

to express the statement

$$\frac{a \to_r b,\ a \to_r c}{\exists d \,.\, b \to_r d \wedge c \to_r d}.$$

The picture notation is more intuitive but not less precise because it can be translated into the corresponding rule notation which can be translated uniquely into a statement of predicate logic.

Definition 2.15. A relation r is *confluent* if r^* is a diamond.

Theorem 2.16. In a confluent relation r all two r-equivalent elements meet at some common element in zero or more steps $\begin{array}{ccc} a & \to_r^\sim & b \\ & \searrow_r^* & \downarrow_r^* \\ & & \exists c \end{array}$.

Proof. By induction on $a \to_r^\sim b$.

1. $a = b$. Trivial. Take $c = a$.
2. Goal $\begin{array}{ccc} a & \to_r^\sim & c \\ & \searrow_r^* & \downarrow_r^* \\ & & \exists e \end{array}$ where $a \to_r^\sim c$ is valid by rule 2. Premises $a \to_r^\sim b$ and $b \to_r c$. Induction hypothesis $\begin{array}{ccc} a & \to_r^\sim & b \\ & \searrow_r^* & \downarrow_* \\ & & \exists d \end{array}$.

$$\begin{array}{ccccc} a & \to_r^\sim & b & \to_r & c \\ \text{Proof} & \searrow_r^* & \downarrow_r^* & & \downarrow_r^*. \ d \text{ exists by induction hypothesis, } e \\ & & \exists d & \to_r^* & \exists e \end{array}$$

exists by confluence.

3. Goal $\begin{array}{ccc} a & \to_r^\sim & c \\ & \searrow_r^* & \downarrow_r^* \\ & & \exists d \end{array}$ where $a \to_r^\sim c$ is valid by rule 3. Premises $a \to_r^\sim b$ and $b \leftarrow_r c$. Induction hypothesis $\begin{array}{ccc} a & \to_r^\sim & b \\ & \searrow_r^* & \downarrow_r^* . \\ & & \exists d \end{array}$

$$\begin{array}{ccccc} a & \to_r^\sim & b & \leftarrow_r & c \\ \text{Proof} & \searrow_r^* & \downarrow_r^* & \swarrow_r^* & . \ d \text{ exists by induction hypothesis.} \\ & & \exists d & & \end{array}$$

□

Theorem 2.17. In a confluent relation all paths from the same object ending at some terminal object end at the same terminal object, i.e.

$$\frac{a \to_r^* b,\ a \to_r^* c,\ b \in T_r,\ c \in T_r}{b = c}$$

.

Proof. Suppose there are two terminal elements b and c with paths starting from the object a. By definition of confluence there must be a d such that $\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r^* & & \downarrow_r^* \\ c & \to_r^* & d \end{array}$ is valid. Since b and c are terminal objects by theorem 2.11 there are only trivial outgoing paths from b and c which implies that $b = c = d$ must be valid. □

Theorem 2.18. A diamond relation is confluent.

Proof. We prove this theorem in two steps.

- Lemma: Let r be a diamond. Then $\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r & & \downarrow_r \\ c & \to_r^* & \exists d \end{array}$ is valid. Proof by induction on $a \to_r^* b$.
  1. Case $a = b$. Trivial, take $d = c$.

$$\begin{array}{ccc} a & \to_r^* & c \\ \downarrow_r & & \downarrow_r \\ d & \to_r^* & \exists f \end{array}$$

2. Goal where $a \to_r^* c$ is valid because of rule
   2. Premises $a \to_r^* b$ and $b \to_r c$. Induction hypothesis
$$\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r & & \downarrow_r . \\ d & \to_r^* & \exists e \end{array}$$
   Proof: $\begin{array}{ccccc} a & \to_r^* & b & \to_r & c \\ \downarrow_r & & \downarrow_r & & \downarrow_r \\ d & \to_r^* & \exists e & \to_r & \exists f \end{array}$. e exists by the induction hypothesis, f exists because $\to_r$ is a diamond.

- Theorem: Let r be a diamond. Then $\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r^* & & \downarrow_r^* \\ c & \to_r^* & \exists d \end{array}$ is valid. Proof by induction on $a \to_r^* c$.
  1. Case $a = b$. Trivial, take $d = c$.
$$\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r^* & & \downarrow_r^* \\ d & \to_r^* & \exists f \end{array}$$
  2. Goal where $a \to_r^* d$ is valid because of rule
     2. Premises $a \to_r^* c$ and $c \to_r d$. Induction hypothesis
$$\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r^* & & \downarrow_r^* . \\ c & \to_r^* & \exists e \end{array}$$
     Proof $\begin{array}{ccc} a & \to_r^* & b \\ \downarrow_r^* & & \downarrow_r^* \\ c & \to_r^* & \exists e \\ \downarrow_r & & \downarrow_r \\ d & \to_r^* & \exists f \end{array}$. e exists by induction hypothesis, f exists by the previous lemma.

□

The last theorem stating that diamonds are confluent gives a way to prove that a relation r is confluent. If r is already a diamond we are ready since a diamond is confluent. If r is not a diamond we try to find a diamond relation s between r and its reflexive transitive closure r^* i.e. a relation s which satisfies $r \subseteq s \subseteq r^*$. From the theorem 2.9 we know that s and r have the same reflexive transitive closure i.e. $r^* = s^*$. Since s is a diamond, s^* is a diamond as well and therefore r is confluent.

In order to find a diamond relation s we can search for rules which are satisfied by r^* and are intuitively the reason which let us assume that r is confluent. Then we can define s inductively as the least relation satisfying the rules and hope that we can prove that s is a diamond with $r \subseteq s$. Note that $s \subseteq r^*$ is satisfied implicitly by this approach since r^* satisfies the rules and s is the least relation satisfying the rules.

3 Lambda Terms

3.1 Basic Definitions

Imagine a mathematical function with one argument which triplicates the argument and adds five to the result. How would you write such a function. In mathematics the most straightforward notation is

$$x \mapsto 3 \times x + 5.$$

The name of the variable x is not important. We could write the same function as $y \mapsto 3 \times y + 5$. The variables x and y are called bound variables because they are bound by the context definining the function.

Now suppose you want to apply the function to an actual argument, say 2. I.e. we want to compute $(x \mapsto 3 \times x + 5)(2)$. We do it by replacing the variable x in the expression $3 \times x + 5$ defining the function by the argument 2 resulting if $3 \times 2 + 5$.

More formally we could write

$$(x \mapsto 3 \times x + 5)(2) \to (3 \times x + 5)[x := 2] = 3 \times 2 + 5.$$

where $\to$ means *reduces to*.

That is already the essence of lambda calculus. In lambda calculus we write the function

$$x \mapsto 3 \times x + 5$$

as

$$\lambda x.3 \times x + 5.$$

and we write function application by juxtaposition

$$(\lambda x.3 \times x + 5)2$$

which reduces to

$$(\lambda x.3 \times x + 5)2 \to_\beta (3 \times x + 5)[x := 2]$$

. The application of the function to an argument is called a β-reduction.

β-reduction is done by variable substitution which can be done purely mechanically i.e. it is the essence of a computation step.

In lambda calculus we have no primitive data types like booleans, numbers pairs etc. There are only functions. However it is possible to

represent data by functions as we shall see later. Data are represented by functions which capture the essence of what can be done by the data.

E.g. boolean values can be used to decide between two alternatives. Therefore a boolean value is represented in lambda calculus by a function with two arguments which chooses the first or second argument depending on its value.

Numbers are represented by functions which take two arguments, a function and a start value and the lambda term representing the number iterates the function n-times on the start value.

Definition of Lambda Terms

Definition 3.1. Let x range over a countably infinite set of variable names $\{x_0, x_1, \ldots\}$ and t over lambda terms, then the set of lambda terms is defined by the grammar

$$t ::= x \mid t\,t \mid \lambda x.t.$$

A lambda term is either a variable x, an application $a\,b$ (the term a applied to the term b) or an abstraction $\lambda x.a$.

We use the convention that application is left associative i.e. abc is parsed as $(ab)c$.

Nested lambda abstractions $\lambda x.\lambda y.\ldots.t$ are parsed as $\lambda x.(\lambda y.\ldots.t)$ and abbreviated as $\lambda xy\ldots.t$.

Free and Bound Variables

In the abstraction

$$\lambda x.t$$

the variable x is a bound variable. It is not visible to the outside world. This is the same convention as used in programming languages which allow the definition of procedures. The formal arguments names of the procedure/function arguments are just visible to the definition of the procedure/function and not to the outside world.

Variables which are not bound by a lambda abstraction are free variables.

Definition 3.2. The set of free variables $FV(t)$ of a lambda term t is defined by

$$FV(t) := \begin{cases} FV(x) & = \{x\} \\ FV(ab) & = FV(a) \cup FV(b) \\ FV(\lambda x.t) & = FV(t) - \{x\} \end{cases}$$

Definition 3.3. A lambda term without free variables is called a *closed lambda term.*

Evidently bound variables can be renamed without changing the meaning of the term. E.g. the two lamda terms

$$\lambda x.x$$

$$\lambda y.y$$

are considered as the same term which represents the identity function. Traditionally the terms are called α-equivalent because you transform one into the other by just renaming bound variables.

We write $t = u$ only if u and t are exactly the same term or α-equivalent terms.

Renaming of bound variables must be done in a way which does not change the structure of the term. The following two rules must be obeyed.

1. Keep different bound variables distinct.

legal:	$\lambda xy.x\,y$	rename to	$\lambda ab.a\,b$
illegal:	$\lambda xy.y$	rename to	$\lambda xx.x$

2. Do not capture free variables.

legal:	$\lambda x.x\,y$	rename to	$\lambda z.z\,y$
illegal:	$\lambda x.x\,y$	rename to	$\lambda y.y\,y$

   The second rename renames the variable x into the variable y which as originally a free variable but captured after the rename.

Variable Substitution

Definition 3.4. The variable substitution $a[x := t]$ is defined by

$$a[x := t] := \begin{cases} x[x := t] & := t \\ y[x := t] & := y \quad \text{for} \quad x \neq y \\ (ab)[x := t] & := a[x := t]\, b[x := t] \\ (\lambda y.a)[x := t] & := \lambda y.a[x := t] \quad \text{for} \quad x \neq y \wedge y \notin FV(t) \end{cases}$$

Note: The condition on the last line is no restriction because we can always rename the bound variable y to a fresh variable z different from x and not occuring free in t since there are infinitely many variables available.

Substitution Swap Lemma The expression

$$a[x := b][y := c]$$

describes the term a where in a first step the variable x is substituted by the term b and then in a second step the variable y is substituted by the term c. Usually it is assumed that x and y are different variables and that x does not occur free in c, i.e. $x \neq y \wedge x \notin FV(c)$.

However two subsequent substitutions do not commute. The term

$$a[y := c][x := b]$$

is in general different from the previous term. Reason: Neither $a[x := b][y := c]$ nor $a[y := c]$ do contain any y. But b might contain y and therefore $a[y := c][x := b]$ might contain y. In order to make the swapping correct we have to do the substitution $b[y := c]$ before substituting the variable x by b.

Theorem 3.5. Substitution Swap lemma: Let $x \neq y$ and $x \notin FV(c)$. Then

$$a[x := b][y := c] = a[y := c]\big[x := b[y := c]\big]$$

. Proof by induction on the structure of a. We use the abbreviations

$$\begin{array}{ll} s_1(a) & := a[x := b][y := c] \\ s_2(a) & := a[y := c]\big[x := b[y := c]\big] \end{array} .$$

1. a is a variable. Lets call it z. Goal $s_1(z) = s_2(z)$
   - $z \neq x \wedge z \neq y$: $s_1(z) = z = s_2(z)$
   - $z = x \wedge z \neq y$: $s_1(z) = b[y := c] = s_2(z)$
   - $z \neq x \wedge z = y$: $s_1(z) = c = s_2(z)$
2. a is the application $t\,u$. Goal $s_1(t\,u) = s_2(t\,u)$. Induction hypotheses $s_1(t) = s_2(t)$ and $s_1(u) = s_2(u)$

$$\begin{array}{rcll} s_1(t\,u) & = & s_1(t)s_1(u) & \text{definition of substitution} \\ & = & s_2(t)s_2(u) & \text{induction hypothesis} \\ & = & s_2(t\,u) & \text{definition of substitution} \end{array}$$

3. a is the abstraction $\lambda z.t$. Goal $s_1(\lambda z.t) = s_2(\lambda z.t)$. Induction hypothesis $s_1(t) = s_2(t)$.

$$\begin{array}{rcll} s_1(\lambda z.t) & = & \lambda z.s_1(t) & \text{definition of substitution} \\ & = & \lambda z.s_2(t) & \text{induction hypothesis} \\ & = & s_2(\lambda z.t) & \text{definition of substitution} \end{array}$$

with appropriate renaming of the bound variable z in order to avoid variable capture (i.e. z must be different from x and y and must not occur free neither in a nor in b).

Beta Reduction Now we are able to define the essential computation step in lambda calculus which is beta reduction. Any term of the form

$$(\lambda x.a)b$$

is called a reducible expression or in short a *redex* which reduces in one step to

$$a[x := b].$$

The redex can appear anywhere inside a lambda term.

Definition 3.6. *Beta reduction* $\rightarrow_\beta$ is a relation defined over lambda terms by the rules

1. $(\lambda x.a)b \rightarrow_\beta a[x := b]$
2. $\dfrac{a \rightarrow_\beta b}{ac \rightarrow_\beta bc}$
3. $\dfrac{b \rightarrow_\beta c}{ab \rightarrow_\beta ac}$
4. $\dfrac{a \rightarrow_\beta b}{\lambda x.a \rightarrow_\beta \lambda x.b}$

Beta reduction $\rightarrow_\beta$ is a one step relation. The expression $t \rightarrow_\beta^+ u$ states that t can be reduced to u in one or more β-reduction steps. The expression $t \rightarrow_\beta^* u$ states that t can be reduced to u in zero or more β-reduction steps.

Two terms t and u are called β-equivalent if $t \rightarrow_\beta^\sim u$ is valid. Recall from section 2 that the equivalence closure 2.5 means that t can be transformed into u by using zero or more beta reduction steps in forward (reduction) or backward (expansion) direction.

Normal Forms

Definition 3.7. A λ-term is in *normal form* if it is a terminal element of the β-reduction relation.

A λ-term is *normalizing* if it is a weakly terminating element of the β-reduction relation.

A λ-term is *strongly normalizing* if it is a strongly terminating element of the β-reduction relation.

In other words

- A λ-term is in normal form if it contains no reducible expression.
- A λ-term t is normalizing if there is a reduction path of zero or more steps $t \rightarrow_\beta^* u$ where u is in normal form.
- A λ-term t is strongly normalizing if all reduction paths end up, after zero or more steps, in some normal form.

Clearly all terms in normal form are trivially normalizing and strongly normalizing.

3.2 Simple Computation with Combinators

In this subsection we demonstrate how lambda terms can be used to do simple computations. We base our terms on combinators which are closed lambda terms i.e. terms without free variables.

The simplest combinator is the identity combinator defined as

$$I := \lambda x.x$$

where I is just an abbreviation for the term on the right hand side of the definition. The lambda calculus does not know the term I, it just knows terms like $\lambda x.x$. We use I for us to formulate the calculus more readable for humans.

The identity function takes one argument and returns exactly the same argument which can be proved by application of the rules for β-reduction

$$\begin{array}{lcll} Ia & = & (\lambda x.x)a & \text{definition of } I \\ & \rightarrow_\beta & x[x := a] & \text{rule 1 of } \beta\text{-reduction} \\ & = & a & \text{definition of substitution} \end{array} .$$

The mockingbird combinator is defined as

$$M := \lambda x.xx.$$

Birdnames are used in this text as the names for combinators to honor Haskell Curry who is one of the inventors of combinatorial logic and who loved to watch birds and to honor Raymond Smullyan who wrote the book *To Mock a Mockingbird* [4] using birds and forests and puzzles about them to teach combinatorial logic in an entertaining and amusing way.

The mockingbird combinator receives one argument and applies it to itself. The term MM has the interesting property to reduce to itself

$$\begin{array}{lcll} MM & = & (\lambda x.xx)M & \text{definition of } M \\ & \to_\beta & (xx)[x := M] & \text{rule 1 of } \beta\text{-reduction} \\ & = & MM & \text{definition of substitution} \end{array}$$

so that we have

$$MM \to_\beta MM \to_\beta MM \ldots$$

which represents the simplest form of an *endless loop* in lambda calculus.

A very important combinator is the kestrel

$$K := \lambda xy.x$$

which receives two arguments and returns the first, easily proved by

$$\begin{array}{lcll} Kab & = & (\lambda xy.x)ab & \text{definition of } K \\ & = & (\lambda x.\lambda y.x)ab & \text{shorthand expanded} \\ & \to_\beta & (\lambda y.x)[x := a]\, b & \text{rule 1 of } \beta\text{-reduction} \\ & = & (\lambda y.a)b & \text{definition of substitution} \\ & \to_\beta & a[y := b] & \text{rule 1 of } \beta\text{-reduction} \\ & = & a & \text{definition of substitution} \end{array}$$

The kestrel shows that λ-terms can in some way *store* values. If we apply the kestrel K only to one argument a we get $\lambda y.a$. This term stores the value a within the abstraction. If later the term receives its second argument it spits out the stored value a ignoring its second argument.

The companion of the kestrel is the kite with the definition

$$K_I := \lambda xy.y$$

which receives two arguments and returns always the second i.e.

$$K_I ab \to_\beta^+ b$$

which can be proved in a similar manner.

A combinator with the same behaviour as the kite can be constructed as an application of the kestrel to the identity function

$$KI$$

where

$$KI \to_\beta \lambda y.I$$

so that we get

$$KIab \to_\beta (\lambda y.I)ab \to_\beta Ib \to_\beta b.$$

Note that KI is not α-equivalent to K_I, but both terms are β-equivalent because they reduce to the α-equivalent terms $\lambda yx.x$ and $\lambda xy.y$.

The kestrel applied to one argument stores the argument a returns it by ignoring the second argument. The trush stores its first argument as well but in a more interesting manner

$$T := \lambda xf.fx.$$

The trush stores its first argument and waits until it receives its second argument. After receiving its second argument it uses the second argument as a function and applies it to the first argument.

Even more interesting in its storage behaviour is the vireo, defined as

$$V := \lambda xyf.fxy.$$

The vireo applied to two arguments stores the arguments (i.e. it stores a pair of values). After receiving its third argument it applies the third argument as a function to the two stored values. Combining the vireo, the kestrel and the kite we can encode pairs. Vab stores the pair (a, b) and $VabK$ returns the first element of the pair and $VabK_I$ returns the second element of the pair i.e. $VabK \to_\beta^+ a$ and $VabK_I \to_\beta^+ b$ which can be proved by using the definitions and applying β-reduction and substitution.

Summary of Combinators Computing with λ-terms is purely mechanical, but it can be tedious if the terms become more complicated. Combinators serve as a kind of abstraction layer to make it easier to manipulate and prove assertions about lambda terms.

Having a definition of an combinator e.g. the kestrel

$$K := \lambda xy.x,$$

the concrete definition is usually not necessery once the crucial property of the kestrel

$$Kab \to_\beta^+ a$$

has been proved to be valid.

In this text we don't use complicated λ-terms. We always use combinators with their corresponding properties to express compactly our claims and proves of these claims.

In the following table the most important combinators together with their specifications and implementations are summarized.

Name	Abbreviation	Specification	Implementation
Bluebird	B	$Bfgx \to_\beta^+ f(gx)$	$\lambda fgx.f(gx)$
Identity	I	$Ix \to_\beta x$	$\lambda x.x$
Kestrel	K	$Kxy \to_\beta^+ x$	$\lambda xy.x$
Kite	K_I	$K_I xy \to_\beta^+ y$	$\lambda xy.y$
Mockingbird	M	$Mx \to_\beta^+ x\,x$	$\lambda x.x\,x$
Starling	S	$Sfgx \to_\beta^+ fx(gx)$	$\lambda fgx.fx(gx)$
Trush	T	$Txf \to_\beta^+ f\,x$	$\lambda xf.f\,x$
Turing	U	$Uxf \to_\beta^+ f(xxf)$	$\lambda xf.f(xxf)$
Vireo	V	$Vxyf \to_\beta^+ fxy$	$\lambda xyf.fxy$

3.3 Confluence - Church Rosser Theorem

A lambda term might contain more than one reducible expression. The β-reduction relation is therefore non-deterministic, you can choose any reducible expression to do a β-reduction step.

Therefore the question arises, if different reduction paths end up at the same result. Is β-reduction confluent?

Remember that confluence is a rather strong property as explained in the section *Inductive Sets and Relations* 2. Confluence guarantees the uniquess of terminal elements (or normal forms in λ-calculus speak) provided that they exist.

In order to prove that β-reduction is confluent we have to prove that $\to_\beta^*$ is a diamond, i.e. that

$$\begin{array}{ccc} a & \to_\beta^* & b \\ \downarrow_\beta^* & & \downarrow_\beta^* \\ c & \to_\beta^* & \exists d \end{array}$$

is valid.

β-reduction is not a diamond If $\to_\beta$ were a diamond we would be ready, because a diamond is confluent as proved in 2.18. Unfortunately $\to_\beta$ is not a diamond for the following reason:

The core of the reduction relation is $(\lambda x.a)b \to_\beta a[x := b]$ where $(\lambda x.a)b$ is the reducible expression. Both subterms a and b might contain further reducible expressions. Since the variable x might be contained in the expression a zero, one or more times all reducible expressions of b can be contained in $a[x := b]$ zero, one or more times.

If b contains a reducible expression a reduction step $b \to_\beta c$ is possible for some c. We have the situation that two reduction paths are possible

$$\begin{array}{lll} (\lambda x.a)b & \to_\beta & a[x := b] \\ \downarrow_\beta & & \\ (\lambda x.a)c & \to_\beta & a[x := c] \end{array} .$$

There are 3 cases:

- Case variable x does not occur in a: Then $a[x := b]$ and $a[x := c]$ are the same expression and since $\to_\beta$ is not reflexive there is no way to complete the diagram.
- Case variable x does occur once in a: Then $a[x := b] \to_\beta a[x := c]$ is a valid reduction step and the diagram can be completed.
- Case variable x does occur 2 or more times in a: Then $a[x := b]$ cannot be reduced to $a[x := c]$ in one step. 2 or more steps are necessary.

Properties of $\to_\beta^*$ As explained in the section *Diamonds and Confluence* 2.3 we can search for properties of $\to_\beta^*$ which indicate that $\to_\beta^*$ is a diamond and use these properties to construct a diamond between $\to_\beta$ and $\to_\beta^*$.

$\to_\beta^*$ has the property that it can do zero or more reduction steps in parallel in *any* subexpression of a lambda expression. I.e. intuitively the following rules are valid:

1. $a \to_\beta^* a$
2. $\dfrac{a \to_\beta^* b}{\lambda x.a \to_\beta^* \lambda x.b}$
3. $\dfrac{a \to_\beta^* b,\ c \to_\beta^* d}{ac \to_\beta^* bd}$
4. $\dfrac{a \to_\beta^* b,\ c \to_\beta^* d}{(\lambda x.a)c \to_\beta^* b[x := d]}$

Although evident by intuition we have to prove these properties.

Theorem 3.8. $\rightarrow_\beta^*$ satisfies $\dfrac{a \rightarrow_\beta^* b \quad c \rightarrow_\beta^* d}{ac \rightarrow_\beta^* bd}$.

Proof by induction on $a \rightarrow_\beta^* b$.

1. Goal $ac \rightarrow_\beta^* ad$ assuming $c \rightarrow_\beta^* d$. Proof by subinduction on $c \rightarrow_\beta^* d$
   (a) Case $c = d$. Trivial by reflexivity.
   (b) Goal $ac \rightarrow_\beta^* ae$. Premises $c \rightarrow_\beta^* d$ and $d \rightarrow_\beta e$. Induction hypothesis $ac \rightarrow_\beta^* ad$.
   $$\begin{bmatrix} c & \rightarrow_\beta^* & d & \rightarrow_\beta & e \\ & \Downarrow_1 & & \Downarrow_2 & \\ ac & \rightarrow_\beta^* & ad & \rightarrow_\beta & ae \\ & & \Downarrow_3 & & \\ ac & & \rightarrow_\beta^* & & ae \end{bmatrix}$$
   $\Downarrow_1$ by induction hypothesis. $\Downarrow_2$ by rule 3 of β-reduction. $\Downarrow_3$ by rule 2 of reflexive transitive closures.
2. Goal $ac \rightarrow_\beta^* ed$. Premises $a \rightarrow_\beta^* b$ and $b \rightarrow_\beta e$. Induction hypothesis $ac \rightarrow_\beta^* bd$.
   $$\begin{bmatrix} a & \rightarrow_\beta^* & b & \rightarrow_\beta & e \\ & \Downarrow_1 & & \Downarrow_2 & \\ ac & \rightarrow_\beta^* & bd & \rightarrow_\beta & ed \\ & & \Downarrow_3 & & \\ ac & & \rightarrow_\beta^* & & ed \end{bmatrix}.$$
   $\Downarrow_1$ by induction hypothesis. $\Downarrow_2$ by rule 2 of β-reduction. $\Downarrow_3$ by rule 2 of reflexive transitive closures.

Theorem 3.9. $\rightarrow_\beta^*$ satisfies $\dfrac{a \rightarrow_\beta^* b \quad c \rightarrow_\beta^* d}{(\lambda x.a)c \rightarrow_\beta^* b[x := d]}$.

Proof in the same manner as the previous theorem with induction on $a \rightarrow_\beta^* b$ and then a subinduction on $c \rightarrow_\beta^* d$ for the reflexive case.

Theorem 3.10. $\rightarrow_\beta^*$ satisfies $\dfrac{a \rightarrow_\beta^* b}{\lambda x.a \rightarrow_\beta^* \lambda x.b}$.

Proof in the same manner as the previous theorems without the need of a subinduction because there is only one premise.

Definition of Parallel β-reduction Now that we have the found the properties of $\rightarrow_\beta^*$ which point into the direction that it is a diamond we can use these properties as rules to define the least relation satisfying these properties.

Definition 3.11. *Parallel beta reduction* $\rightarrow_p$ is a relation defined over lambda terms by the rules

1. $a \rightarrow_p a$
2. $$\frac{a \rightarrow_p b}{\lambda x.a \rightarrow_p \lambda x.b}$$
3. $$\frac{\begin{array}{l} a \rightarrow_p c \\ b \rightarrow_p d \end{array}}{ab \rightarrow_\beta cd}$$
4. $$\frac{\begin{array}{l} a \rightarrow_p c \\ b \rightarrow_p d \end{array}}{(\lambda x.a)b \rightarrow_p c[x := d]}$$

Obviously β-reduction is a subset of parallel β-reduction.

Lemma 3.12. Beta reduction is a subset of parallel beta reduction i.e. $a \rightarrow_\beta b \Rightarrow a \rightarrow_p b$. Proof by induction on $a \rightarrow_\beta b$. Trivial because each rule of $\rightarrow_\beta$ is a special case of some rule of $\rightarrow_p$.

Parallel β-Reduction is a Diamond In order to prove that $\rightarrow_p$ is a diamond we need some lemmas.

Lemma 3.13. Parallel beta reduction preserves abstraction i.e. $\lambda x.a \rightarrow_p c \Rightarrow \exists b : a \rightarrow_p b \wedge c = \lambda x.b$. Proof by induction on $\rightarrow_p$.

1. $c = \lambda x.a$. Trivial. Take $b = a$.
2. $\lambda x.a \rightarrow_p \lambda x.b$ with $a \rightarrow_p b$. Trivial. Take b.
3. The case $\lambda x.a = tu$ is syntactically impossible. Abstraction and application are different.
4. The case $\lambda x.a = (\lambda x.u)v$ is syntactically impossible. Abstraction and application are different.

Lemma 3.14. Basic compatibility of substitution and parallel reduction. $t \rightarrow_p u \Rightarrow a[x := t] \rightarrow_p a[x := u]$. Proof by induction on the structure of a.

1. a is a variable. Goal $z[x := t] \rightarrow_p z[x := u]$. Case $z = x$ is satisfied because of the assumption $t \rightarrow_p u$. Case $z \neq x$ is satisfied by reflexivity $z \rightarrow_p z$.

2. a is an application $b\,c$. Goal $(bc)[x := t] \to_p (bc)[x := u]$

$$\begin{array}{lcll} (bc)[x := t] & = & b[x := t]\,c[x := t] & \text{definition of substitution} \\ & \to_p & b[x := u]\,c[x := u] & \text{ind hypo + rule 3} \\ & = & (bc)[x := u] & \text{definition of substitution} \end{array}$$

3. a is an abstraction $\lambda y.b$. Goal $(\lambda y.b)[x := t] \to_p (\lambda y.b)[x := u]$.

$$\begin{array}{lcll} (\lambda y.b)[x := t] & = & \lambda y.b[x := t] & \text{definition of substitution} \\ & \to_p & \lambda y.b[x := u] & \text{ind hypo + rule 2} \\ & = & (\lambda y.b)[x := u] & \text{definition of substitution} \end{array}$$

Lemma 3.15. Full compatibility of substitution and parallel reduction. $a \to_p c \wedge b \to_p d \Rightarrow a[x := b] \to_p c[x := d]$. Proof by induction on $a \to_p c$.

1. $a = c$. Prove by the previous lemma.
2. Goal $(\lambda y.a)[x := b] \to_p (\lambda y.c)[x := d]$. Premise $a \to_p c$. Induction hypothesis $a[x := b] \to_p c[x := d]$

$$\begin{array}{lcll} (\lambda y.a)[x := b] & = & \lambda y.a[x := b] & \text{definition substitution} \\ & \to_p & \lambda y.c[x := d] & \text{induction hypo + rule 2} \\ & = & (\lambda y.c)[x := d] & \text{definition substitution} \end{array}$$

.

3. Goal $(ae)[x := b] \to_p (cf)[x := d]$. Premises $a \to_p c, e \to_p f$. Induction hypotheses $a[x := b] \to_p c[x := d], e[x := b] \to_p f[x := d]$

$$\begin{array}{lcll} (ae)[x := b] & = & a[x := b]\,e[x := b] & \text{definition substitution} \\ & \to_p & c[x := d]\,f[x := d] & \text{induction hypo + rule 3} \\ & = & (cf)[x := d] & \text{definition substitution} \end{array}$$

.

4. Goal $\big((\lambda y.a)c\big)[x := b] \to_p \big(e[y := f]\big)[x := d]$. Premises $a \to_p e, c \to_p f$. Induction hypotheses $a[x := b] \to_p e[x := d], c[x := b] \to_p f[x := d]$

$$\begin{array}{lcll} ((\lambda y.a)c)[x := b] & = & (\lambda y.a[x := b])\,c[x := b] & \text{definition substitution} \\ & \to_p & e[x := d]\big[y := f[x := d]\big] & \text{induction hypo + rule 4} \\ & = & \big(e[y := f]\big)[x := d] & \text{substitution swap lemma} \end{array}$$

□

Theorem 3.16. Parallel reduction $\rightarrow_p$ is a diamond i.e. $\begin{array}{ccc} a & \rightarrow_p & b \\ \downarrow_p & & \downarrow_p \\ c & \rightarrow_p & \exists d \end{array}$.

Proof by induction on $a \rightarrow_p b$.

1. Trivial reflexive case. $\begin{array}{ccc} a & \rightarrow_p & a \\ \downarrow_p & & \downarrow_p \\ c & \rightarrow_p & c \end{array}$

2. Goal $\begin{array}{ccc} \lambda x.a & \rightarrow_p & \lambda x.b \\ \downarrow_p & & \downarrow_p \\ \lambda x.c & \rightarrow_p & ? \end{array}$. Parallel reduction preserves abstraction. Therefore the specific $\lambda x.c$ instead of the more general c. The premises $a \rightarrow_p b, a \rightarrow_p c$ and the induction hypothesis guarantee the existence of a d such that $\lambda x.d$ is the element to fill the gap.

3. Goal $\begin{array}{ccc} ae & \rightarrow_p & bf \\ \downarrow_p & & \downarrow_p \\ c & \rightarrow_p & ? \end{array}$. Premises $a \rightarrow_p b, e \rightarrow_p f$. Proof by subinduction on $ae \rightarrow_p c$.

   (a) Trivial reflexive case.

   (b) Syntactically impossible

   (c) Goal $\begin{array}{ccc} ae & \rightarrow_p & bf \\ \downarrow_p & & \downarrow_p \\ gh & \rightarrow_p & ? \end{array}$.
   The premises $a \rightarrow_p b, a \rightarrow_p g, e \rightarrow_p f, e \rightarrow_p h$ with the corresponding induction hypotheses guarantee the existence of two element k and m so that km can fill the missing element in the goal.

   (d) Goal $\begin{array}{ccc} (\lambda x.a)e & \rightarrow_p & (\lambda x.b)f \\ \downarrow_p & & \downarrow_p \\ c[x := g] & \rightarrow_p & ? \end{array}$. Parallel reduction preserves abstraction. Therefore the specific $\lambda x.b$ in the upper right corner. The premises $a \rightarrow_p b, a \rightarrow_p c, e \rightarrow_p f, e \rightarrow_p g$ and the corresponding induction hypotheses guarantee the existence of two elements d and h so that $d[x := h]$ fills the missing element in the goal.

4. Goal $\begin{array}{ccc} (\lambda x.a)e & \rightarrow_p & b[x := f] \\ \downarrow_p & & \downarrow_p \\ c & \rightarrow_p & ? \end{array}$. Premises $a \rightarrow_p b, e \rightarrow_p f$. Proof by subinduction on $(\lambda x.a)e \rightarrow_p c$.

(a) Trivial reflexive case.

(b) Syntactically impossible

(c) Mirror image of case 3d, just flipped at the northwest-southeast diagonal.

(d) Goal
$$\begin{array}{ccc} (\lambda x.a)e & \rightarrow_p & b[x := f] \\ \downarrow_p & & \downarrow_p \\ c[x := g] & \rightarrow_p & ? \end{array}$$
The premises $a \rightarrow_p b, a \rightarrow_p c, e \rightarrow_p f, e \rightarrow_p g$ and the corresponding induction hypotheses guarantee the existence of two elements d and h so that $d[x := h]$ fills the missing element in the goal.

□

β-Reduction is Confluent

Theorem 3.17. Beta reduction is confluent. Proof: With the parallel beta reduction$\rightarrow_p$ we have found a diamond relation between beta reduction $\rightarrow_\beta$ and its transitive closure $\rightarrow_\beta^*$. According to the confluence theorems of the chapter "Inductive Sets and Relations" this is sufficient to prove the confluence of beta reduction.

4 Computable Functions

4.1 Boolean Functions

Boolean Values In lambda calculus there are no primitive data types. All lambda terms are functions. If we want to represent boolean values in lambda calculus we have to ask *What can be done with a boolean value?*

Evidently boolean values can be used to decide between alternatives. We can make the convention that the boolean value *true* always decides for the first alternative and the boolean value *false* always decides for the second alternative.

In the section *Lambda Terms* 3 we have already seen the combinator kestrel K which always returns the first argument of two arguments and the kite K_I which always returns the second argument of two arguments. Therefore we use the definitions

$$\begin{array}{rcl} \text{true} & := & K \\ \text{false} & := & K_I \end{array}$$

Definition 4.1. A lambda term t defines a *boolean value* if it reduces to true or false in zero or more steps i.e. if $t \to_\beta^* \begin{cases} \text{true} \\ \text{false} \end{cases}$ is valid.

Boolean Functions

Definition 4.2. A lambda term t represents an n-ary boolean function if given n arguments $b_1, b_2, \ldots b_n$ reduces to a boolean value i.e. if

$$tb_1b_2 \ldots b_n \to_\beta^* \begin{cases} \text{true} \\ \text{false} \end{cases}$$

is valid.

Negation Boolean negation must be a function taking one boolean argument and returning a boolean value which is a two argument function representing a choice.

Remember the vireo which has the specification $Vabf \to_\beta^+ fab$. The term V false true stores the two boolean values and waits for the function to apply it to the two stored values. If we provide a boolean value it selects false in case its value is true and true in case its value is

false. This is exactly the specifaction of boolean negation. Therefore we define

$$(\neg) := V\,\text{false}\,\text{true}.$$

Note that we use the logical symbol $\neg$ to represent the lambda term for boolean negation and we use the same symbol to denote negation on a logical level. It should be clear from the context wheather the lambda term or the logical symbol is meant.

Conjunction The expression $a \wedge b$ where a and b represent boolean values shall return b in case that a represents true and a otherwise. Its nearly trivial to define a lambda expression which has exactly this behaviour

$$(\wedge) := \lambda ab.aba.$$

Disjunction The lambda term representing disjunction can be defined as

$$(\vee) := \lambda ab.aab.$$

You can verify the validity of this definition by applying the definition to all 4 possible cases of the truth table of disjunction.

4.2 Composition of Decomposition of Pairs

In order to represent pairs in lambda calculus we have to ask the question *What can be done with a pair?*. The most natural answer: *Extract either the first or the second element.*

We have seen already the vireo V which stores two values and waits for the third argument to apply the third argument to the first two values. We have the kestrel K to select the first element and the kite K_I to select the second element. Therefore we can define the lambda terms

$$\begin{array}{lcl} \text{pair} & := & V \\ \text{fst} & := & \lambda p.pK \\ \text{snd} & := & \lambda p.pK_I \end{array} .$$

Let us verify that the definitions are correct

$$\begin{array}{lcll} \text{fst}\,(\text{pair}\,a\,b) & = & (\lambda p.pK)(Vab) & \text{definitions} \\ & \to_\beta & (pK)[p := Vab] & \beta\text{-reduction} \\ & = & VabK & \text{substitution} \\ & \to_\beta^+ & Kab & \text{specification of } V \\ & \to_\beta^+ & a & \text{specification of } K \end{array}$$

In the same manner the validity of $\mathrm{snd}\,(\mathrm{pair}\,a\,b) \to_\beta^+ b$ can be verified.

4.3 Numeric Functions

Numbers are represented by lambda terms as iterations. The lambda term c_n representing the natural number n takes two arguments, a function f and a start value a and iterates the function f n times beginning with the start value a.

Definition 4.3. The n iteration of the function f on the start value a is defined as

$$f^n a := \begin{cases} f^0 a & := a \\ f^{n'} a & := f(f^n a) \end{cases}$$

where n' denotes the successor of the natural number n.

Church Numeral

Definition 4.4. The church numeral c_n defined as

$$c_n := \lambda f a.f^n a$$

is the lambda term representing the natural number n.

Definition 4.5. The lambda term t represents a number if it reduces in zero or more steps to a church numeral i.e. if

$$t \to_\beta^* c_n$$

is valid for some n.

Definition 4.6. The term t represents an n-ary numeric function if given n arguments $a_1, a_2, \ldots, a_n$ each one representing a number reduces in zero or more steps to a church numeral i.e. if

$$t a_1 a_2 \ldots a_n \to_\beta^* c_m$$

is valid for some number m.

Definition 4.7. The term t represents an n-ary numeric predicate if given n arguments $a_1, a_2, \ldots, a_n$ each one representing a number reduces in zero or more steps to a boolean value i.e. if

$$t a_1 a_2 \ldots a_n \to_\beta^* \begin{cases} \text{true} \\ \text{false} \end{cases}$$

is valid.

Zero Tester A zero tester is a unary predicate on church numerals. It should return true if the number is the church numeral c_0 and false for all other church numerals.

Remember that church numerals are functions with a function argument f and a start value argument a which iterate the function n times starting at a. Therefore $c_0 f a$ always returns a regardless of the function f. So our zero tester can have the shape

$$\lambda x.x f \,\mathrm{true}$$

which does already the right thing if applied to the argument c_0.

Now we need a lambda term for the iterated function f. The kestrel K applied to two arguments always returns the first argument. If applied only to one argument, it stores the argument and spits it out if it is applied to the second argument. Therefore $K\,\mathrm{false}$ always returns false for any argument. So we can use $K\,\mathrm{false}$ as the function argument f in the above template.

A valid definition of a zero tester is

$$0? := \lambda x.x(K\,\mathrm{false})\,\mathrm{true}.$$

Proof:

$$\begin{array}{lll} 0?\,c_0 & = & (\lambda x.x(K\,\mathrm{false})\,\mathrm{true})c_0 \\ & \to_\beta & (x(K\,\mathrm{false})\,\mathrm{true})[x := c_0] \\ & = & c_0(K\,\mathrm{false})\,\mathrm{true} \\ & \to_\beta^+ & \mathrm{true} \end{array}$$

$$\begin{array}{lll} 0?\,c_{n'} & = & (\lambda x.x(K\,\mathrm{false})\,\mathrm{true})c_{n'} \\ & \to_\beta & (x(K\,\mathrm{false})\,\mathrm{true})[x := c_{n'}] \\ & = & c_{n'}(K\,\mathrm{false})\,\mathrm{true} \\ & = & (K\,\mathrm{false})((K\,\mathrm{false})^n\,\mathrm{true}) \\ & \to_\beta & \mathrm{false} \end{array}$$

Successor Function A lambda term representing the successor function takes a church numeral and returns a church numeral representing the successor of its argument. The signature of the successor function must be

$$\lambda x f a.\cdots$$

where the first argument x is the church numeral and the result must be a church numeral i.e. a function taking two arguments.

We know that the term xfa where x represents a church numeral already does n iterations of f on n. The successor function just has to do one more iteration

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

We prove the desired property $\text{succ}\, c_n \to_\beta^* c_{n'}$ by

$$\begin{array}{lcll} \text{succ}\, c_n & = & (\lambda xfa.f(xfa))c_n & \text{definition} \\ & \to_\beta & \lambda fa.f(c_n fa) & \beta\text{-reduction} \\ & = & \lambda fa.f(f^n a) & \text{definition of } c_n \\ & = & \lambda fa.f^{n'} a) & \text{definition of } f^n a \\ & = & c_{n'} & \text{definition of } c_{n'} \end{array}$$

4.4 Primitive Recursion

The addition of natural numbers is defined recursively as

$$(+) := \begin{cases} 0 + m & := m \\ n' + m & := (n + m)' \end{cases}$$

However lambda calculus does not work on numbers, it works on church numerals representing numbers. Wouldn't it be nice to define a lambda term $(+)$ representing to addition of two lambda terms representing numbers as

$$(+) := \begin{cases} c_0 + m & := m \\ c_{n'} + m & := \text{succ}\,(n + m) \end{cases}$$

And indeed, this is possible. We just have to explain how the definition is mapped into lambda calculus.

Note that, strictly spoken, the expression $a + b$ where a, b and $(+)$ are lambda terms is not a lambda term according to our syntactic definition. In order to be precise we have to define a lambda term "plus" which represents the addition function and write "plus $a\, b$" instead of the expression $a + b$. However the latter is better readable and therefore we use it to denote the more cumbersome expression "plus $a\, b$".

For convenience we use in the following the notation $\mathbf{x}$ to represent n arguments $x_1 x_2 \ldots x_n$ and $g\mathbf{x}$ to represent the function application $gx_1 x_2 \ldots x_n$ i.e.

$$\begin{array}{ll} \mathbf{x} & := x_1 x_2 \ldots x_n \\ g\mathbf{x} & := g x_1 x_2 \ldots x_n \end{array} .$$

Suppose we have a lambda term g representing an n-ary numeric function and a lambda term h representing an n''-ary numeric function. Then in order to define an n'-ary function f we can write

$$f := \begin{cases} fc_0\mathbf{x} & := g\mathbf{x} \\ fc_{n'}\mathbf{x} & := h(fc_n\mathbf{x})c_{n'}\mathbf{x} \end{cases}$$

and interpret this definition as an iteration in the following manner:

- A pair of to church numerals c_i and c_j is used to represent the state of the iteration. The first numeral c_i represent the number of iterations done and the second numeral represents the intermediary result.
- The iteration is started with $p_0 := \text{pair}\, c_0\, (g\mathbf{x})$.
- The step function is represented by the lambda term

  $$s := \lambda p.\text{pair}\, i(hji\mathbf{x})$$

  where $i := \text{succ}(\text{fst}\, p)$ and $j := \text{snd}\, p$.

  The step function extracts the iteration counter and the intermediate result from the pair p and constructs a new pair by applying the successor function to the iteration counter and the function h to the intermediate result and the remaining arguments.
- The function f is defined by the lambda term

  $$f := \lambda y\mathbf{x}.y\, s\, p_0$$

  where y is a church numeral representing the first argument and $\mathbf{x}$ is the array of church numerals representing the remaining n' arguments. Remember that the expression ysp_0 iterates the step function s over its intial value p_0 as long as y is a church numeral.

4.5 Some Primitive Recursive Functions

With the method of the last section we can define a lot of numeric functions.

Addition

$$(+) := \begin{cases} c_0 + m := m \\ c_{n'} + m := \text{succ}\, (c_n + m) \end{cases}$$

Multiplication

$$(\times) := \begin{cases} c_0 \times m := c_0 \\ c_{n'} \times m := m + c_n \times m \end{cases}$$

Exponentiation

$$\bullet^{\bullet} := \begin{cases} a^{c_0} := c_1 \\ a^{c_{n'}} := a \times a^{c_n} \end{cases}$$

Factorial

$$(!) := \begin{cases} c_0! := c_1 \\ c_{n'}! := c_{n'} \times c_n! \end{cases}$$

Predecessor

$$\mathrm{pred} := \begin{cases} \mathrm{pred}\, c_0 := c_0 \\ \mathrm{pred}\, c_{n'} := c_n \end{cases}$$

Difference

$$(-) := \lambda ab.b\, \mathrm{pred}\, a$$

The term $c_n - c_m$ applies the predecessor function m times to c_n. The result is the difference if $n > m$ or c_0 if $n \leq m$.

Comparison

$$(\leq) := \lambda ab.\, 0?\, (a - b)$$

Strict Comparison

$$(<) := \lambda ab.\, \mathrm{succ}\, a \leq b$$

Numeric Equality

$$(\equiv) := \lambda ab.\, a \leq b \wedge b \leq a$$

The symbol $=$ is already reserved for exact equality or α-equivalence of lambda terms. Therefore the symbol $\equiv$ is used to denote the lambda term which represents the equality function for church numerals.

Bounded Minimization Let g be a lambda term representing an n'-ary predicate over church numerals. Then it makes sense to ask, if there is a least number y which makes $gy\mathbf{x}$ true. The expression $\mu^y g\mathbf{x}$ should return this least number or y in case that there is no number strictly below y such that $gy\mathbf{x}$ is satisfied.

$$\mu := \begin{cases} \mu^{c_0} g\mathbf{x} & := c_0 \\ \mu^{c_{n'}} g\mathbf{x} & := \big(\lambda z.\, (gz\mathbf{x})z(\operatorname{succ} z)\big)(\mu^{c_n} g\mathbf{x}) \end{cases}$$

Once $\mu^{c_n} g\mathbf{x}$ satisfies the predicate, the number remains constant throughout the iteration. As long as $\mu^{c_n} g\mathbf{x}$ does not yet satisfy the predicate, the successor is tried until the predicate is satisfied or upper limit is reached.

Division

$$(\div) := \lambda ab.\, \mu^a\, (\lambda x.a < \operatorname{succ} x \times b)$$

This function only works correctly if the divisor b is not zero. It computes the least church numeral x such that $\operatorname{succ} x \times b$ is greater than the church numeral a. In that case the church numeral $x \times b$ is exactly a or leaves some remainder less than b.

Divides Exactly The term $a \mid b$ shall return true if a divides b without remainder, otherwise it shall return false. The definition

$$(|) := \lambda ab. \neg\, a \equiv c_0 \wedge (b \div a) \times a \equiv b$$

satisfies this requirement.

Prime Number Tester

$$\Pr? := \lambda x.\, c_2 \leq x \wedge x \equiv \mu^x(\lambda z.\, c_2 \leq z \wedge z \mid x)$$

The term $\mu^x(\lambda z.\, c_2 \leq z \wedge z \mid x)$ computes the least church numeral z strictly below x which is greater or equal c_2 and divides x exactly. If this number does not exist, the term computes x. In that case x is a prime number.

i-th Prime Number We need a function Pr such that $\mathrm{Pr}\, c_0$ computes c_2, $\mathrm{Pr}\, c_1$ computes c_3, $\mathrm{Pr}\, c_2$ computes c_5 ...

If z is a prime number then there is a prime number between z and $z!'$. We can use this fact to define Pr recursively.

$$\mathrm{Pr} := \begin{cases} \mathrm{Pr}\, c_0 & := c_2 \\ \mathrm{Pr}\, c_{n'} & := \big(\lambda z.\, \mu^{\mathrm{succ}\ z!}(\lambda y.\ \mathrm{Pr}\, y \wedge z < y)\big)(\mathrm{Pr}\, c_n) \end{cases}$$

Prime Exponent If we have a church numeral x we want to be able to compute the exponent e of the ith prime number such that $(\mathrm{Pr}\, i)^e$ divides x exactly. The term $\mathrm{Pr}_{\mathrm{exp}} ix$ shall compute the exponent.

The definition

$$\mathrm{Pr}_{\mathrm{exp}} := \lambda ix.\, \mu^x(\lambda e.\, \neg\, (\mathrm{Pr}\, i)^{\mathrm{succ}\, e} \mid x)$$

satisfies that requirement.

Encode Pairs of Church Numerals into a Church Numeral We can map a pair of natural numbers n and m into another natural number by the formula

$$2^n(2m+1) - 1.$$

It is not too difficult to see that both numbers n and m can be recovered from a number k to which the pair has been mapped i.e. that the mapping is bijective. The number n is the exponent of the prime factor of 2 in $z+1$. Then the number m can be found in an obvious way from $\frac{z+1}{2^n}$.

We have already defined all functions to compute the mapping σ_2 and its two inverses σ_{21} and σ_{22} as lambda terms.

$$\begin{array}{lcl} \sigma_2 & := & \lambda ab.\, {c_2}^a \times (c_2 \times b + c_1) - c_1 \\ \sigma_{21} & := & \lambda x.\, \mathrm{Pr}_{\mathrm{exp}} c_0(\mathrm{succ}\, x) \\ \sigma_{22} & := & \lambda x.\, (\mathrm{pred}\, (\mathrm{succ}\, x \mathbin{\dot{\div}} {c_2}^{\sigma_{21} x})) \mathbin{\dot{\div}} c_2 \end{array}$$

Note that this encoding of pairs of church numerals is completely different from the lambda term pair and its inverses fst and snd. The latter functions encode pairs of arbitrary lambda terms and extract the first and the second component of the pair while the functions σ_2, σ_{21} and σ_{22} perform an arithmetic encoding of pairs of numbers and extract the first and the second number of the pair of numbers.

4.6 General Recursion

The Problem Suppose we have a lambda term g representing an n'-ary predicate and we know that for all arrays of church numerals $\mathbf{x}$ there exists a church numeral y such that $gy\mathbf{x}$ is satisfied.

If we had the ability to program a loop in lambda calculus we could start an unbounded iteration at the church numeral c_0 and stepwise increase the number by one until the predicate g is satisfied.

Unfortunately we just know that a number exists, but we are not able to specify any bound in order to use the bounded μ operator as done in the previous chapter.

Step Function In order to program some unbounded search in lambda calculus we need a function which performs one step in this search, i.e.

- Check if the boolean expression $gy\mathbf{x}$ is satisfied.
- If yes, return y.
- If no, apply a next step on $\operatorname{succ} y$.

The last point would be a recursive call. Unfortunately we have no direct means to perform a recursive call in lambda calculus. Therefore we define the step function s in a way that it receives the function f which does the next step as an argument

$$s := \lambda g\mathbf{x}fy.\,(gy\mathbf{x})y\big(f(\operatorname{succ} y)\big).$$

Turing Combinator Now we need a method to perform the unbounded search. Remember the specification of the turing combinator

$$Uaf \to_\beta^+ f(aaf).$$

If we use instead of the term a the turing combinator U and add an additional argument y we get the potential infinite loop

$$UUfy \to_\beta^+ f(UUf)y \to_\beta^+ f(f(UUf))y \to_\beta^+ \cdots$$

Now let's see what happens if we use $sg\mathbf{x}$ for f and c_m for y:

$$\begin{array}{lll} UUfc_m & \to_\beta^+ & sg\mathbf{x}(UUf)c_m \\ & \to_\beta^+ & (gc_m\mathbf{x})c_m\big(UUf(\operatorname{succ} c_m)\big) \end{array}.$$

The term returns c_m if $gc_m\mathbf{x}$ is satisfied. Otherwise it starts the next iteration on $\operatorname{succ} c_m$.

Unbounded Minimization Having all this, we can define the unbouded μ operator which receives an n'-ary predicate g, an n array of church numerals $\mathbf{x}$ such that the μ operator returns the least church numeral y which satisfies the predicate $gy\mathbf{x}$ if such a term exists

$$\mu := \lambda g\mathbf{x}.\, UU(sg\mathbf{x})c_0$$

where s is the above defined step function.

5 Undecidability

In the section *Computable Functions* 4 we have defined a class of numeric functions and predicates which can be computed in lambda calculus. We call this class of functions/predicates lambda-computable.

Now the question arises: *Are there functions or predicates which are not lambda computable*? The answer to this question is yes. This section of the paper proves that there are undecidable predicates.

In order to prove the existence of undecidable predicates we use sets of lambda terms A which are closed and nontrivial. We make this precise by the following definitions.

Definition 5.1. A set of lambda terms A is *closed* if it contains with each lambda term t also all lambda terms which are α- and β-equivalent to t.

Definition 5.2. A set of lambda terms A is *nontrivial* if it is neither empty nor does it contain all possible lambda terms.

Gödel Numbering Since computable lambda function operate on church numerals we have to transform a lambda term t into a church numeral $\ulcorner t \urcorner$ which is a description of the lambda term t in a way that having the church numeral the corresponding term can be reconstructed. We define $\ulcorner t \urcorner$ by

$$\ulcorner \urcorner := \begin{cases} \ulcorner x_i \urcorner & := \sigma_2 c_0 c_i \\ \ulcorner ab \urcorner & := \sigma_2 c_1 (\sigma_2 \ulcorner a \urcorner \ulcorner b \urcorner) \\ \ulcorner \lambda x_i.a \urcorner & := \sigma_2 c_2 (\sigma_2 \ulcorner x_i \urcorner \ulcorner a \urcorner) \end{cases}$$

Note that $\ulcorner \urcorner$ is not a lambda term. It is easy, but very tedious, to construct by hand for every lambda term t the corresponding lambda term $\ulcorner t \urcorner$. However it is not possible to define a lambda term to do this compilation because the lambda calculus cannot do pattern match on lambda terms i.e. it cannot case split on the way the lambda term t has been constructed. Later we define a lambda term which can compute $\ulcorner c_n \urcorner$ for church numerals c_n.

Definition 5.3. A set A of lambda terms is *decidable* if there is a lambda term p_A representing a unary predicate on church numerals such that $p_A \ulcorner t \urcorner$ returns true if $t \in A$ and false if $t \notin A$.

Self Application Since lambda calculus does not have any restrictions on functions and arguments (they only have to be valid lambda terms) we can apply any lambda term t to its description $\ulcorner t \urcorner$ i.e. the term

$$t \ulcorner t \urcorner$$

is a legal lambda term.

Therefore we can define for all sets of lambda terms A a set B_A by

$$B_A := \{b \mid b \ulcorner b \urcorner \in A\}.$$

We assume that there is a lambda term self which satifies the specification

$$\text{self} \ulcorner t \urcorner \to_\beta^+ \ulcorner t \ulcorner t \urcorner \urcorner.$$

A concrete definition of the lambda term self will be given later.

We can use the lambda term self to see that for every decidable set of lambda terms A the set B_A is decidable as well. Proof: The term

$$\lambda x.\, p_A(\text{self}\, x)$$

is a unary predicate which given a description $\ulcorner b \urcorner$ of a term b decides wheather b is an element of B_A.

Basic Undecidability Theorem

Theorem 5.4. Every closed nontrivial set of lambda term A is undecidable.

Proof. Assume that A is decidable and p_A is a lambda term which decides A.

1. There are the terms $m_0 \in A$ and $m_1 \notin A$, because A is nontrivial.
2. We define the lambda term g by

   $$g := \lambda x. p_A(\text{self}\, x) m_1 m_0$$

   which has the property

   $$g \ulcorner b \urcorner \to_\beta^+ \begin{cases} m_1 & \text{if } b \in B_A \\ m_0 & \text{if } b \notin B_A \end{cases}.$$

3. Assuming $g \in B_A$ leads to a contradition

$$\begin{array}{rcll} g \in B & \Rightarrow & g^{\ulcorner} g^{\urcorner} \rightarrow_{\beta}^{+} m_1 & \text{definition of } g \\ & \Rightarrow & g^{\ulcorner} g^{\urcorner} \notin A & A \text{ is closed} \\ & \Rightarrow & g \notin B_A & \text{defintion of } B_A \end{array}$$

4. Assuming $g \notin B_A$ leads to a contradition

$$\begin{array}{rcll} g \notin B & \Rightarrow & g^{\ulcorner} g^{\urcorner} \rightarrow_{\beta}^{+} m_0 & \text{definition of } g \\ & \Rightarrow & g^{\ulcorner} g^{\urcorner} \in A & A \text{ is closed} \\ & \Rightarrow & g \in B_A & \text{definition of } B_A \end{array}$$

5. Therefore the assumption that A is decidable cannot be valid.

□

Undecidability of Beta Equivalence

Theorem 5.5. Beta equivalence is undecidable.

Proof. Assume that beta equivalence is decidable.

1. Then there is some binary predicate p such that for two lambda terms a and b the term $p^{\ulcorner} a^{\urcorner\ulcorner} b^{\urcorner}$ returns true if they are beta equivalent and false if they are not equivalent.
2. Let A be the set of lambda terms which contains a and all α- and β-equivalent terms. Then by assumption the term $p^{\ulcorner} a^{\urcorner}$ is a decider for the set A.
3. The set A is nontrivial for the following reason: If a is normalizing than A can contain only normalizing terms. Therefore e.g. the term MM where M is the mockingbird combinator which is not normalizing is not in the set. If a is not normalizing then A cannot contain any term in normal form.
4. The set A is closed and nontrivial, therefore it cannot be decidable which contradicts the assumption that there is a decider for β-equivalence.

□

Undecidability of the Halting Problem Like the halting problem for Turing machines there is a halting problem for lambda calculus. The halting problem is solvable if there is a lambda term which determines if another lambda term is normalizing.

Theorem 5.6. It is undecidable whether a lambda term a is normalizing.

Proof. Assume that there is a lambda term p such that $p\ulcorner a\urcorner$ returns true if a is normalizing and false if a is not normalizing.

1. Let A be the set which contains all lambda terms in normal form and all α- and β-equivalent terms. By definition A is closed and it is nontrivial (it contains all variables and it does not contain MM.)
2. The lambda term p would be a decider for A, because a term a must be in this set if it is normalizing.
3. The set A however being closed and nontrivial cannot be decidable which contradicts the assumption that being normalizing is decidable.

□

Implementation of self In the proof of the main undecidability theorem we used a lambda term self with the specification

$$\text{self}\ulcorner a\urcorner \to_\beta^+ \ulcorner a\ulcorner a\urcorner\urcorner.$$

Now we give the still missing implementation of that term. By definition of $\ulcorner\urcorner$ the term we have the equality

$$\ulcorner a\ulcorner a\urcorner\urcorner = \sigma_2 c_1(\sigma_2\ulcorner a\urcorner\ulcorner\ulcorner a\urcorner\urcorner).$$

The only unknown term on the right hand side of the equality is $\ulcorner\ulcorner a\urcorner\urcorner$ which is the description of the church numeral representing the lambda term a.

Any church numeral has the form

$$c_n = \lambda f x.f^n x.$$

Since the names of bound variables are irrelevant we use x_0 for x and x_1 for f. By definition of $\ulcorner\urcorner$ we get

$$\begin{array}{rcl} \ulcorner x_0\urcorner & = & \sigma_2 c_0 c_0 \\ \ulcorner x_1\urcorner & = & \sigma_2 c_0 c_1 \end{array}.$$

With the step function

$$s := \lambda x.\, \sigma_2 c_2 (\sigma_2 \ulcorner x_1 \urcorner x)$$

the term

$$c_n s \ulcorner x_0 \urcorner$$

computes $\ulcorner x_1{}^n x_0 \urcorner$ and the function f defined by

$$f := \lambda x. \sigma_2 c_2 (\sigma_2 \ulcorner x_1 \urcorner x)$$

computes $\ulcorner \lambda x_1.z \urcorner$ given $\ulcorner z \urcorner$ as argument. I.e.

$$f(f(c_n s \ulcorner x_0 \urcorner))$$

computes the church numberal $\ulcorner \lambda x_1 x_0 . x_1{}^n x_0 \urcorner$. The term self can be defined by

$$\text{self} := \lambda x.\, \sigma_2 c_1 (\sigma_2 x (f(f(x s \ulcorner x_0 \urcorner))))$$

References

[1] Turing A.M. On computable numbers, with application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936/7.

[2] Alonzo Church. An unsolvable problem of elementary number theory. *Journal of Mathematics*, 58:354–363, 1936.

[3] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[4] Raymond Smullyan. *To Mock a Mockingbird and Other Logic Puzzles*. Knopf, 1985.