λ-calculus and Combinatory Logic A brief introduction TDDA43

Jan Małuszyński

March 2001

Contents

1	Intr	roduction	3
-	1 1	Motivation	2
	1.1		ປີ
	1.2	Computation as rewriting	4
	1.3	Abstract reduction systems	7
	1.4	Properties of ARSs	8
2	Lambda calculus and combinatory logic		
	2.1	Lambda terms	11
	2.2		10
		I ne rewrite rules	12
	2.3	Rewrite strategies	12 15
	2.3 2.4	The rewrite rules	12 15 16

CONTENTS

Chapter 1

Introduction

These notes present the material discussed at a separate lecture of the course TDDA43 *Programming Theory* in Spring 2000.

1.1 Motivation

The main topic of the course is the semantics of programming language. Semantics of programming languages is often described by (higher-order) functions. A natural question is thus what is a suitable way of specifying computable higher-order functions. In this lecture we present two related formalisms: λ -calculus (alternative spelling: lambda calculus) and combinatory *logic.* Each of them makes it possible to describe computable functions with a small number of primitive concepts. Lambda calculus introduced by Church [3] in 1941 (as a proper formulation of his earlier ideas of 1930's) is based on primitives that closely resemble the mechanisms for defining and calling procedures used in contemporary programming languages, and is therefore of primary importance for the main topic of this course. On the other hand, combinatory logic, originating from Schönfinkel [10](1924) and Curry [5] (1930) is a related formalism that has later played important role in implementation of functional programming languages. Both formalisms describe computation of a function as a *rewriting* process. We give here only a very brief and incomplete introduction to these formalisms. More material can be found in the introductory textbook by Hindley and Seldin [6] and in the monograph by Barendregt [1].

Before presenting the formalisms we discuss first some notions related with this view of computations. We will refer to them when presenting the formalisms.

1.2 Computation as rewriting

An introductory example

The view we will adopt is that computation can be thought of as making transformations in a state space, where one step of computation corresponds to a move from one state to another. When a final state is reached, the computation halts and the result is presented. Describing this a little bit differently, computation could be viewed as successive *rewriting* of *expressions* into other expressions, determined by a set of *rewrite rules*. When no rewrite rule is applicable to an expression, the expression is said to be a *normal form*, and considered to be the result of the computation. This process is nondeterministic in general, since several rewrite rules may be applicable to a given expression, yielding different resulting expressions.

The rewriting view on computation seems particularly appropriate when dealing with problems of the type: "Simplify the expression e as much as possible". For example the polynomial expression (0 + a)(a + b) simplifies to $a^2 + ab$, using rules as

$$\begin{array}{l} 0+x \rightarrow x\\ 0\cdot x \rightarrow 0\\ x(y+z) \rightarrow xy + xz\\ (x+y)z \rightarrow xz + yz \end{array}$$

Note that there are many possible reductions from the initial expression to the result, two of which are: 1

$$(0+a)(a+b) \to a(a+b) \to a^{2} + ab$$

$$(0+a)(a+b) \to (0+a)a + (0+a)b \to (0\cdot a + a^{2}) + (0+a)b \to (0+a^{2}) + (0+a)b \to a^{2} + (0+a)b \to a^{2} + (0+a)b \to a^{2} + ab$$

For the expression considered (i) all reductions are finite, and (ii) the result will always be the same, no matter how the reduction is carried out. If these two desirable and important properties holds for any expression, the rewrite rules are called (i) *terminating* and (ii) *confluent*. However, these properties are in no way guaranteed to hold for an arbitrary set of rewrite rules. Even our example rules are not confluent, since the expression x(0+y) can be simplified either to xy or to $x \cdot 0 + xy$, depending on the choice of the used rules.

The view of computation as rewriting is common in the theory of computation, as illustrated by the following examples.

¹We assume here that the operators have the usual priorities so that the superfluous parantheses in the expression $a^2 + (ab)$ can be omitted.

1.2. COMPUTATION AS REWRITING

Turing machines

The Turing machine (TM), conceived by Alan Turing in 1936 [11], is a simple but powerful mathematical model of a computational device. A TM has a finite set Q of states, and a storage device consisting of a one-dimensional array (a *tape*) of cells, each of which can contain a symbol out of a finite set Σ , that includes a distinguished *blank symbol*. Associated with the tape is a read/write head that can move right or left on the tape and that can read and write a single symbol on each move. The tape is infinitely long in both directions.



Figure 1.1: A snapshot of a Turing machine at work

When the computation starts, the TM is in its *initial state*, the tape contains a finite string of non-blank symbols placed in consequtive cells, while all other cells contain blank symbols, and the head is scanning the leftmost symbol of the string. The machine's actions are determined by its program, which is given as a partial function δ from $Q \times S$ to $Q \times S \times \{left, right\}$. Thus if the machine is in state q, and the read/write head is scanning the symbol s, then the program specifies the next state, the symbol to repalce s on the tape, and the direction in which to move the read/write head. If $\delta(q, s)$ is undefined, then the machine halts in an error state. If a *final state* is reached, the computation terminates and the contents of the tape is the result of the computation. Notice that at every step of a computation only a finite number of cells contains non-blank symbols; the contents of the tape may thus be described by a finite string.

At any given moment, the TM's configuration is described by the contents on the tape and the current state. Suppose that the contents of the tape is $s_1 \dots s_{i-1} s_i \dots s_n$, the read/write head is scanning the symbol s_i , and the TM is in state q. Then we let the string

$$s_1 \ldots s_{i-1} q s_i \ldots s_n$$

represent the configuration of the machine. Now if the program contains the instruction $\delta(q, s_i) = (q', t, left)$, then the machine may change its configuration according to the rewrite rule

$$s_1 \dots s_{i-1} q s_i \dots s_n \rightarrow s_1 \dots s_{i-2} q' s_{i-1} t \dots s_n$$

If the program contains the instruction $\delta(q, s_i) = (q', t, right)$, then the corresponding rewrite rule is

$$s_1 \dots s_{i-1} q s_i \dots s_n \rightarrow s_1 \dots s_{i-1} t q' s_{i+1} \dots s_n$$

The renowned *Church-Turing's hypothesis* (also known as Church's thesis) states that these functions are indeed all functions that are computable in the intuitive sense.

Context-free grammars

Context-free grammars play a fundamental role in string-processing applications, notably in the definition of programming languages, parsing of programming languages and natural languages, etc. The origin of the formalism can be found in Chomsky [2].

A context-free grammar is a specification of a (context-free) language (over some alphabet Σ) by means of *production rules* of the form:

 $A \to \alpha$

where A belongs to a set V of non-terminals, and α is a string in $(\Sigma \cup V)^*$. We assume that V and Σ are disjoint.

Suppose β_1 and β_2 are (possibly empty) strings in $(\Sigma \cup V)^*$. Then $\beta_1 A \beta_2$ derives $\beta_1 \alpha \beta_2$, using the production rule above. We write this as

$$\beta_1 A \beta_2 \Rightarrow \beta_1 \alpha \beta_2$$

If it is possible to reach a string $\gamma \in \Sigma^*$, starting from the special start non-terminal S, and by using a sequence of derivation steps, then γ belongs to the language generated by the grammar. As an example, we give a grammar that generates all strings over $\{a, b\}^*$ with an equal number of a's and b's. Here S is the only non-terminal, and ϵ denotes the empty string.

$$S \to SS$$

$$S \to \epsilon$$

$$S \to aSb$$

$$S \to bSa$$

For instance, the string *abba* can be derived as follows:

 $S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abbSa \Rightarrow abba$

We have used the first, third, second, fourth and second rules of the grammar in the different steps.

1.3 Abstract reduction systems

An abstraction based on the examples of the preceding sections gives the following definition:

Definition 1.1 An abstract reduction system (ARS) is a pair (A, \rightarrow) , where A is a set, and each \rightarrow is a binary relation on A, called a *rewrite relation*. \Box

Notation: If $a, b \in A$ and $(a, b) \in \rightarrow$, then we write this as $a \to b$ and call b a reduct of a. The reflexive and transitive closure of \rightarrow is written as \twoheadrightarrow (a usual notation R^* for the transitive and reflexive closure of the relation R may also be used). \rightarrow^i denotes reduction in i steps. The inverse relation of $\rightarrow (\twoheadrightarrow)$ is denoted by \leftarrow or \rightarrow^{-1} (\ll or \rightarrow^{-1}). If a and b are identical (i.e. they denote the same element in A), we write $a \equiv b$. a and b are convertible (a = b) if a can be rewritten into b using the \rightarrow relation forwards or backwards a finite number of times. Thus, we let = be the relation ($\rightarrow \cup (\sim)^*$.

Figure 1.2 shows an example rewrite relation, in which a and b are convertible.



Figure 1.2: a = b

Definition 1.2 Given an ARS (A, \rightarrow) , $a \in A$ is a normal form if there is no b such that $a \rightarrow b$. We write NF(A) to denote the set of normal forms in A

(w.r.t. \rightarrow). *a has* a normal form if there is a $b \in NF(A)$ such that $a \twoheadrightarrow b$ (we say that *b* is a normal form of *a*).

Note that an element may have zero, one, or many normal forms.

The operational intuition behind the relation(s) \rightarrow is that if $a \rightarrow b$, then a can be *reduced to b*. As shown by the examples in the previous chapter, our aim is often to reduce (in a number of steps) a given element to (one of) its normal forms, using the rewrite relation. This naturally raises questions about existence and uniqueness of normal forms; questions we will address in the next section.

1.4 Properties of ARSS

Throughout this section, we assume that an ARS (A, \rightarrow) is given.

Definition 1.3

- (1) \rightarrow is weakly normalizing (WN) if every $a \in A$ has a normal form.
- (2) \rightarrow is strongly normalizing (SN) (or terminating, or Noetherian) if there is no infinite sequence $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots$

If \rightarrow is weakly normalizing, then we can reduce every element to a normal form. Practically however, the computation might not terminate if the wrong reduction strategy is used. In the example below, the reduction sequence $a_0 \rightarrow a_1 \rightarrow \ldots$ is infinite.



Figure 1.3: A rewrite relation which is WN but not SN

If \rightarrow is terminating, then every reduction sequence will eventually end in a normal form. Generally, there may be many normal forms. We are interested in computing values of functions. The following notion gives a sufficient condition for an ARS under which each element has at most one normal form. This is important for our purposes, since we want to represent function calls as elements of ARS's, and the corresponding normal forms as the computed results of these calls. Hence we cannot admit *ars*'s where the normal forms are not unique.

Definition 1.4

- (1) \rightarrow is Church-Rosser (CR) (or confluent) if for all $a, b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$, there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.
- (2) \rightarrow is weakly Church-Rosser (WCR) (or weakly confluent) if for all $a, b, c \in A$ such that $a \rightarrow b$ and $a \rightarrow c$, there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.

The above definition is probably easiest understood in diagram format (see Fig. 1.4).



Figure 1.4: Schematic description of the CR and WCR properties (dashed arrows are existentially quantified)

Theorem 1.5 If \rightarrow is confluent, then the normal form of every element (if it exists) is unique (i.e. if $a \twoheadrightarrow b$, $a \twoheadrightarrow c$, and $b, c \in NF(A)$, then $b \equiv c$).

The proof is left as an exercise.

The distinction between confluence and weak confluence is subtle but important. The following example shows the difference.

Example 1.6 Consider the rewrite relation defined by the following diagram:



Figure 1.5: A rewrite relation which is WCR but not CR

By a simple case analysis we see that \rightarrow is weakly confluent. For instance, b can be reduced to both a and c, but $a \rightarrow a$ and $c \rightarrow a$. However, \rightarrow is not confluent, since $b \rightarrow a$ and $b \rightarrow d$, and a and d have no common reduct. \Box

For terminating rewrite relations, confluence and weak confluence are equivalent. This is a consequence of the following fundamental theorem, first stated by Newman [9]. We quote it without proof.

Theorem 1.7 [Newman's lemma] If \rightarrow is weakly confluent and terminating, then \rightarrow is confluent.

Notice that a confluent ARS $\mathcal{A} = (A, \rightarrow)$ defines a partial function $f_{\mathcal{A}}$ on A:

$$f_{\mathcal{A}}(a) = \begin{cases} b, & \text{if } b \in NF(A) \text{ and } a \twoheadrightarrow b \\ \text{undefined else} \end{cases}$$

In addition:

- if \rightarrow is weakly normalizing, then f is a total function.
- if \rightarrow is strongly normalizing, then f can be computed with any reduction strategy.

In the next chapter we survey *lambda calculus* and *combinatory logic* as ARS's: defining functions in terms of reduction to normal forms.

Chapter 2

Lambda calculus and combinatory logic

This chapter introduces lambda calculus and combinatory logic as rewrite systems and relates them to each other.

2.1 Lambda terms

As any rewrite system the lambda calculus defines a domain of expressions and a rewrite relation on this domain. The expressions are called *lambda terms*. Intuitively, lambda terms represent functions on lambda terms. This may seem somewhat confusing, since one might have expected first a domain of elementary values on which an hierarchy of first- and higher-order functions could be defined. However, as will be discussed later, the elementary values such as natural numbers can easily be coded as functions.

The functional intuition is reflected by the syntax of lambda terms. A basic lambda term is a variable. The role of variables in lambda calculus can be compared to the role of the variables used in procedures of programming languages. Certain kinds of expressions in programming languages, e.g. arithmetic expressions like x + y can be transformed into procedures by identifying some of the variables as formal parameters.

For example, in ML the expression $2 \times x$ is transformed into a unary function twice with a formal parameter x in the following way:

fun twice $x = 2^*x;$

A similar kind of syntactic construct is defined for lambda terms and it is called *functional abstraction*. A difference is that the expression above gives the name twice to the created function, while, as we will see below, the functional abstraction gives no name to the created function.

The function twice can then be *called* with an argument, represented typically by an arithmetic expression, e.g twice(2). A similar kind of syntactic construct is defined for lambda terms and it is called *application*.

The call twice (2) in ML will produce the value 4 as a result. In lambda calculus an application expression can be reduced and the reduction may produce a normal form, which will be considered the computed result. Encoding of twice in lambda calculus will be discussed later.

Formally, the lambda terms are defined as follows:

- (i) a variable x, y, z, \ldots is a lambda term;
- (ii) if M and N are lambda terms, then (MN) is a lambda term, (called *functional application*);
- (iii) if M is a lambda term, then $(\lambda x.M)$ is a lambda term (called *functional abstraction*).

A term of the form (ii) represents the application of a function M to N. A term of the form (iii) makes the term M into a unary function; $(\lambda x.M)$ represents the function of one variable x.

A commonly used notational convention allows for suppression of some parantheses. Application is understood to associate to the left, so we can write xyz instead of ((xy)z). The notation $\lambda x.PQ$ is used instead of $(\lambda x.(PQ))$ and $\lambda x_1 x_2 ... x_n.M$ instead of $(\lambda x_1.(\lambda x_2.(...(\lambda x_n.M)...)))$.

2.2 The rewrite rules

In a lambda term of the form $\lambda x.M$, every occurrence of x in M is said to be *bound*. If an occurrence of x is not bound in M, then it is said to be *free* in M.

The rewrite rules of lambda calculus make an essential distinction between free and bound variable occurrences. Intuitively the bound occurrences play a role similar to that of the formal parameters of procedures in programming languages, while the free occurrences can be compared to global variables in the procedure. The essential mechanism used in rewriting is the operation of simultaneous replacement of all free occurrences of a variable x in a term M by a term N. The operation should be defined with care since it should not affect the status of the occurrences of the remaining variables. In particular, free occurrences should not be changed to bound occurrences. Such a danger exists if M and N share some variables. Problem can be avoided by renaming some of the bound variables in the case of conflicts.

The commonly adopted solution can be presented as the following recursive definition of the replacement operation. We define $[x \mapsto N]M$ to be the lambda term obtained from M as follows:

- (i) $[x \mapsto N]x$ is N,
- (ii) $[x \mapsto N]y$ is y for any variable y different from x,
- (iii) $[x \mapsto N](PQ)$ is $([x \mapsto N]P[x \mapsto N]Q)$,
- (iv) $[x \mapsto N] \lambda x.P$ is $\lambda x.P$,
- (v) $[x \mapsto N]\lambda y.P$ is $\lambda y.[x \mapsto N]P$ if y is different from x and has no free occurrence in N, or if x has no free occurrence in P,
- (vi) $[x \mapsto N] \lambda y.P$ is $\lambda z[x \mapsto N]([y \mapsto z]P)$ if y is different from x and has a free occurrence in N, and x has a free occurrence in P. z is a variable that does not occur free in N nor in P.

The cases (i) and (ii) above are basic ones, when no conflict is possible. Replacement in the term which is a functional application is defined by case (iii). In that case the replacement of x is to be made independently in both component terms. The difficulties may only concern functional abstraction, which is handled by the cases (iv), (v) and (vi). Intuitively, the replacement concerns only the free occurrences of x in M. If M is of the form $\lambda x.P$ (case (iv)), there is no free occurrence of x in M hence the operation leaves Munchanged. Case (v) identifies the conflict-free situations. Replacing x by Nin $\lambda y.P$ would have changed the status of free occurrences of y in N. This cannot happen if y does not appear free in N. On the other hand, if x does not appear free in P the replacement will not change P, hence will not change the status of any variable occurrence.

The problem may only arise if y occurs free in N and x occurs free in P. For example take the term y as N and the term xy as P. In that case M is of the form $\lambda y.xy$. A direct replacement of x by N in M would have resulted in the term $\lambda y.yy$, thus changing the status of y in N. To avoid this, case (vi) requires that the bound variable in M is renamed, e.g. to z. The effect of the operation is thus the term $\lambda z.yz$, and the status of y in N remains unchanged after the replacement.

14 CHAPTER 2. LAMBDA CALCULUS AND COMBINATORY LOGIC

The rewrite relation of lambda calculus is defined by the α - and β -rules:

- (α) If y is not free in M, then $\lambda x.M \to_{\alpha} \lambda y.[x \mapsto y]M$.
- (β) $(\lambda x.M)N \rightarrow_{\beta} [x \mapsto N]M$

For example:

$$(\lambda x.xx)(\lambda y.y)z \to (\lambda y.y)(\lambda y.y)z \to (\lambda y.y)z \to z$$

The intuition of the α -rule is that the name of a bound variable is irrelevant. We have seen already that renaming is used in the definition of the replacement operation. The restriction in the α -rule requires that "fresh" variables are used for renaming to avoid confusion with the free variables already present in the term, which in that case would become bound by this transformation. For example, the variable x in the term $\lambda x.xy$ cannot be renamed to y. It should be noticed that the rewrite relation $\twoheadrightarrow_{\alpha}$ is symmetric. This means that one need not to distinguish between "forward" and "backward" application of the α -rule. Any "backward" rewriting of a given term can also be achieved by a "forward" rewriting. For this reason the α rule is called α -conversion rule in accordance with the terminology of Section 1.3. The lambda terms X and Y such that $X \twoheadrightarrow_{\alpha} Y$ are called α -congruent terms.

The β -rule captures the intuition of computation by rewriting. It can be applied to any term which has the form of functional application, where the first component is a functional abstraction, representing the function applied and the other is the actual argument. Such a term, which may appear within a larger lambda term, is called a *redex*. Intuitively, redexes resemble procedural calls in programming languages. The β -reduction can be compared with invocation of a procedure with one argument. The computation step consists in removing the procedure header and replacing every occurrence of the (only) formal parameter by the actual parameter of the call. As a result a new lambda term is obtained which may also include redexes.

This process may or may not terminate. A normal form obtained upon termination can be seen as the value of the function for the actual argument. Generally, a lambda term can have more than one redex, and can thus be subject to more than one reduction strategy. For the above intuition to be valid, a terminating computation in any strategy should give the same result. Otherwise, the term having different normal forms could not be considered as a function. Note that the formal definition of β -reduction uses the replacement operation, which in certain cases may cause renaming. Therefore the result of the β -reduction is only unique up to renaming of the bound variables. The uniqueness of normal form follows from the famous *Church-Rosser theorem*[4] that states confluence of lambda calculus up to α -conversion. We quote the result without proof which can be found e.g. in [6]. There are several proofs of this result in the literature but all of them are quite technical and long.

Theorem 2.1 If $P \twoheadrightarrow_{\beta} M$ and $P \twoheadrightarrow_{\beta} N$ then there exist α -congruent terms T_1 and T_2 such that $M \twoheadrightarrow_{\beta} T_1$ and $N \twoheadrightarrow_{\beta} T_2$.

Consequently, every lambda term has a unique (up to α -conversion) normal form, or no normal form at all. The latter case is due to the fact that lambda calculus is not a terminating ARS. For example the term $(\lambda x.xx)(\lambda x.xx)$ is a redex and it reduces to itself.

The Church-Rosser theorem justifies the intuition of viewing lambda terms as partial functions. A lambda term X can be seen as a function on lambda terms. If its application (XY) to another lambda term Y reduces to a normal form N then N is unique and can be seen as a value of the function X on the argument Y. The function is undefined on the arguments for which no normal form exists.

2.3 **Rewrite strategies**

The two most important strategies are *normal-order* reduction, where the leftmost redex is chosen, and *applicative-order* reduction, where the leftmost innermost redex is chosen. Consider a lambda term which is a redex. As discussed above, such a term resembles a call of a procedure in a programming language. The normal-order reduction step will thus replace every occurrence of the formal parameter in the body by the (unchanged) actual parameter. But the actual parameter may be/include another redex. Thus, under this strategy the computation of the value of the actual parameter is postponed. This resembles the mechanism of procedure invocation known as "call-by-name" or "lazy evaluation", since the computation of the argument is deferred as long as possible. The situation is different in the applicative-order strategy. Here the normal forms of the nested redexes are computed starting from the innermost level. This resembles the "call-by-value" mechanism of procedure invocation where the values of the actual parameters are computed first.

Consider for example the term $(\lambda x.xx)((\lambda y.y)(\lambda y.y))$. Its normal-order reduction goes as follows:

 $\begin{array}{l} (\lambda x.xx)((\lambda y.y)(\lambda y.y)) \rightarrow \\ ((\lambda y.y)(\lambda y.y))((\lambda y.y)(\lambda y.y)) \rightarrow \\ (\lambda y.y)((\lambda y.y)(\lambda y.y)) \rightarrow \\ ((\lambda y.y)(\lambda y.y)) \rightarrow \\ \lambda y.y \end{array}$

On the other hand, the applicative-order reduction requires fewer steps to obtain the same normal form :

$$\begin{array}{l} (\lambda x.xx)((\lambda y.y)(\lambda y.y)) \rightarrow \\ (\lambda x.xx)(\lambda y.y) \rightarrow \\ ((\lambda y.y)(\lambda y.y)) \rightarrow \\ \lambda y.y \end{array}$$

Termination may depend on the reduction strategy. Consider for example the term Y of the form $(\lambda y.yy)(\lambda y.yy)$. It has only one β -redex and the reduction step rewrites the term into itself. Hence Y has no normal form. Consider also the term Z of the form $\lambda x.\lambda z.z$. Application of Z to any term T can be β -reduced in one step to the term $\lambda z.z$ regardless of the form of T since x does not appear in $\lambda z.z$. Consider now the term ZY. it has two redexes: ZY and Y. The normal-order reduction will in one step produce the normal form $\lambda z.z$. On the other hand, the applicative-order reduction will keep reducing Y to Y in ZY, hence it will not terminate.

An important result is that if the normal form exists, it is reachable by a finite normal-order reduction sequence.

2.4 The power of lambda calculus

This section shows that both natural numbers and functions on natural numbers can be represented as lambda terms.

Church proposed to represent a natural number n by the term $\lambda xy.x^n y$, where $x^n y$ denotes n applications of x to y, i.e. the term x(x...(xy) including n ocurrences of x. Such terms are called *Church numerals*. In the sequel we will abbreviate $\lambda xy.y$ as $\mathbf{0}$, $\lambda xy.xy$ as $\mathbf{1}$, etc. Notice that a Church numeral is in normal form. It can also be seen as a binary function on lambda terms: for given terms M and N the term $\mathbf{n}MN$ reduces to M^nN .

An *m*-ary function ϕ on natural numbers is said to be *lambda-defined* by a lambda term *F* iff for arbitrary natural numbers $n_1, ..., n_m$

$$F\mathbf{n_1} \dots \mathbf{n_m} \twoheadrightarrow \mathbf{n} \text{ iff } \phi(n_1, \dots, n_m) = n.$$

A function is *lambda-definable* if it is lambda-defined by some lambda term.

A classical result is that the class of lambda-definable functions is exactly the class of the functions defined by Turing machines. Thus, lambda calculus can be used as a foundation of computation theory.

On the other hand, some concepts of lambda calculus turn out to be useful in the context of programming languages. As already pointed out, the β reduction mechanism resembles parameter passing in programming languages. Another interesting observation is that all functions represented by lambda terms have fixed points. More precisely, for every term M there exists a term P such that $MP =_{\beta} P$. Moreover, there exists a lambda term Y such that YM is a fixpoint of M for every M. In particular consider Y defined as $\lambda x.((\lambda y.x(yy))(\lambda y.x(yy)))$. Then for any M

 $\begin{array}{l} YM = \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))M \rightarrow \\ (\lambda y.M(yy))(\lambda y.M(yy)) \rightarrow \\ M((\lambda y.M(yy))(\lambda y.M(yy))) = M(YM) \end{array}$

Hence $M(YM) =_{\beta} YM$. Thus YM is a fixpoint of M. This has some relevance for the theory of programming languages, where semantics is often defined in terms of fixpoints.

In the pure version of lambda calculus discussed here, the functions are defined from very few primitive concepts. For this reason, even relatively simple functions can only be coded by rather complex lambda terms, and computation of their values requires a large number of reduction steps. In this sense, the expressive power of pure lambda calculus is rather low. We illustrate this statement by an example.

The first step towards definition of the arithmetic operations is to define the successor function s by a lambda term s such that $\mathbf{sn} \rightarrow \mathbf{n} + \mathbf{1}$. A solution proposed in the literature is to define s as $\lambda uxy.x(uxy)$. We get:

$$\begin{aligned} \mathbf{sn} &= \lambda uxy.x(uxy)(\lambda zv.z^n v) \rightarrow \\ \lambda xy.x((\lambda zv.z^n v)xy) \rightarrow \\ \lambda xy.x((\lambda v.x^n v)y) \rightarrow \\ \lambda xy.x(x^n y) &= \mathbf{n} + \mathbf{1} \end{aligned}$$

Thus, computation of the successor is not too complicated. Now the addition can be expressed in terms of the successor. Knowing that $\mathbf{n}xy \twoheadrightarrow x^n y$, and that $x + y = s^x s^y(0)$ we can define addition by the lambda term $\lambda xy.x\mathbf{s}(y\mathbf{s0})$. In the extended form it is:

 $\lambda xy.x(\lambda uxy.x(uxy)(y(\lambda uxy.x(uxy))(\lambda xy.y))).$

This term is indeed not that easy to grasp and following the reduction steps needed to compute 2+2 would definitely be beyond one's patience.

Another way of defining addition is by the following recursive definition

$$plus(0, y) = y$$

$$plus(s(x), y) = s(plus(x, y))$$

This is a special case of the use of the *primitive recursion* scheme

 $\phi(0, y) = \psi(y)$ $\phi(s(x), y) = \theta(x, \phi(x, y))$

defining a function ϕ on natural numbers in terms of given functions ψ and θ . It turns out that such a scheme can be represented by a lambda term, which is however quite complex. We refer for details e.g. to [6].

While the pure lambda calculus is too primitive for defining functions in practice, the β -reduction is very close to procedure invocation mechanism in high-level languages. We may enrich the pure calculus by a number of predefined data types. The theory tells us that in principle it is possible to compile them into the pure calculus, but we can implement them in some alternative way. Taking this approach we can define a realistic programming language starting from more convenient higher-level primitives and use lambda calculus as a basis for defining its semantics. Understanding of this relation between procedures and lambda calculus has had a big impact on the design and implementation of programming languages, starting from ALGOL 60 design in late fifties (see e.g. [8]).

2.5 Combinatory logic

This section presents yet another rewriting system, called the *combinatory logic*. A motivation behind it is to represent functions without the technical difficulties related to the use of bound variables in lambda calculus. The combinatory logic has only free variables, so that it does not suffer from these difficulties. Still it has the computational power equivalent to that of lambda calculus. However, it does not seem to have as clear intuition as the lambda calculus. The combinatory logic plays an important role in the implementation of functional languages. This topic is, however, outside of the scope of these notes.

Combinatory terms

We define first the domain of the rewriting system. Its elements will be called *combinatory logic terms*, or briefly *combinatory terms*.

They are constructed by *application* from the basic elements which are the variables and the two *basic combinators* K and S. Thus, formally the combinatory terms are defined as follows:

- (i) A variable is a combinatory term,
- (ii) The basic combinators K and S are combinatory terms,
- (iii) If X and Y are combinatory terms, then (XY) is a combinatory term.

For example, ((SK)K) and (((SK)K)x) are combinatory terms. A commonly used notation is to omit parantheses following the convention of association to the left. For example, the term ((SS)(K((SK)K))) will be represented as SS(K(SKK))

A combinatory term not including variables is called a *combinator*. The intuition of combinators is to represent operators on functions, e.g. composition of functions, identity operator, etc. Formally the transformation corresponding to such an operator is achieved by reduction of combinator terms using the rewrite rules of the system. According to the definition, any combinator is constructed from the basic combinators K and S. The examples in the next section show that even operators with simple intuition may require quite complicated combinators for their representation.

The rewrite relation

We now define the rewrite relation on combinatory terms.

Any combinatory term of the form KXY or SXYZ, where X, Y and Z are arbitrary terms, is called a *redex*. The *contractum* of a redex is defined as:

- X for a redex of the form KXY, and
- XZ(YZ) for a redex of the form SXYZ.

The rewrite relation \rightarrow on combinatory terms is defined as follows: $P \rightarrow Q$ iff Q can be obtained by replacing one occurrence of a redex in P by its contractum. For example, $SKKy \rightarrow Ky(Ky) \rightarrow y$.

This example shows that the combinator SKK behaves as identity operator on combinatory terms. Its application to an arbitrary term Y reduces to Y. Thus SKK is an important combinator, with a very simple intuition. It is often denoted by I. Still its definition is relatively complicated.

The intuition of the basic combinators K and S is not as simple as that of I. K forms constant functions: application of K to a term P is a function returning P for every argument, since $(KP)x \rightarrow P$. S is a function composition operator. Given the arguments X, Y, Z it applies the function XZto the function YZ. This kind of composition does not seem very natural. On the other hand, the combinator corresponding to the usual composition of functions is more complicated. It has the form S(KS)K and it is denoted B. Indeed, for every X, Y, Z the application BXYZ reduces to X(YZ):

 $BXYZ = S(KS)KXYZ \rightarrow (KS)X(KX)YZ \rightarrow S(KX)YZ \rightarrow (KX)Z(YZ) \rightarrow X(YZ)$

Thus the result is the application of X to the (result of) application of Y to Z. This corresponds to the usual intuition of composition of functions.

An important result about the rewrite system of combinatory logic is the Church-Rosser theorem, analogous to that for lambda calculus, stating confluence of this rewrite system.

Another important result analogous to that for lambda calculus states that if a combinatory term has a normal form, then the reduction under the normal strategy terminates. Due to the confluence of the system, such a reduction results in this normal form, which is unique.

Relating combinatory logic to lambda calculus

As mentioned above, both lambda calculus and combinatory logic are intended to represent and compute functions. The question is then how the formalisms are related to each other. A part of the answer can be obtained by studying the "compilation" of combinatory terms into lambda terms, called in the literature the transformation λ . As discussed below, this transformation preserves the rewrite relation.

The transformation λ associates to each combinatory term X a lambda term X_{λ} defined as follows, by induction on the structure of X:

- $y_{\lambda} = y$, for every variable y,
- $K_{\lambda} = \lambda xy.x, \ S_{\lambda} = \lambda xyz.xz(yz),$
- $(YZ)_{\lambda} = Y_{\lambda}Z_{\lambda}$ for arbitrary combinatory terms Y and Z.

Thus, for example,

$$(SKK)_{\lambda} = (SK)_{\lambda}K_{\lambda} = S_{\lambda}K_{\lambda}K_{\lambda} = \lambda xyz.xz(yz)(\lambda xy.x)(\lambda xy.x) \rightarrow \lambda yz.(\lambda xy.x)z(yz)\lambda xy.x \rightarrow \lambda yz.(\lambda xy.x)z(yz)\lambda xy.x \rightarrow \lambda yz.z\lambda xy.x \rightarrow \lambda z.z$$

As illustrated by the above example, the transformation preserves the "nature" of the function. Both SKK and $\lambda z.z$ represent the identity function. For arbitrary combinatory term X the application SKKX reduces to X in combinatory logic. On the other hand, for arbitrary lambda term t the application $(\lambda z.z)t \beta$ -reduces to t. Notice that the transformation preserves the free variables of the term transformed. Thus, the combinators are transformed to lambda terms without free variables. Such lambda terms are sometimes also called combinators (of lambda calculus).

Generally, the transformation λ preserves the rewrite relation. Formally, if Y can be obtained from a combinatory term X by a number of reduction

steps in the combinatory logic then the lambda term Y_{λ} can be obtained by a number of β -reduction steps from X_{λ} .

A related question, how to compile lambda terms into combinatory terms, has also been discussed in the literature. One of the proposed solutions, a transformation denoted H will be described now.

The first step concerns simulation of functional abstraction in combinatory terms. For every variable x and for every combinatory term M we construct a combinatory term denoted $\lambda^c x.M$. The intention is that combinatory rewriting of $(\lambda^c x.MN)$ would replace the occurrences of x in M by N. In this way β -reduction would be simulated by rewriting of combinatory terms.

One can check that the goal can be achieved with the following definition:

- $\lambda^c x.x = SKK$
- $\lambda^c x \cdot P = KP$ if x does not appear in P,
- $\lambda^c x. PQ = S(\lambda^c x. P)(\lambda^c x. Q)$ if the previous case do not apply.

For example,

$$\lambda^{c}x.Kx = S(\lambda^{c}x.K)(\lambda^{c}x.x) = S(KK)(SKK).$$

Now, for any term P,

$$S(KK)(SKK)P \to (KKP)(SKKP) \to KP$$

The same term can be obtained by replacement of x by P in Kx.

The transformation H is now defined as follows:

- $x_H = x$ for every variable x,
- $(PQ)_H = P_H Q_H$ for any lambda terms P and Q,
- $(\lambda x.P)_H = \lambda^c x.(P)_H$

For example:

$$(\lambda xy.y)_H = \lambda^c x.(\lambda y.y)_H = \lambda^c x.(\lambda^c y.y) = \lambda^c x.SKK = K(SKK)$$

The transformation λ discussed above is a kind of inverse of H, since for every lambda term M the term $M_{H\lambda}$ is convertible to M (in lambda calculus).

For example:

$$(\lambda x.x)_{H\lambda} = (SKK)_{\lambda} = \lambda xyz.xz(yz)(\lambda xy.x)(\lambda xy.x) \twoheadrightarrow_{\beta} \lambda z.z =_{\alpha} \lambda x.x$$

One can compile a lambda term M to the combinatory term M_H , reduce it, and transform the result R of the reduction to the lambda term R_{λ} . As mentioned above, the transformation λ preserves the rewrite relation. Hence $M_{H,\lambda} \twoheadrightarrow_{\beta} R_{\lambda}$. This may motivate the use of combinatory logic as the implementation language for lambda calculus.

It should be stressed that the results outlined above do not give a full picture of the relation between lambda calculus and combinatory logic. A more comprehensive discussion of the issue can be found in [6].

Bibliography

- H.P. Barendregt. The Lambda calculus: its syntax and semantics, 2nd edition. North-Holland, 1984.
- [2] N. Chomsky. Three models for the description of language. In IRE transactions on information theory, 2:3, pp. 113–124, 1956.
- [3] A. Church. *The calculi of Lambda-conversion*. Annals of mathematical studies 6, Princeton University Press, 1941.
- [4] A. Church and J.B. Rosser. Some properties of conversion. *Transactions* of the American Mathematical Society 39, pp. 472–482, 1936.
- [5] H. Curry. Grundlagen der kombinatorischen Logik. American J. Math. 52, pp.509-536,789-834, 1930.
- [6] R. Hindley and J. Seldin. Introduction to combinators and λ -calculus. Cambridge University Press, 1986.
- [7] J. Hopcroft and J. Ullman. Introduction to automata theory, languages and computation, 2nd edition. Addison-Wesley, 1979.
- [8] Peter Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. of the ACM*, 8 89–101, 1965.
- [9] M. Newman. On theories with a combinatorial definition of "equivalence". Ann. Math. 43, pp. 223-243, 1942.
- [10] M. Schönfinkel. Über die Bausteine der matematischen Logik. Math. Ann. 92, pp. 305-316, 1924.
- [11] A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2:42, pp. 230–265. Corrected in *ibid.* 43, pp. 544–546, 1936.