Combinatory Logic: From Philosophy and Mathematics to Computer Science

Alexander Farrugia
alex.farrugia@um.edu.mt

Abstract

In 1920, Moses Schönfinkel provided the first rough details of what later became known as combinatory logic. This endeavour was part of Hilbert's program to formulate mathematics as a consistent logic system based on a finite set of axioms and inference rules. This program's importance to the foundations and philosophical aspects of mathematics is still celebrated today. In the 1930s, Haskell Curry furthered Schönfinkel's work on combinatory logic, attempting – and failing – to show that it can be used as a foundation for mathematics. However, in 1947, he described a high-level functional programming language based on combinatory logic. Research on functional programming languages continued, reaching a high point in the eighties. However, by this time, object-oriented programming languages began taking over and functional languages started to lose their appeal. Lately, however, a resurgence of functional languages is being noted. Indeed, many of the commonly-used programming languages nowadays incorporate functional programming elements in them, while functional languages such as Haskell, OCaml and Erlang are gaining in popularity. Thanks to this revival, it is appropriate to breathe new life into combinatory logic by presenting its main ideas and techniques in this paper.

Keywords: combinatory logic, combinator, functional programming, logic and philosophy, foundations of mathematics.

Introduction: Schönfinkel's Idea

In first-order logic, *well-formed formulas* such as the following are frequently encountered:

$$\forall x\big(Nx \rightarrow \exists y(Ny \wedge Gyx)\big).$$

Nx here means 'x is a number', while Gyx means 'y is greater than x'. The above well-formed formula thus reads 'For all x, if x is a number, then there exists y such that y is a number and y is greater than x.'

Schönfinkel wanted to write down well-formed formulas in first-order logic in a way that did not require variables (Bimbó, 2016). To do this, he invented a new symbol U and defined it as follows:

$$UAB \stackrel{\text{def}}{=} \neg\exists x(Ax \wedge Bx).$$

The above reads 'There does not exist x such that both Ax and Bx are true'. For example, the logical sentence 'No rabbit eats meat' can be written as URM, where Rx means 'x is a rabbit' and Mx means 'x eats meat'. In this way, the variable x is eliminated.

To do the same thing for the expression $\forall x\big(Nx \rightarrow \exists y(Ny \wedge Gyx)\big)$, the implication symbol $\rightarrow$ is first converted into a conjunction symbol $\wedge$. This is done by noticing that $A \rightarrow B$ means the same as $\neg A \vee B$ (Mendelson, 1997, p.12). In other words, the statement 'If A, then B' is equivalent to the statement 'either A is false or B is true'. It is assumed that the semantics of the word 'or' allow *at least* one of A or B to be true. For instance, the sentences 'If it is raining, then it is cloudy' and 'it is either not raining or it is cloudy (or both)' mean the same thing. Furthermore, the expression $\neg(A \wedge B)$ means the same as $\neg A \vee \neg B$ – in other words, 'it is not the case that both A and B are true' and 'either A is false or B is false (or both)' are equivalent (Mendelson, 1997, pp.11-12). In this fashion, the bracket $Nx \rightarrow \exists y(Ny \wedge Gyx)$ first gets transformed into $\neg Nx \vee \exists y(Ny \wedge Gyx)$ and then into $\neg\big(Nx \wedge \neg\exists y(Ny \wedge Gyx)\big)$. Lastly, the well-formed formula $\forall x(\neg Ax)$ is equivalent to $\neg\exists x(Ax)$ (Mendelson, 1997, p.52). In layman's terms, the statements 'all x do not satisfy property A' and 'there exists no x that satisfies property A' mean the same thing. For example, the sentences 'all rabbits do not eat meat' and 'there is no rabbit that eats meat' are synonymous. Thus, the well-formed formula $\forall x\big(Nx \rightarrow \exists y(Ny \wedge Gyx)\big)$ is converted to $\forall x\Big(\neg\big(Nx \wedge \neg\exists y(Ny \wedge Gyx)\big)\Big)$ as explained beforehand, and then to the well-formed formula

$$\neg\exists x\big(Nx \wedge \neg\exists y(Ny \wedge Gyx)\big).$$

The reason why these transformations were performed is that the above expression is much more amenable to using the U symbol mentioned above. For example, the subexpression

$$\neg\exists y(Ny \wedge Gyx)$$

is almost of the form $\neg\exists x(Ax \wedge Bx)$ needed. Unfortunately, the variable y in the expression Gyx needs to be in the end for these two expressions to match. Thus, Schönfinkel introduced the following *combinators* (1924), having the following rules:

$$\mathbf{S}xyz \stackrel{\text{def}}{=} xz(yz);$$
$$\mathbf{K}xy \stackrel{\text{def}}{=} x;$$
$$\mathbf{I}x \stackrel{\text{def}}{=} x;$$
$$\mathbf{B}xyz \stackrel{\text{def}}{=} x(yz);$$
$$\mathbf{C}xyz \stackrel{\text{def}}{=} xzy.$$

In our case, Gyx is written as $\mathbf{C}Gxy$, so that $\neg\exists y(Ny \wedge Gyx)$ becomes $\neg\exists y(Ny \wedge \mathbf{C}Gxy)$. The U symbol may now be applied to this expression, replacing it with $UN(\mathbf{C}Gx)$. The well-formed formula $\neg\exists x\big(Nx \wedge \neg\exists y(Ny \wedge Gyx)\big)$ hence becomes $\neg\exists x\big(Nx \wedge UN(\mathbf{C}Gx)\big)$.

Moreover, the $\mathbf{B}$ combinator is used to convert $UN(\mathbf{C}Gx)$ into $\mathbf{B}(UN)(\mathbf{C}G)x$. This allows the use of the U symbol again; $\neg\exists x\big(Nx \wedge UN(\mathbf{C}Gx)\big)$ is converted to $\neg\exists x(Nx \wedge \mathbf{B}(UN)(\mathbf{C}G)x)$, and then, finally, to $UN\big(\mathbf{B}(UN)(\mathbf{C}G)\big)$.

In this manner, Schönfinkel showed how to convert first-order well-formed formulas into expressions involving no variables at all by using the combinators $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$. This new way of writing well-formed formulas became known as *combinatory logic* (Bimbó, 2016). Combinatory logic completely solves the problem of deciding how variables are bounded to the quantifiers ∀ and ∃, by dispensing of variables altogether. The disadvantage is that the final expression $UN\big(\mathbf{B}(UN)(\mathbf{C}G)\big)$ is less readable than the original expression $\forall x\big(Nx \rightarrow \exists y(Ny \wedge Gyx)\big)$. However, this is of no concern to a computer, and, indeed, combinatory logic is the basis upon which computer functional programming languages work (Peyton Jones, 1987, pp.260-280).

Computer Science

In computer science, Schönfinkel's $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ combinators are treated as *functions*, each of which behaves in the way defined above. This idea is the basis of what is nowadays called *functional programming*. For example, the combinator $\mathbf{I}$, having the rule $\mathbf{I}x \stackrel{\text{def}}{=} x$, is treated as a function that takes a variable x as input and returns x itself as output. Similarly, the $\mathbf{K}$ combinator can be interpreted as being a function that takes two inputs x and y and returns x as output, completely ignoring the second input y (Bird & Wadler, 1988, pp.8-9).

This interpretation of combinators as functions allows the input variables, like x and y above, to be functions themselves. Indeed, they may even be other combinators. For example, the $\mathbf{B}$ combinator, having the rule $\mathbf{B}xyz \stackrel{\text{def}}{=} x(yz)$, is interpreted as being a function taking three inputs x, y and z and which outputs the result obtained after applying the *function* x to the result of applying the *function* y to z. This means, essentially, that the expression yz means, from a functional programming perspective, 'the function y applied to z' (Bird & Wadler, 1988, p.9). In mathematics, such an expression is usually written as $y(z)$.

Another useful idea in functional programming is the fact that the output of a function may be a function itself. This is illustrated by the $\mathbf{C}$ combinator $\mathbf{C}xyz \stackrel{\text{def}}{=} xzy$, which takes three inputs x, y and z. The $\mathbf{C}$ combinator first applies the function x to z, resulting in the *function* xz. This function xz is then applied to the variable y, and the output of this final function application is the output of the $\mathbf{C}$

combinator. In essence, the expression xzy really means $(xz)y$. Likewise, the expression $xz(yz)$ taken from the definition of the $\mathbf{S}$ combinator means $(xz)(yz)$, and the expression $\mathbf{C}xyz$ means $(\mathbf{C}x)yz$ or even $((\mathbf{C}x)y)z$. This idea is called *currying* (Bird & Wadler, 1988, p.12; Peyton Jones, 1987, p.10), after Haskell Curry, who rediscovered combinatory logic independently of Schönfinkel and dedicated a lot of research on it.

As an aside, Schönfinkel only published two papers during his lifetime, one in 1924 (1924) and another in 1929 (Bernays & Schönfinkel, 1929). Only the 1924 paper was on combinatory logic and this idea of currying. This lack of proficiency by Schönfinkel is attributed to a mental illness, starting from 1927 till his death in 1942 (Kline & Anovskaa, 1951). Indeed, his second paper in 1929 was only published due to the efforts of his coauthor Paul Bernays. Curry always credited the concept of currying to Schönfinkel, although it must be said that the idea of currying had already been mentioned by the philosopher Gottlob Frege even before Schönfinkel (Quine, 1967). Nowadays, the programming language *Haskell* is named after Haskell Curry (Hudak *et al*, 2007).

The sufficiency of $\mathbf{S}$ and $\mathbf{K}$

Incredibly, Schönfinkel (1924) also showed that the combinators $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ are superfluous, because they can be defined in terms of the other two combinators, $\mathbf{S}$ and $\mathbf{K}$. Indeed:

$$\mathbf{I} = \mathbf{SKK};$$
$$\mathbf{B} = \mathbf{S}(\mathbf{KS})\mathbf{K};$$
$$\mathbf{C} = \mathbf{S}(\mathbf{BBS})(\mathbf{KK}).$$

For example, $\mathbf{SKK}x$ reduces to $\mathbf{K}x(\mathbf{K}x)$ by using the $\mathbf{S}$ combinator rule, and then $\mathbf{K}x(\mathbf{K}x)$ reduces to x by using the $\mathbf{K}$ combinator rule. This may be written down as

$$\mathbf{SKK}x \longrightarrow \mathbf{K}x(\mathbf{K}x) \longrightarrow x$$

or simply as

$$\mathbf{SKK}x \longrightarrow x.$$

But since $\mathbf{I}x \longrightarrow x$ using the $\mathbf{I}$ combinator rule, $\mathbf{I}$ and $\mathbf{SKK}$ affect the variable x in the same way. Because of this, the $\mathbf{I}$ combinator is said to be *extensionally equal* to the *combinator expression* $\mathbf{SKK}$, written $\mathbf{I} = \mathbf{SKK}$ (Peyton Jones, 1987, p.266; Barendregt, 1984, pp.151-163; Hindley & Seldin, 2008, p.26). In a similar way, the $\mathbf{B}$ combinator may be proved to be extensionally equal to the combinator expression $\mathbf{S}(\mathbf{KS})\mathbf{K}$:

$$\mathbf{S}(\mathbf{KS})\mathbf{K}xyz \longrightarrow \mathbf{KS}x(\mathbf{K}x)yz \longrightarrow \mathbf{S}(\mathbf{K}x)yz \longrightarrow \mathbf{K}xz(yz) \longrightarrow x(yz).$$

Here is the proof that the $\mathbf{C}$ combinator is extensionally equal to the combinator expression $\mathbf{S}(\mathbf{BBS})(\mathbf{KK})$:

$$\mathbf{S}(\mathbf{BBS})(\mathbf{KK})xyz \longrightarrow \mathbf{BBS}x(\mathbf{KK}x)yz \longrightarrow \mathbf{B}(\mathbf{S}x)(\mathbf{KK}x)yz$$
$$\longrightarrow \mathbf{S}x(\mathbf{KK}xy)z \longrightarrow xz(\mathbf{KK}xyz) \longrightarrow xz(\mathbf{K}yz) \longrightarrow xzy.$$

Turing Completeness

Even more surprisingly, any computable function can be expressed using any or all of Schönfinkel's five combinators $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$. This means that any computer program may be written using these five combinators only (Hindley & Seldin, 2008, p.47). The combinators $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ are said to be *Turing complete*. In fact, by applying the results of the previous section, since $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ may be themselves expressed in terms of $\mathbf{S}$ and $\mathbf{K}$, the two combinators $\mathbf{S}$ and $\mathbf{K}$ alone suffice to write *any* computer program – a result that is as astounding as it is beautiful.

The following recursive algorithm converts any function $\mathbf{F}$, having the n inputs $x_1, x_2, x_3, \dots, x_n$, into a combinator expression involving some or all the combinators $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ (Peyton Jones, 1987, p.270; Curry & Feys, 1958). This provides an informal proof for the claim in the previous paragraph.

Algorithm:

Step 1: If the function $\mathbf{F}$ has no arguments and is defined as the combinator expression f, then output $\mathbf{F} \stackrel{\text{def}}{=} f$ and halt.

Step 2: Otherwise, $\mathbf{F}x_1x_2x_3 \dots x_{n-1}x \stackrel{\text{def}}{=} f$ is equivalent to the function $\mathbf{F}x_1x_2x_3 \dots x_{n-1} \stackrel{\text{def}}{=} [f]_x$, where $[f]_x$ is defined as follows:

a. $[x]_x \stackrel{\text{def}}{=} \mathbf{I}$;
b. $[f]_x \stackrel{\text{def}}{=} \mathbf{K}f$, if the expression f does not contain the variable x;
c. $[f]_x \stackrel{\text{def}}{=} g$, if the expression f is the function application gx and g does not contain the variable x;
d. $[f]_x \stackrel{\text{def}}{=} \mathbf{B}f_1[f_2]_x$, if the expression f is the function application f_1f_2, where f_1 does not contain the variable x and f_2 contains the variable x;
e. $[f]_x \stackrel{\text{def}}{=} \mathbf{C}[f_1]_xf_2$, if the expression f is the function application f_1f_2, where f_1 contains the variable x and f_2 does not contain the variable x;
f. $[f]_x \stackrel{\text{def}}{=} \mathbf{S}[f_1]_x[f_2]_x$, if the expression f is the function application f_1f_2, where both f_1 and f_2 contain the variable x.

As an illustration, the function $\mathbf{W}$, defined as $\mathbf{W}xy \stackrel{\text{def}}{=} xyy$, will be converted into a combinator expression using the above algorithm. Since this function has two arguments, Step 2 of the algorithm is invoked, writing down $\mathbf{W}xy \stackrel{\text{def}}{=} xyy$ as $\mathbf{W}x \stackrel{\text{def}}{=} [(xy)y]_y$. By Step 2f., $[(xy)y]_y \stackrel{\text{def}}{=} \mathbf{S}[xy]_y[y]_y$. By Step 2c. and Step 2a. respectively, $[xy]_y \stackrel{\text{def}}{=} x$ and $[y]_y \stackrel{\text{def}}{=} \mathbf{I}$. Hence

$$\mathbf{W}xy \stackrel{\text{def}}{=} xyy \text{ is the same as } \mathbf{W}x \stackrel{\text{def}}{=} \mathbf{S}x\mathbf{I}.$$

The variable y has been eliminated. To eliminate variable x, Step 2 of the algorithm is again used, so that $\mathbf{W}x \stackrel{\text{def}}{=} \mathbf{S}x\mathbf{I}$ is written down as $\mathbf{W} \stackrel{\text{def}}{=} [(\mathbf{S}x)\mathbf{I}]_x$. By Step 2e., $[(\mathbf{S}x)\mathbf{I}]_x \stackrel{\text{def}}{=} \mathbf{C}[\mathbf{S}x]_x\mathbf{I}$. Moreover, by Step 2c., $[\mathbf{S}x]_x \stackrel{\text{def}}{=} \mathbf{S}$. Thus

$$\mathbf{W}x \stackrel{\text{def}}{=} \mathbf{S}x\mathbf{I} \text{ is the same as } \mathbf{W} \stackrel{\text{def}}{=} \mathbf{CSI}.$$

Step 1 of the algorithm is now called to finalize the function $\mathbf{W}$ as the combinator expression $\mathbf{CSI}$. The expression $\mathbf{CSI}xy$ is now checked to confirm that

it indeed reduces to the expression xyy, proving that both $\mathbf{W}$ and $\mathbf{CSI}$ produce the same output:

$$\mathbf{CSI}xy \longrightarrow \mathbf{S}x\mathbf{I}y \longrightarrow xy(\mathbf{I}y) \longrightarrow xyy.$$

The above algorithm may be thought of as a *compiler* that translates any function into a combinator expression. Indeed, compilers that translated code into combinators were used for the functional programming languages SASL and Miranda (Peyton Jones, 1987, p.2). These two programming languages later inspired the creation of the much more successful Haskell functional programming language.

Perhaps so far, the reader is not very convinced that combinatory logic may be used to write any computer program. The following sections should alleviate this concern.

Boolean Values

One very important feature of any programming language is the presence of *conditional statements*, in which different parts of a program are evaluated depending on whether a *Boolean value* is *true* or *false*. This idea may be encapsulated using so-called *Church Booleans,* named after Alonzo Church (1940) as follows:

$$txy \stackrel{\text{def}}{=} x;$$
$$fxy \stackrel{\text{def}}{=} y.$$

The symbols t and f stand for 'true' and 'false' respectively. If a Boolean value is t, then x is evaluated, while if it f, then y is evaluated. By using the algorithm in the previous section to compile these Boolean values into combinators, the following are obtained (Smullyan, 2000, p.212):

$$t \stackrel{\text{def}}{=} \mathbf{K};$$
$$f \stackrel{\text{def}}{=} \mathbf{KI}.$$

The definition of t as being the $\mathbf{K}$ combinator shouldn't be surprising, as clearly t and $\mathbf{K}$ have the same behaviour.

Using these definitions of t and f, logical connectives such as ∧ (AND), ∨ (OR) and ¬ (NOT) may be implemented (note that the prefix notation $\wedge xy$ and $\vee xy$ is being used here, instead of the more usual infix notation $x \wedge y$ and $x \vee y$) (Church, 1940):

$$\neg x \stackrel{\text{def}}{=} xft;$$
$$\wedge xy \stackrel{\text{def}}{=} xyf;$$
$$\vee xy \stackrel{\text{def}}{=} xty.$$

The first definition, $\neg x \stackrel{\text{def}}{=} xft$, states that if x is t, then $\neg x$ evaluates to f, while if x is f, then $\neg x$ evaluates to t. This is, of course, what is expected from the behaviour of the NOT logical connective. The definition of the AND logical connective, $\wedge xy \stackrel{\text{def}}{=} xyf$, states that if x is t, then $\wedge xy$ evaluates to y, while if x is

f, then $\wedge xy$ evaluates to f. This may be verified by the reader to conform with the usual interpretation of the AND logical connective that returns t if and only if both inputs x and y are t. Finally, $\vee xy \stackrel{\text{def}}{=} xty$ states that if x is t, then $\vee xy$ returns t, while if x is f, then $\vee xy$ returns y. Again, this corresponds with how the OR logical connective is usually defined, returning f if and only if both inputs x and y are f.

Compiling these definitions using the algorithm in the previous section results in the following combinator expressions for $\neg$, $\wedge$ and $\vee$:

$$\neg \stackrel{\text{def}}{=} \mathbf{C}(\mathbf{CI}\mathit{f})t;$$
$$\wedge \stackrel{\text{def}}{=} \mathbf{CC}\mathit{f};$$
$$\vee \stackrel{\text{def}}{=} \mathbf{CI}t.$$

Below, the expression $\vee(\wedge t\mathit{f})(\neg\mathit{f})$ is evaluated, to illustrate that these definitions do indeed work as intended:

$$\begin{aligned}
&\vee(\wedge t\mathit{f})(\neg\mathit{f})\\
&\stackrel{\text{def}}{=} \mathbf{CI}t(\wedge t\mathit{f})(\neg\mathit{f}) \longrightarrow \mathbf{I}(\wedge t\mathit{f})t(\neg\mathit{f}) \longrightarrow \wedge t\mathit{f}t(\neg\mathit{f})\\
&\stackrel{\text{def}}{=} \mathbf{CC}\mathit{f}t\mathit{f}t(\neg\mathit{f}) \longrightarrow \mathbf{C}t\mathit{f}\mathit{f}t(\neg\mathit{f}) \longrightarrow t\mathit{f}\mathit{f}t(\neg\mathit{f})\\
&\stackrel{\text{def}}{=} \mathbf{K}\mathit{f}\mathit{f}t(\neg\mathit{f}) \longrightarrow \mathit{f}t(\neg\mathit{f})\\
&\stackrel{\text{def}}{=} \mathbf{KI}t(\neg\mathit{f}) \longrightarrow \mathbf{I}(\neg\mathit{f}) \longrightarrow \neg\mathit{f}\\
&\stackrel{\text{def}}{=} \mathbf{C}(\mathbf{CI}\mathit{f})t\mathit{f} \longrightarrow \mathbf{CI}\mathit{f}\mathit{f}t \longrightarrow \mathbf{I}\mathit{f}\mathit{f}t \longrightarrow \mathit{f}\mathit{f}t\\
&\stackrel{\text{def}}{=} \mathbf{KI}\mathit{f}t \longrightarrow \mathbf{I}t \longrightarrow t.
\end{aligned}$$

Since $\wedge t\mathit{f}$ evaluates to f and $\neg\mathit{f}$ evaluates to t, $\vee(\wedge t\mathit{f})(\neg\mathit{f})$ is equivalent to $\vee\mathit{f}t$, which evaluates to t. This shows that the above output is, indeed, the correct one.

Pairs

Combinatory logic can also model *pairs* of objects. One way to do this is to denote a pair (x, y) as $\mathcal{P}xy$, and then define the left and right functions $\mathcal{L}$ and $\mathcal{R}$ as follows:

$$\mathcal{L}(\mathcal{P}xy) \stackrel{\text{def}}{=} x;$$
$$\mathcal{R}(\mathcal{P}xy) \stackrel{\text{def}}{=} y.$$

To accomplish this behaviour, the symbols $\mathcal{P}$, $\mathcal{L}$ and $\mathcal{R}$ may be defined as underneath (Pierce, 2002):

$$\mathcal{P}xyz \stackrel{\text{def}}{=} zxy;$$
$$\mathcal{L}x \stackrel{\text{def}}{=} xt;$$
$$\mathcal{R}x \stackrel{\text{def}}{=} x\mathit{f}.$$

In this way, $\mathcal{L}(\mathcal{P}xy)$ would reduce to $\mathcal{P}xyt$ and then to txy, which outputs the correct value x. Similarly, $\mathcal{R}(\mathcal{P}xy)$ would reduce to $\mathit{f}xy$, which outputs y.

Compiling these definitions of $\mathcal{P}$, $\mathcal{L}$ and $\mathcal{R}$ into combinator expressions using the algorithm described earlier yields:

$$\mathcal{P} \stackrel{\text{def}}{=} \mathbf{BC}(\mathbf{CI});$$
$$\mathcal{L} \stackrel{\text{def}}{=} \mathbf{CI}t;$$
$$\mathcal{R} \stackrel{\text{def}}{=} \mathbf{CI}\mathit{f}.$$

A very important construct in functional programming is the concept of a *list*, which may simply be interpreted as an appropriately nested pair (Pierce, 2002, p.500).

Numbers

Another surprising fact about combinatory logic is that it can model the *natural numbers* $0,1,2,3,\ldots$. There are several ways of accomplishing this, for example using *Church numerals* (Church, 1933). Here, Barendregt's construction of the natural numbers using combinatory logic (Barendregt, 1976; Smullyan, 2000, pp.215-216) is described:

$$0 \stackrel{\text{def}}{=} \mathbf{I};$$
$$\mathfrak{S} \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}.$$

The symbol $\mathfrak{S}$ stands for the *successor* of a number. For example, the number 1 is defined as $\mathfrak{S}0$, the successor of the number 0. Likewise, 2 is defined as $\mathfrak{S}1$, or as $\mathfrak{S}(\mathfrak{S}0)$, and so on. As defined above, then, the numbers $0,1,2,3,\ldots$ are defined as follows:

$$\begin{aligned}
&0 \stackrel{\text{def}}{=} \mathbf{I};\\
&1 \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}\mathbf{I};\\
&2 \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}(\mathcal{P}\mathfrak{f}\mathbf{I});\\
&3 \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}\big(\mathcal{P}\mathfrak{f}(\mathcal{P}\mathfrak{f}\mathbf{I})\big);\\
&\vdots
\end{aligned}$$

In this way, Barendregt numerals define a number n as n nested pairs (1976).

A *predicate* is a function that returns a Boolean value. The way Barendregt numerals are defined makes it easy to create a function $\mathcal{Z}$ that tests whether its input is zero or not:

$$\mathcal{Z} \stackrel{\text{def}}{=} \mathcal{L}.$$

Since nonzero numerals are pairs whose left element is always $\mathfrak{f}$, the symbol for 'false', the definition of $\mathcal{Z}$ is simply $\mathcal{L}$, the function that extracts this left element. When $\mathcal{Z}$ is applied to 0, the following happens:

$$\mathcal{Z}0 \stackrel{\text{def}}{=} \mathcal{L}0 \stackrel{\text{def}}{=} \mathbf{CI}t0 \longrightarrow \mathbf{I}0t \longrightarrow 0t \stackrel{\text{def}}{=} \mathbf{I}t \longrightarrow t.$$

This new predicate $\mathcal{Z}$ may now be used to write a function $\mathfrak{P}$ that returns the *predecessor* of a number. The predecessor of the number 1 is 0, the predecessor of the number 2 is 1, and so on. Unfortunately, 0 does not have a predecessor among the natural numbers. Because of this, the predecessor of 0 is defined as 0.

$$\mathfrak{P}n \stackrel{\text{def}}{=} (\mathcal{Z}n)0(\mathcal{R}n).$$

The predecessor function $\mathfrak{P}$ first tests whether the input number n is 0, by invoking $\mathcal{Z}n$. If this function $\mathcal{Z}n$ returns t, then $\mathfrak{P}$ returns 0. If $\mathcal{Z}n$ returns $\mathfrak{f}$, then $\mathfrak{P}$ returns $\mathcal{R}n$, the right element of the pair that n is composed of. The function $\mathfrak{P}$ compiles to the following combinator expression using the algorithm presented earlier in this paper:

$$\mathfrak{P} \stackrel{\text{def}}{=} \mathbf{S}(\mathbf{C}\mathcal{Z}0)\mathcal{R}.$$

For example, $\mathfrak{P}2$ reduces to 1 in the following way:

$$\begin{aligned}
&\mathfrak{P}2\\
&\stackrel{\text{def}}{=} \mathbf{S}(\mathbf{C}\mathcal{Z}0)\mathcal{R}2 \longrightarrow \mathbf{C}\mathcal{Z}02(\mathcal{R}2) \longrightarrow \mathcal{Z}20(\mathcal{R}2)\\
&\stackrel{\text{def}}{=} \mathbf{CI}t20(\mathcal{R}2) \longrightarrow \mathbf{I}2t0(\mathcal{R}2) \longrightarrow 2t0(\mathcal{R}2)\\
&\stackrel{\text{def}}{=} \mathfrak{S}1t0(\mathcal{R}2) \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}1t0(\mathcal{R}2) \stackrel{\text{def}}{=} \mathbf{BC}(\mathbf{CI})\mathfrak{f}1t0(\mathcal{R}2)\\
&\longrightarrow \mathbf{C}(\mathbf{CI}\mathfrak{f})1t0(\mathcal{R}2) \longrightarrow \mathbf{CI}\mathfrak{f}t10(\mathcal{R}2) \longrightarrow \mathbf{I}t\mathfrak{f}10(\mathcal{R}2) \longrightarrow t\mathfrak{f}10(\mathcal{R}2)\\
&\stackrel{\text{def}}{=} \mathbf{K}\mathfrak{f}10(\mathcal{R}2) \longrightarrow \mathfrak{f}0(\mathcal{R}2)\\
&\stackrel{\text{def}}{=} \mathbf{KI}0(\mathcal{R}2) \longrightarrow \mathbf{I}(\mathcal{R}2) \longrightarrow \mathcal{R}2\\
&\stackrel{\text{def}}{=} \mathbf{CI}\mathfrak{f}2 \longrightarrow \mathbf{I}2\mathfrak{f} \longrightarrow 2\mathfrak{f}\\
&\stackrel{\text{def}}{=} \mathfrak{S}1\mathfrak{f} \stackrel{\text{def}}{=} \mathcal{P}\mathfrak{f}1\mathfrak{f} \stackrel{\text{def}}{=} \mathbf{BC}(\mathbf{CI})\mathfrak{f}1\mathfrak{f}\\
&\longrightarrow \mathbf{C}(\mathbf{CI}\mathfrak{f})1\mathfrak{f} \longrightarrow \mathbf{CI}\mathfrak{f}\mathfrak{f}1 \longrightarrow \mathbf{I}\mathfrak{f}\mathfrak{f}1 \longrightarrow \mathfrak{f}\mathfrak{f}1\\
&\stackrel{\text{def}}{=} \mathbf{KI}\mathfrak{f}1 \longrightarrow \mathbf{I}1 \longrightarrow 1.
\end{aligned}$$

The reduction process above shows how the predecessor of 2 is evaluated in a considerable number of very simple steps – a task that is very suitable to a computer.

Of course, it would be relatively useless to define numbers if no operations, like addition or multiplication, were defined on them. Before doing this, the $\mathbf{Y}$ combinator needs to be discussed first.

Recursion

A very useful programming construct is the concept of a *loop*. In functional programming, loops are expressed using *recursion*, where functions call themselves (Bird & Wadler, 1988, pp.104-108). Recursion can be expressed in combinatory logic using the $\mathbf{Y}$ combinator.

Consider the function $\mathbf{U}$ defined as follows:

$$\mathbf{U}xy \stackrel{\text{def}}{=} y(xxy).$$

Using the algorithm presented earlier, this function compiles into the following combinator expression:

$$\mathbf{U} \stackrel{\text{def}}{=} \mathbf{B}(\mathbf{SI})(\mathbf{SII}).$$

Note that, from the first definition $\mathbf{U}xy \stackrel{\text{def}}{=} y(xxy)$, it is clear that $\mathbf{UU}y$ reduces to $y(\mathbf{UU}y)$. Of course, $y(\mathbf{UU}y)$ reduces to $y\big(y(\mathbf{UU}y)\big)$, and then to $y\Big(y\big(y(\mathbf{UU}y)\big)\Big)$, and so on. Thus, the function $\mathbf{UU}$ is encapsulating the concept of recursion, in the sense that $\mathbf{UU}y$ keeps calling the function y forever. This is exactly how the notable computer scientist *Alan Turing* defined the $\mathbf{Y}$ combinator: as this function $\mathbf{UU}$ (1937). In terms of the five combinators $\mathbf{S}$, $\mathbf{K}$, $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$, the $\mathbf{Y}$ combinator is defined as

$$\mathbf{Y} \stackrel{\text{def}}{=} \mathbf{UU} \stackrel{\text{def}}{=} \mathbf{B}(\mathbf{SI})(\mathbf{SII})\big(\mathbf{B}(\mathbf{SI})(\mathbf{SII})\big).$$

This combinator expression may be slightly simplified by evaluating the $\mathbf{B}$ combinator rule, as follows:

$$\mathbf{B}(\mathbf{SI})(\mathbf{SII})\big(\mathbf{B}(\mathbf{SI})(\mathbf{SII})\big) \longrightarrow \mathbf{SI}\Big(\mathbf{SII}\big(\mathbf{B}(\mathbf{SI})(\mathbf{SII})\big)\Big).$$

Thus

$$\mathbf{Y} \stackrel{\text{def}}{=} \mathbf{SI}\Big(\mathbf{SII}\big(\mathbf{B}(\mathbf{SI})(\mathbf{SII})\big)\Big).$$

Thanks to the $\mathbf{Y}$ combinator, addition of two numbers, for instance, may now be defined recursively, as follows:

$$+xy \stackrel{\text{def}}{=} (\mathcal{Z}x)y\big(\mathfrak{S}(+(\mathfrak{P}x)y)\big).$$

The sum of x and y ($+xy$) is obtained by first testing if x is 0. If it is, then the sum of x and y is y. If x is not 0, then $+xy$ outputs the successor of the sum of the predecessor of x and y. For example, the sum of 2 and 3 would be evaluated, informally, as follows:

$$+23 \longrightarrow \mathfrak{S}(+13) \longrightarrow \mathfrak{S}\big(\mathfrak{S}(+03)\big) \longrightarrow \mathfrak{S}(\mathfrak{S}3) \longrightarrow \mathfrak{S}4 \longrightarrow 5.$$

Since $+xy$ is defined in terms of itself, $+$ is a recursive function. In order to compile this function $+$ into a combinator expression, the referral of $+$ to itself needs to be removed first. This is accomplished by the compiler first defining the following very similar function to $+$:

$$\mathbf{F}fxy \stackrel{\text{def}}{=} (\mathcal{Z}x)y\big(\mathfrak{S}(f(\mathfrak{P}x)y)\big).$$

Essentially, $\mathbf{F}$ is like $+$, but it has an extra input f that replaces all instances of $+$ on the right-hand side. This new function $\mathbf{F}$ is, of course, not recursive, so it may be compiled into a combinator expression:

$$\mathbf{F} \stackrel{\text{def}}{=} \mathbf{B}\big(\mathbf{S}(\mathbf{BS}\mathcal{Z})\big)\Big(\mathbf{B}\big(\mathbf{B}(\mathbf{B}\mathfrak{S})\big)(\mathbf{CB}\mathfrak{P})\Big).$$

Finally, $+$ is compiled using the $\mathbf{Y}$ combinator and the above $\mathbf{F}$ function, simply as:

$$+ \stackrel{\text{def}}{=} \mathbf{YF}.$$

Functional Programming

The combination of the previous sections has produced the makings of a crude functional programming language, which first compiles statements into combinator expressions, and then evaluates the user's input. The combinator expressions would be hidden from the user's view. In this section, therefore, the internal combinator expressions are hidden from the reader too.

For example, below is a possible definition of subtraction of two numbers, made in our crude programming language:

$$-xy \stackrel{\text{def}}{=} (\mathcal{Z}y)x\big(-(\mathfrak{P}x)(\mathfrak{P}y)\big).$$

$-xy$ means $x - y$ here. The subtraction $-xy$ is equal to x if y is zero, otherwise it is equal to the subtraction of the predecessor of y from the predecessor of x.

Note that if x is smaller than y, then $-xy$ returns 0. This can be used to test whether x is less than or equal to y, by defining the $\leq$ predicate in our functional programming language:

$$\leq xy \stackrel{\text{def}}{=} \mathcal{Z}(-xy).$$

Other relational tests may easily be written, too:

$$\geq xy \stackrel{\text{def}}{=} \mathcal{Z}(-yx);$$
$$< xy \stackrel{\text{def}}{=} \neg(\geq xy);$$
$$> xy \stackrel{\text{def}}{=} \neg(\leq xy);$$
$$= xy \stackrel{\text{def}}{=} \wedge(\leq xy)(\leq yx).$$

Note the use of the NOT ($\neg$) and AND ($\wedge$) logical connectives that were defined in an earlier section.

To conclude, here is a short program that produces the nth Fibonacci number, that is, the nth number of the sequence

$$1,1,2,3,5,8,13,21, \ldots$$

in which the first two elements are both 1 and whose subsequent elements are the sum of the previous two:

$$Qp \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{R}p)\big(+(\mathcal{L}p)(\mathcal{R}p)\big)$$
$$Anfx \stackrel{\text{def}}{=} (\mathcal{Z}n)x\big(f(A(\mathfrak{P}n)fx)\big)$$
$$Fn \stackrel{\text{def}}{=} \mathcal{L}\big(AnQ(\mathcal{P}01)\big)$$

The function Q takes a pair p as input and outputs the pair whose left element is the right element of p and whose right element is the sum of the elements of p. For example, $Q(\mathcal{P}35)$ outputs $\mathcal{P}58$.

The function A takes three inputs n, f and x and outputs f applied to x n times. For example, $A2fx$ outputs $f(fx)$, while $A3\mathfrak{S}2$ outputs $\mathfrak{S}\big(\mathfrak{S}(\mathfrak{S}2)\big)$, which represents the number 5.

Finally, the function F is our Fibonacci function. It takes a number n as input and applies the function Q to the pair $\mathcal{P}01$ n times. After doing this, F extracts the left element of the resulting pair and produces it as its output. For example, $F4$ reduces to 3, informally, as follows:

$$F4 \longrightarrow \mathcal{L}\left(Q\left(Q\left(Q(Q(\mathcal{P}01))\right)\right)\right) \longrightarrow \mathcal{L}\left(Q\left(Q(Q(\mathcal{P}11))\right)\right) \longrightarrow \mathcal{L}\left(Q(Q(\mathcal{P}12))\right)$$
$$\longrightarrow \mathcal{L}\big(Q(\mathcal{P}23)\big) \longrightarrow \mathcal{L}(\mathcal{P}35) \longrightarrow 3.$$

In a real-life functional programming language such as Haskell, the above program would have a much more readable form:

```
FibPairNext (a,b) = (b,a+b)
Apply 0 _ x = x
Apply n f x = f (Apply (n-1) f x)
Fibonacci n = fst (Apply n FibPairNext (0,1))
```

However, this is only a question of syntax. What was presented in this paper applies to the possible implementation of any functional programming language, including Haskell.

References

Barendregt, H. P. (1976). A Global Representation of the Recursive Functions in the Lambda-calculus. *Theoretical Computer Science*, 3, 225–242.

Barendregt, H. P. (1984). *The Lambda Calculus, Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics.* Volume 103. North Holland.

Bernays, P. & Schönfinkel, M. (1929). Zum Entscheidungsproblem der mathematischen Logik. *Mathematische Annalen* (in German). 99, 342–372.

Bimbó, K. Combinatory Logic. *The Stanford Encyclopedia of Philosophy (Fall 2018 Edition) Edward N. Zalta (ed.)*. 10 November 2016. Available at https://plato.stanford.edu/archives/fall2018/entries/logic-combinatory/ [Accessed 9 August 2018]

Bird, R. & Wadler, P. (1988). *Introduction to Functional Programming*. New York: Prentice Hall.

Church, A. (1933). A set of postulates for the foundation of logic (second paper). *The Annals of Mathematics Second Series*, 34(4), 839–864.

Church, A. (1940). A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic,* 5(2), 56–68.

Curry, H. B. & Feys, R. (1958) *Combinatory Logic, Volume I*. Amsterdam: North-Holland Co.

Hindley, J. R., & Seldin, J. P. (2008). *Lambda-calculus and Combinators, an Introduction.* Cambridge: Cambridge University Press.

Hudak, P., Hughes, J., Peyton Jones, S. and Wadler, P. (2007). A History of Haskell: Being Lazy with Class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, 12, 1–55.

Kline, G. L. & Anovskaa, S. A. (1951). Review of Foundations of Mathematics and Mathematical Logic. *Journal of Symbolic Logic*, 16(1), 46–48.

Mendelson, E. (1997). *Introduction to Mathematical Logic*. London: Chapman & Hall.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. New York: Prentice-Hall.

Pierce, B. C. (2002). *Types and Programming Languages.* Cambridge: MIT Press.

Quine, W. V. O. (1967). On the building blocks of mathematical logic. In J. van Heijenoort, *A Source Book in Mathematical Logic, 1879–1931*, USA: Harvard University Press, 355–366.

Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. *Mathematische Annalen* (in German), 92, 305–316.

Smullyan, R. (2000). *To Mock a Mockingbird and Other Logic Puzzles Including an Amazing Adventure in Combinatory Logic*. Oxford: Oxford University Press.

Turing, A. M. (1937). The p-function in lambda-K-conversion. *Journal of Symbolic Logic*, 2, 164.

Bio-note

Dr Alexander Farrugia is Lecturer of Mathematics at the University of Malta, Junior College and a part-time lecturer at the University of Malta. He was awarded a BSc. (Hons) in Mathematics and Computer Science in 2001 and an MSc in Mathematics in 2003 by the University of Malta. In 2016, he obtained his PhD from the same university, studying under the supervision of Professor Irene Sciriha. His area of expertise includes algebraic and spectral graph theory, with applications in molecular chemistry and control theory. In his free time, he writes about mathematics on Quora, a website where people share their knowledge with the world. He was recognised as a Top Writer on Quora in 2017, with his writings garnering more than 6.7 million views (as of February 2018).