MARTIN BUNDER

COMBINATORS, PROOFS AND IMPLICATIONAL LOGICS

1 INTRODUCTION

In this chapter we first look at operators called **combinators**. These are very simple but extremely powerful. They provide a means of doing logic and mathematics without using variables, are powerful enough to allow the definition of all recursive functions and have more recently been used as a basis for certain "functional" computer languages.

We are interested in another use here which involves the **functional character** or **type** possessed by many combinators. Each type can be interpreted as a theorem of the intuitionistic implicational logic $H_{\rightarrow}$ and combinators possessing that type can be interpreted as Hilbert-style proofs of that theorem. Weaker sets of combinators can be used to represent proofs in sublogics of $H_{\rightarrow}$, these include the substructural logics, such as the relevance logics $R_{\rightarrow}$ and $T_{\rightarrow}$. There is a further interpretation of combinators and types as programs and specifications which we will not discuss here.

Next we look at lambda calculus. This also allows the definition of all recursive functions and has also been used in foundations of mathematics and computer language development. Many lambda terms also have types and these again are the theorems of $H_{\rightarrow}$. The lambda terms represent natural deduction style proofs of these theorems.

In the third section of this chapter we look at translations from combinators to lambda terms and vice versa. For the combinators and lambda terms that represent proofs in $H_{\rightarrow}$ these translations are well known, for those corresponding to proofs in weaker logics they are quite new.

In a fourth section we develop a new algorithm which, given an implicational formula, allows us to find lambda terms representing natural deduction style proofs of the formula or demonstrates that the formula has no proof.

Most implicational substructural logics are specified by substructural rules or by axioms and not by rules in the natural deduction form. Our translation procedure, together with the algorithm, provides us with a simple constructive means of finding Hilbert-style proofs in many of these logics. As the translation procedure tells us which lambda terms are translatable into which sets of combinators, the algorithm can be directed to look only for the lambda terms of the appropriate kind. The algorithm is inherently finite; for any given formula, and for many logics, bounds for the proof searches can be written down.

D. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, Volume 6, 229–286.

The $H_{\rightarrow}$ algorithm has been implemented by Anthony Dekker as Brouwer 7.9.0 (see [Dekker, 1996]) and, for any implicational formula, produces a λ-term proof (or even 50 alternative proofs) or a guarantee that there is no proof, virtually instantly. The implementation has more recently been extended by Martijn Oostdijk as LambdaCal2, (see [Oostdijk, 1996]) to cover the other implicational systems in this chapter as well as certain systems with other connectives. This implementation supplies combinator and lambda calculus proofs.

2 COMBINATORY LOGIC

Combinators are operators which manipulate arbitrary expressions by cancellation, duplication, bracketing and permutation. Combinators were first defined by Schönfinkel in his 1924 paper and rediscovered by Curry in [1930].

To illustrate their use we consider the following examples:

let Axy (rather than $A(x, y)$) represent $x + y$. The commutative law for addition can then be written as

$$Axy = Ayx.$$

Given a combinator $\mathbf{C}$ with the property:

$$\mathbf{C}xyz = xzy$$

this becomes

$$Axy = \mathbf{C}Axy$$

which could be simply written, without variables, as

$$A = \mathbf{C}A.$$

Given an identity combinator $\mathbf{I}$, i.e. one such that

$$\mathbf{I}x = x$$

we can write

$$0 + x = x$$

as

$$A0x = x$$

or

$$A0x = \mathbf{I}x$$

and so, without variables, as

$$A0 = \mathbf{I}.$$

$$x + 0 = x$$

would be

$$\mathbf{C}A0 = \mathbf{I}.$$

Schönfinkel found that only two combinators, **K** and **S**, were enough to define all others.

We will now introduce these, other combinators, and our method of writing functional expressions (such as Axy rather than $A(x, y)$) more formally.

2.1 Combinators and Application

DEFINITION 1 (Combinator).

1. **K** and **S** are combinators.
2. If X and Y are combinators so is (XY).

(The operation in (2) is called **application**.)

Though it is possible, it is often not convenient to work without variables; we therefore introduce terms which are made up of combinators and variables using application. Other constants could also be included in (1) below.

DEFINITION 2 (Term).

1. **K**, **S**, x, y, $z, \ldots, x_1, x_2, \ldots$ are terms
2. If X and Y are terms so is (XY).

Notation We use association to the left for terms. This means that our Axy is short for $((Ax)y)$. A binary function over the real numbers such as A is therefore interpreted as a unary function from real numbers into the set of unary functions from real numbers to real numbers.

The process of going from $\mathbf{C}XYZ$ to XZY or from $\mathbf{I}X$ to X is called **reduction**. This is defined as follows:

DEFINITION 3 (Reduction). The relation $X \triangleright Y$ (X reduces weakly to Y) is defined as follows:

(**K**) $\mathbf{K}XY \triangleright X$

(**S**) $\mathbf{S}XYZ \triangleright XZ(YZ)$

(ρ) $X \triangleright X$

(μ) $X \triangleright Y \Rightarrow UX \triangleright UY$

(ν) $X \triangleright Y \Rightarrow XU \triangleright YU$

(τ) $X \triangleright Y$ and $Y \triangleright Z \Rightarrow X \triangleright Z$

(**K**) and (**S**) are called the **reduction axioms** for **K** and **S**.

DEFINITION 4 (Weak equality). $X = Y$ if this can be derived from the axioms and rules of Definition 3 with "=" instead of " $\triangleright$", together with

(σ) $X = Y \Rightarrow Y = X$.

The formal system consisting of at least Definition 1 and the postulates in Definitions 3 we call **combinatory logic**. Other axioms and rules may be added.

We now show how the combinators we met earlier, and others, can be defined. We use "$\equiv$" for "equals by definition".

DEFINITION 5.

$$\begin{aligned} \mathbf{I} &\equiv \mathbf{SKK} \\ \mathbf{B} &\equiv \mathbf{S(KS)K} \\ \mathbf{C} &\equiv \mathbf{S(BBS)(KK)} \\ \mathbf{B'} &\equiv \mathbf{CB} \\ \mathbf{W} &\equiv \mathbf{SS(KI)} \\ \mathbf{S'} &\equiv \mathbf{B(BW)(BBB')} \end{aligned}$$

Each of these defined combinators has a characteristic reduction theorem:

THEOREM 6.

1. $\mathbf{I}X \triangleright X$
2. $\mathbf{B}XYZ \triangleright X(YZ)$
3. $\mathbf{C}XYZ \triangleright XZY$
4. $\mathbf{B'}XYZ \triangleright Y(XZ)$
5. $\mathbf{W}XY \triangleright XYY$
6. $\mathbf{S'}XYZ \triangleright YZ(XZ)$

Proof.

1.	$\mathbf{SKK}X \triangleright \mathbf{K}X(\mathbf{K}X)$		by (**S**)
so	$\mathbf{SKK}X \triangleright X$		by (**K**) and (τ)
2.	$\mathbf{S(KS)K}XYZ \triangleright \mathbf{KS}X(\mathbf{K}X)YZ$		by (τ) and (ν)
	$\triangleright \mathbf{S}(\mathbf{K}X)YZ$		by (**K**) and (ν)
	$\triangleright \mathbf{K}XZ(YZ)$		by (**S**)
	$\triangleright X(YZ)$		by (**K**) and (ν).
so	$\mathbf{S(KS)K}XYZ \triangleright X(YZ)$		by (τ) ■

If $M(X_1, \ldots, X_n)$ is a term made up by application using zero or more occurrences of each of $X_1, \ldots, X_n$, we can find a combinator Z such that $ZX_1X_2 \ldots X_n \triangleright M(X_1, \ldots, X_n)$.

This property is called the "**combinatory completeness**" of the combinatory logic based on **K** and **S**.

The combinator Z above is represented by

$$Z \equiv [x_1]([x_2](\ldots([x_n]M(x_1,\ldots,x_n))\ldots))$$

where each $[x_i](\ldots)$ is called a **bracket abstraction**.

Bracket abstractions can be defined in various ways, a simple definition, involving **S** and **K**, is as follows:

DEFINITION 7 (Bracket abstraction $[x_i]$).

(i) $[x_i]\, x_i \equiv \mathbf{I}$

(k) $[x_i]\, Y \equiv \mathbf{K}Y$ if $x_i \notin Y$

(η) $[x_i]Yx_i \equiv Y$ if $x_i \notin Y$

(s) $[x_i]YZ \equiv \mathbf{S}([x_i]Y)([x_i]Z)$,

where $x_i \notin Y$ stands for x_i does not appear in Y.

The above clauses must be used in the order given i.e. $(ik\eta s)$. In the order $(iks\eta)$, we would always obtain $\mathbf{S}([x_i]Y)\mathbf{I}$ for $[x_i]Yx_i$, if $x_i \notin Y$, instead of the simpler Y.

Repeated bracket abstraction as in $[x_1]([x_2](\ldots([x_n]M)\ldots))$ we will write as $[x_1, x_2, \ldots, x_n]M$.

EXAMPLE 8.

$$\begin{array}{lcll} [x_1,x_2,x_3]\ x_3(x_1x_3) & \equiv & [x_1,x_2]\mathbf{S}([x_3]x_3)([x_3](x_1x_3)) & \text{by } (s) \\ & \equiv & [x_1,x_2]\mathbf{S}\mathbf{I}x_1 & \text{by } (i) \text{ and } (\eta) \\ & \equiv & [x_1]\mathbf{K}(\mathbf{S}\mathbf{I}x_1) & \text{by } (k) \\ & \equiv & \mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}\mathbf{I}) & \text{by } (k) \text{ and } (\eta). \end{array}$$

$$\begin{array}{rcl} \mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}\mathbf{I})x_1x_2x_3 & \triangleright & \mathbf{K}\mathbf{K}x_1(\mathbf{S}\mathbf{I}x_1)x_2x_3 \\ & \triangleright & \mathbf{K}(\mathbf{S}\mathbf{I}x_1)x_2x_3 \\ & \triangleright & \mathbf{S}\mathbf{I}x_1x_3 \\ & \triangleright & \mathbf{I}x_3(x_1x_3) \\ & \triangleright & x_3(x_1x_3). \end{array}$$

Bracket abstraction has the following property which we call (β) for lambda abstraction in Section 3.

THEOREM 9. *$([x]X)Y \triangleright [Y/x]X$ where $[Y/x]X$ is the result of substituting Y for all occurrences of x in X.*

Proof. By a simple induction on the length of X. ■

http://www.springer.com/978-1-4020-0583-1

Handbook of Philosophical Logic
Gabbay, D.; Guenthner, F. (Eds.)
2002, XIII, 406 p. 1 illus., Hardcover
ISBN: 978-1-4020-0583-1