Athens Journal of Sciences- Volume 9, Issue 1, March 2022 – Pages 47-64

The Fixpoint Combinator in Combinatory Logic – A Step towards Autonomous Real-time Testing of Software?

By Thomas Fehlmann[*] *& Eberhard Kranich*[±]

Combinatory Logic is an elegant and powerful logical theory that is used in computer science as a theoretical model for computation. Its algebraic structure supports self-application and is Turing-complete. However, contrary to Lambda Calculus, it untangles the problem of substitution, because bound variables are eliminated by inserting specific terms called Combinators. It was introduced by Schönfinkel (1924) and Curry (1930). Combinatory Logic uses just one algebraic operation, namely combining two terms, yielding another valid term of Combinatory Logic. Terms in models of Combinatory Logic look like some sort of assembly language for mathematical logic. A Neural Algebra, modeling the way we think, constitutes an interesting model of Combinatory Logic. There are other models, also based on the Graph Model (Engeler 1981), such as software testing. This paper investigates what Combinatory Logic contributes to modern software testing.

Keywords: *combinatory logic, combinatory algebra, autonomous real-time testing, recursion, software testing, artificial intelligence*

The Organon

Aristotle's legacy regarding formal logic has been transferred to us in a collection of his thoughts compiled into a set of six books called the *Organon* around 40 BCE by Andronicus of Rhodes or others among his followers (Aristoteles 367-344 BCE). The Organon with its syllogisms was the dominant form of Western logic until 19th-century advances in mathematical logic.

Engeler recently noted the apparent lack of something that we today consider fundamental for axiomatic geometry: relations. The question is why. Aristotle had the means of developing this concept as well; however, he chose not to do so.

Aristotle had the means of combining predicates. It is therefore possible to construct an adequate model for Aristotle' syllogism based on the structures of Combinatory Logic. Relations then become part of the model. Engeler shows that Aristotle had no need for relations because the main model he used – the Euclidean Geometry – does not require relations (Engeler 2020).

[*]Senior Researcher, Euro Project Office AG, Switzerland.
[±]Senior Researcher, Euro Project Office AG, Switzerland.

https://doi.org/10.30958/ajs.9-1-3 doi=10.30958/ajs.9-1-3

Introduction

A model of Combinatory Logic is an algebraic structure implementing combinators in a non-trivial way. Such a model is called *Combinatory Algebra*. As a minimum, it contains implementations for the **I** combinator (identity), the **K** combinator for extracting parts of another term, and the **S** combinator that substitutes parts of a term by some other combinator. Another famous combinator is the *Fixpoint Combinator* **Y**, explaining recursion and possible infinite iteration. These specific combinators are represented as *Constants* in the language of Combinatory Logic, whereas other terms may contain *Variables*. These general terms are called *Combinatory Terms* (Bimbó 2012, p. 2); the combination of any two terms X and Y is written as $X \bullet Y$.

Given a problem as a term X in some suitable model, what should be its solution? A problem is something that displays specific behavior, sometimes unpredictable, and produces specific effects, often unwanted. Also, a certain persistence is part of a problem; problems that disappear by themselves are not particularly exciting.

A fixpoint point $\mathbf{Y} \bullet X$ with the property that $\mathbf{Y} \bullet X = X \bullet (\mathbf{Y} \bullet X)$, for any term X of Combinatory Logic, is thus something like a solution to the problem X. You can apply the solution combinator **Y** as many times as necessary and the problem solution remains stable and confined.

When we encounter the problem of how to test a piece of software X, and we have a test suite $\mathbf{Y} \bullet X$ with the fixpoint property, it looks like a solution to our testing problem. Since we can measure tests, by counting its test size, we can assess what means minimal effort for a test, and thus can get an optimum.

The clue to Combinatory Logic is that "everything is a function" – and indeed, a unary function. Whenever anything can be understood as function depending on two variables – $f(x, y)$ – it is an application of a unary function $g = f \bullet x$ on a variable y. Thus, $f(x, y) = g(y) = (f \bullet x) \bullet y = f \bullet x \bullet y$; always assuming association to the left. This is known as *Currying*, converting n-ary functions into a sequence of unary functions.

Combinatory Algebras

Combinatory Algebras are models of Combinatory Logic (Curry and Feys 1958, Curry et al. 1972). Such algebras are closed under a combination operation $M \bullet N$ for all terms of the algebra M, N; and two distinct *Combinators* **S** and **K** can be defined with the following properties:

$$\mathbf{K} \bullet P \bullet Q = P \tag{1}$$

and

$$\mathbf{S} \bullet P \bullet Q \bullet R = P \bullet Q \bullet (P \bullet R) \tag{2}$$

where P, Q, R are elements in the combinatory algebra[1].

Thus, the combinator **K** acts as projection, and **S** is a substitution operator for terms in the combinatory algebra. Like an assembly language, the **S**-**K** terms become quite lengthy and are barely readable by humans, but they work fine as a foundation for computer science.

The power of these two operators is best understood when we use them to define further, more manageable, and more reasonable combinators. Church (Church, 1941) presented a list of functions that can be implemented as combinators, and Zachos investigated them in the settings of Combinatory Logic (Zachos 1978). Bimbó (2012, p. 6) gives a good overview; however, without reference to the original contributors. We present here only a few.

Identity

The identity combinator is defined as

$$\mathbf{I} \coloneqq \mathbf{S}\bullet\mathbf{K}\bullet\mathbf{K} \tag{3}$$

Indeed, $\mathbf{I}\bullet M = \mathbf{S}\bullet\mathbf{K}\bullet\mathbf{K}\bullet M = \mathbf{K}\bullet M\bullet(\mathbf{K}\bullet M) = M$. Association is to the left.

Functionality by the Lambda Combinator

Church's *Lambda Calculus* is a formal language that can be understood as a prototype programming language (see Church 1941, Barendregt 1977).

Lambda calculus can be expressed by **S**-**K** terms. We define recursively the *Lambda Combinator* $\mathbf{L_x}$ for a variable x as follows:

$$\mathbf{L_x}\bullet x = \mathbf{I} \tag{4}$$

$$\mathbf{L_x}\bullet M = \mathbf{K}\bullet M \text{ if } M \text{ different from } \mathbf{x} \tag{5}$$

$$\mathbf{L_x}\bullet M\bullet N = \mathbf{S}\bullet\mathbf{L_x}\bullet M\bullet(\mathbf{L_x}\bullet N) \tag{6}$$

The definition (5) holds for any variable term **x** in the combinatory algebra. We can extend the definition of the Lambda combinator by getting rid of the specific variable x. For any combinatory term M, the *Abstraction Operator* $\lambda x.$ is defined on M recursively by applying $\mathbf{L_x}$ to all sub-terms of M. Applying $\lambda x.M$ to any other combinatory term N replaces all occurrences of the variable x in the term M by N and is written as $(\lambda x.M)\bullet N$.

The abstraction operator binds weaker than the combination operator. Thus, $\lambda x.$ binds all variables x in $M\bullet N$, such that we can omit parentheses as in $\lambda x.M\bullet N = \lambda x.(M\bullet N)$. Lambda abstraction provides a much more readable and intuitively understandable notation for terms of Combinatory Logic.

[1]The use of variables named P, Q, R is borrowed from Engeler (2020).

The Fixpoint Combinator

Given any combinatory term Z, the *Fixpoint Combinator* **Y** generates a combinatory term $\mathbf{Y}\bullet Z$, called *Fixpoint of* Z**,** that fulfills $\mathbf{Y}\bullet Z = Z\bullet(\mathbf{Y}\bullet Z)$. This means that Z can be applied as many times as wanted to its fixpoint and still yields back the same combinatory term.

In linear algebra, such fixpoint combinators yield an eigenvector solution to some problem Z; for instance, when solving a linear matrix. It is therefore tempting to say, that $\mathbf{Y}\bullet Z$ is a solution for the problem Z. For more details, consult Fehlmann (2016).

Using Lambda Calculus notation, the fixpoint combinator can be written as (Barendregt 1984):

$$\mathbf{Y} := \lambda f.\big(\lambda x.\, f\bullet(x\bullet x)\big)\bullet\big(\lambda x.\, f\bullet(x\bullet x)\big) \tag{7}$$

Translating (7) into an **S**-**K** term proves possible, becomes a bit lengthy but demonstrates how Combinatory Logic works.

By applying (6), (5):

$$\begin{aligned}\lambda x.\, f\bullet(x\bullet x) &= \mathbf{S}\bullet\lambda x.\, f\bullet\lambda x.\, x\bullet x\\ \lambda x.\, f &= \mathbf{K}\bullet f\end{aligned}$$

Then applying (6) and (4)

$$\begin{aligned}\lambda x.\, x\bullet x &= (\mathbf{S}\bullet\lambda x.\, x\bullet\lambda x.\, x)\\ &= (\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})\end{aligned}$$

yields

$$\lambda x.\, f\bullet(x\bullet x) = \mathbf{S}\bullet(\mathbf{K}\bullet f)\bullet(\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})$$

and therefore

$$\begin{aligned}\mathbf{Y} &= \lambda f.\big(\lambda x.\, f\bullet(x\bullet x)\big)\bullet\big(\lambda x.\, f\bullet(x\bullet x)\big)\\ &= \lambda f.\big(\mathbf{S}\bullet(\mathbf{K}\bullet f)\bullet(\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})\big)\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet f)\bullet(\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})\big)\\ &= \mathbf{S}\bullet\big(\lambda f.\,\mathbf{S}\bullet(\mathbf{K}\bullet f)\bullet(\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})\big)\bullet\big(\lambda f.\,\mathbf{S}\bullet(\mathbf{K}\bullet f)\bullet(\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I})\big)\\ &= \mathbf{S}\bullet\big(\mathbf{S}\bullet\big(\lambda f.\,\mathbf{S}\bullet(\mathbf{K}\bullet f)\big)\bullet\lambda f.\,\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I}\big)\bullet\big(\mathbf{S}\bullet\big(\lambda f.\,\mathbf{S}\bullet(\mathbf{K}\bullet f)\big)\bullet\lambda f.\,\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I}\big)\end{aligned}$$

Now solving the remaining λ-terms:

$$\begin{aligned}\lambda f.\,\mathbf{S}\bullet(\mathbf{K}\bullet f) &= \mathbf{S}\bullet\lambda f.\,\mathbf{S}\bullet\lambda f.\,\mathbf{K}\bullet f\\ &= \mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{S}\bullet\lambda f.\,\mathbf{K}\bullet\lambda f.\, f)\\ &= \mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{K})\bullet\mathbf{I})\end{aligned}$$

considering

$$\lambda f.\,\mathbf{S} = \mathbf{K}\bullet\mathbf{S}$$

and

$$\lambda f.\,\mathbf{K}\bullet f = \mathbf{S}\bullet\lambda f.\,\mathbf{K}\bullet\lambda f.\, f = \mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{K})\bullet\mathbf{I}.$$

The latter holds by first applying (6), then (5) and (4). Moreover

$$\begin{aligned}\lambda f.\,\mathbf{S}\bullet\mathbf{I}\bullet\mathbf{I} &= \mathbf{S}\bullet\lambda f.\,\mathbf{S}\bullet\mathbf{I}\bullet\lambda f.\,\mathbf{I}\\ &= \mathbf{S}\bullet(\mathbf{S}\bullet\lambda f.\,\mathbf{S}\bullet\lambda f.\,\mathbf{I})\bullet(\mathbf{K}\bullet\mathbf{I})\\ &= \mathbf{S}\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{K}\bullet\mathbf{I})\big)\bullet(\mathbf{K}\bullet\mathbf{I})\end{aligned}$$

applying again (6) and (5).

Putting things together:

$$\begin{aligned}\mathbf{Y} &= \lambda f.\big(\lambda x.\,f\bullet(x\bullet x)\big)\bullet\big(\lambda x.\,f\bullet(x\bullet x)\big)\\ &= \mathbf{S}\bullet\Big(\mathbf{S}\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{K})\bullet\mathbf{I})\big)\bullet\Big(\mathbf{S}\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{K}\bullet\mathbf{I})\big)\bullet(\mathbf{K}\bullet\mathbf{I})\Big)\Big)\\ &\quad\bullet\Big(\mathbf{S}\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{K})\bullet\mathbf{I})\big)\bullet\Big(\mathbf{S}\bullet\big(\mathbf{S}\bullet(\mathbf{K}\bullet\mathbf{S})\bullet(\mathbf{K}\bullet\mathbf{I})\big)\bullet(\mathbf{K}\bullet\mathbf{I})\Big)\Big)\end{aligned}$$

Applying **Y** to any combinatory term Z now explicitly transports Z on the top of the formula and keeps the rest of the structure of **Y** such that **Y** can be applied repeatedly.

This exercise should give the reader an impression how Combinatory Logic works.

Applying the fixpoint combinator **Y** to some combinator Z in the Lambda-style is much simpler:

$$\begin{aligned}&\lambda f.\big(\lambda x.\,f\bullet(x\bullet x)\big)\bullet\big(\lambda x.\,f\bullet(x\bullet x)\big)\bullet Z\\ &= \big(\lambda x.\,Z\bullet(x\bullet x)\big)\bullet\big(\lambda x.\,Z\bullet(x\bullet x)\big)\\ &= Z\bullet\big(\lambda x.\,Z\bullet(x\bullet x)\big)\bullet\big(\lambda x.\,Z\bullet(x\bullet x)\big)\end{aligned}$$

by applying the Lambda combinator twice, replacing the two $x\bullet x$ twice by $\lambda x.\,Z\bullet(x\bullet x)$. Thus, this explains reasoning as a repeated substitution process.

When applying **Y**, or $\mathbf{Y}'$, or any other equivalent fixpoint combinator to a combinatory term Z, reducing the term by repeatedly using rule (1) or (2) does not always terminate. An infinite loop can occur, and must sometimes occur, otherwise we would always find a solution to any problem that can be stated within a programming language. Thus, Turing would be wrong and all finite state machines would reach a finishing state (Turing 1937).

Thus, the fixpoint combinator is not the solution of all our practical problems. But Engeler teaches us in his Neural Algebra fixpoints can be approximated using a *Construction Operator* (Engeler 2019), see below.

For more details about the foundations of Mathematical Logic, see for instance Potter (2004).

Arrow Terms

The *Graph Model* of *Combinatory Logic* (Engeler 1995) is a model of Combinatory Logic with explains how to combine topics in areas of knowledge. Combination is not only on the basic level possible; you can also explain how to

combine topics on the second level; sometimes called meta-level. Intuitively, we would expect such a meta-level describing knowledge about how to deal with different knowledge areas.

Whenever two terms M and N are embodied in a combinatory algebra, the application of M onto N is also a term of this combinatory algebra, denoted as $M \bullet N$.

Let $\mathcal{L}$ be the set of all assertions over a given domain. Examples include statements about customer's needs, solution characteristics, methods used, program states, test conditions, etc. These statements are assertions about the domain we are dealing with. This could be a business domain, or the state of some software, i.e., the description of the values for all controls and data.

An *Arrow Term* is recursively defined as follows:

- Every element of $\mathcal{L}$ is an arrow term.
- Let $a_1, \dots, a_m, b$ be arrow terms. Then

$$\{a_1, \dots, a_m\} \to b \tag{8}$$

is also an arrow term. Thus, arrow terms are relations between finite subsets of arrow terms and another arrow term, emphasized as successor.

For instance, in software testing, we use arrow terms to represent test cases. On the base level, the left-hand sides $a_1, \dots, a_m$ represent test data, the term b is the known expected response of the test case (8). Higher levels of arrow terms represent test strategies and tests of tests.

The left-hand side of an arrow term is a finite set of arrow terms and the right-hand side is a single arrow term. This definition is recursive. The arrows are a formal graph notation; they might suggest cause-effect, not logical imply.

The Graph Model as an Algebra of Arrow Terms

We can extend the definition of arrow terms to become a combinatory algebra, allowing for the combination of arrow terms.

Denote by $\mathcal{G}(\mathcal{L})$ the power set containing all *Arrow Terms* of the form (8).

The formal, recursive, definition of the Graph Model as a power set, in set-theoretical language, is given in equation (9):

$$\begin{gathered} \mathcal{G}_0(\mathcal{L}) = \mathcal{L} \\ \mathcal{G}_{n+1}(\mathcal{L}) = \\ \mathcal{G}_n(\mathcal{L}) \cup \left\{ \{a_1, \dots, a_m\} \to b \middle| a_1, \dots, a_m, b \in G_n(\mathcal{L}), m \in \mathbb{N} \right\} \end{gathered} \tag{9}$$

for $n = 0, 1, 2, \dots$ $\mathcal{G}(\mathcal{L})$ is the set of all (finite and infinite) subsets of the union of all $\mathcal{G}_n(\mathcal{L})$:

$$\mathcal{G}(\mathcal{L}) = \bigcup_{n \in \mathbb{N}} \mathcal{G}_n(\mathcal{L}) \tag{10}$$

The elements of $\mathcal{G}_n(\mathcal{L})$ are arrow terms of level n. Terms of level 0 are *Assertions*, terms of level 1 *Rules*. A set of rules is called *Rule Set* (Fehlmann 2016). In general, a rule set is a finite set of arrow terms. We call infinite rule sets a *Knowledge Base*. Hence, knowledge is a potentially unlimited collection of rules sets containing rules about assertions regarding our domain.

Combining Knowledge Bases

We can combine knowledge bases sets as follows:

$$M \bullet N = \{c | \exists \{b_1, b_2, \dots, b_m\} \to c \in M;\ b_i \in N\} \tag{11}$$

Arrow Term Notation

To avoid the many set-theoretical parenthesis, the following notations, called *Arrow Schemes*, are applied:

- a_i for a finite set of arrow terms, i denoting some finite indexing function for arrow terms.
- a_1 for a singleton set of arrow terms: $a_1 = \{a\}$ for an arrow term a.
- $\emptyset$ for the empty set, such as in the arrow term $\emptyset \to a$.
- $a_i \cup b_j$ for the union of two sets a_i and b_j of arrow terms.
- (a) for a potentially infinite set of arrow terms, where a is an arrow term.

Note that arrow schemes denote sets when put into outermost parenthesis. Without an index, the set might be infinite; an index makes the set finite.

The indexing function cascades; thus, $a_{i,j}$ denotes the union of a finite number of sets of arrow schemes

$$a_{i,j} = a_{i,1} \cup a_{i,2} \cup \dots \cup a_{i,j} \cup \dots \cup a_{i,m} = \bigcup_{k=1}^{m} a_{i,k} \tag{12}$$

In terms of these conventions, $(x_i \to y)_j$ denotes a rule set; i.e., a non-empty finite set of arrow terms, each having at least one arrow. Thus, such set has level 1 or higher. Moreover, it has two selection functions, i and j, selecting a finite number of arrow terms for x and $x_i \to y$.

With this notation, the application rule for M and N reads:

$$M \bullet N = \big((b_i \to a) \bullet (b_i)\big) = \{a | \exists b_i \to a \in M;\ b_i \subset N\} \tag{13}$$

Arrow Terms – A Model of Combinatory Logic

The algebra of arrow terms is a combinatory algebra and thus a model of Combinatory Logic. It is called the *Graph Model*.

The following definitions demonstrate how arrow terms implement the combinators **S** and **K** fulfilling equations (1) and (2).

- $\mathbf{I} = (a_1 \to a)$ is the *Identification*; i.e., $(a_1 \to a) \bullet (b) = (b)$
- $\mathbf{K} = (a_1 \to \emptyset \to a)$ selects the 1st argument:
  $\mathbf{K} \bullet (b) \bullet (c) = \big((b_1 \to \emptyset \to b) \bullet (b)\big) \bullet (c) = (\emptyset \to b) \bullet (c) = (b)$
- $\mathbf{KI} = (\emptyset \to a_1 \to a)$ selects the 2nd argument:
  $\mathbf{KI} \bullet (b) \bullet (c) = \big((\emptyset \to c_1 \to c) \bullet (b)\big) \bullet (c) = (c_1 \to c) \bullet (c) = (c)$
- $\mathbf{S} = \Big(\big(a_i \to \big(b_j \to c\big)\big)_1 \to (d_k \to b)_i \to \big(a_i \cup b_{j,i} \to c\big)\Big)$

Therefore, the algebra of arrow terms is a model of Combinatory Logic.

The proof that the arrow terms' definition of **S** fulfils equation (2) is somewhat more complex. Readers interested in that proof are referred to Engeler (1981, p. 389). With **S** and **K,** an abstraction operator can be constructed that builds new knowledge bases. This is the *Lambda Theorem*; it is proved along the same way as Barendregt's Lambda combinatory (Barendregt 1977). See also in Fehlmann (1981, p. 37).

The Role of the Indexing Function in Arrow Terms

The arrow in the terms of the Graph Model is somewhat confusing. It is easily mistaken as representing *Predicate Logic*; however, this must be viewed with care. Interpreting the arrow as an implication in predicate logic is not per se dangerous. In some sense, logical imply is a transition from preconditions to conclusion and arrow terms are fine for representing them. The problem is that if the left-hand side of an arrow term, which is an otherwise unstructured set, is interpreted as a conjunction of predicates – a sequence of logical AND-clauses – you run into a conflict with the undecidability of first-order logic. Arrow terms would then reduce to either of the form $a_1 \to b$ or $\emptyset \to b$. This reduces the model to become the trivial one.

As an example, see Bimbó (2012, p. 237ff). There she explains how typed Combinatory Logic gets around the triviality problem. Instead of the indexing functions, selecting finite sets of arrow terms on the left-hand side, she postulates proofs for the predicates.

Thus, the indexing function for selecting elements of a finite set of arrow terms is a key element of the Graph Model. Interested readers will find related considerations in the paper of Fehlmann and Kranich (2020). For the application of the Graph Model to testing, the indexing function means selection of test cases and test data, and this is always a collection of program state predicates that do typically not leave the program under test in a consistent state.

Neural Algebra

Engeler uses the Graph Model as a model how the brain thinks (Engeler 2019). A directed graph, together with a firing law at all its nodes, constitutes the connective basis of the brain model $\mathcal{A}$. The model itself is built on this basis by identifying brain functions with parts of the firing history. Its elements may be visualized as a directed graph, whose nodes indicate the firing of a neuron. As before, we consider $\mathcal{G}(\mathcal{A})$, constructed as in (10). The elements of $\mathcal{G}(\mathcal{A})$ are called *Cascades*. Cascades describe firing between nodes (neurons) when represented by finite sets of arrow terms $a_i \to b$ where a_i are sub-cascades, while the right sub-cascade b describes the characteristic leave of its firing history graph. The *Neural Algebra* is defined as a collection of cascades representing brain functions in the brain model, closed under applications and union. With the application rule (13), we have an algebraic structure; the application representing brain functions, interpreted as thoughts.

The Fixpoint Combinator in the Neural Algebra

The fixpoint combinator **Y** can be written as an arrow scheme; however, this calculation is better left to some suitable rewriting tool, as otherwise this article would exceed all reasonable length. Applying **Y** to an arbitrary arrow scheme might result in an infinite loop of arrow schemes, representing a never-ending computation. Combinatory Logic, as any kind of programming, may result in an infinite loop in its model, and it is not decidable when this happens.

If infinite loops occur, or infinite sequences of digits like for real numbers that are not rationales, we need the notion of controlling operators that approximate the possibly infinite solution, and metrics for measuring how near the approximations to the solutions are, and get even nearer when required.

Reasoning, Problem Solving and Controlling

Within this setting, it is possible to define models for reasoning and problem solving. However, not only flat reasoning, also for solving problems, even if their fixpoint is infinite. For a controlled object X, the *Controlling Operator* $\mathbf{C}$ solves the control problem $\mathbf{C} \bullet X = X$. The brain function $\mathbf{C}$ gathers all faculties that may help in the solution. The control problem is a repeated process of substitution, like finding the fixpoint of a combinator. However, since cascades are always finite – all brain activity remains finite, unfortunately – solving the control problem is by a series of finite *Attractors*, a control sequence $X_0 \subseteq X_1 \subseteq X_2 \subseteq \cdots$ determined by

$$X_{i+1} = \mathbf{C} \bullet X_i, i \in \mathbb{N} \tag{14}$$

starting with an initial X_0. This process is called *Focusing*. The details can be found in Engeler (2019, p. 301). We will rely on the observation that attractors represent reasoning in a neural algebra.

Attractors are ordered by inclusion (14), meaning that the solution space becomes smaller and smaller until a smallest possible solution is found that cannot be further reduced by the controlling operator. It is possible that this ultimate solution remains empty.

The controlling operator is closely linked to the fixpoint operator **Y.** If X has a solution $\mathbf{C} \bullet X$, then X is of the form $X = \mathbf{Y} \bullet Z$ for some suitable cascade Z. Thus, not all combinators X have a solution; the related control sequence may end with the empty cascade, obviously. These considerations share a stunning resemblance with transfer functions, whose solution profiles are also approximations rather than precise solutions (Fehlmann 2016, p. 14).

The controlling operator is not like one of the basic or the fixpoint combinators but is more of a prescription, how to find suitable attractors. Engeler (2019, p. 300ff) presents in an elegant way representations of basic thought processes, e.g., reflection, discrimination, simultaneous and joint control, but also learning, teaching, focusing with eyes, and comprehension.

Since the number of cascades that a brain can produce is finite and limited – by the lifespan of the brain – solution to the fixpoint control problems turn out to be finite attractor sequences, characterizing thought processes.

Transfer Functions

For managing complex systems, transfer functions are used to analyze controls and approximate the expected result (Fehlmann 2016).

An obvious interpretation of arrow terms is by transfer functions. In *Quality Function Deployment* (QFD), the building blocks – and the origin – are cause-effect relations as used in Ishikawa Diagram (Ishikawa and Loftus 1990). These diagrams describe the cause-effect relations between topics and are considered the initial form of QFD matrices. Converting a series of Ishikawa diagrams into a QFD matrix is straightforward (see Fehlmann 2016, p. 231). Thus, transfer functions can be described by finite sets of arrow terms.

Deming Chains

Composition of transfer functions is called a *Deming Chain* (Fehlmann 2016, p. 100) because Deming identified the value chains in manufacturing processes (Deming 1986). Akao called it *Comprehensive QFD*, also known as *QFD in the Large*. He drafted extensive Deming chains in this famous book on QFD (Akao 1990).

For transfer functions, the Graph Model provides similar services as for tests. The model proves that transfer functions have universal applicability and power for explaining cause-effect, and they provide a framework for automation also for Deming chains (Fehlmann 2001).

The Algebra of Tests

Very interesting instantiations of the Graph Model can be found in *Software Testing*, especially when seen from an economical viewpoint. In fact, test cases are best described as arrow terms, with the left-hand sides describing program states before executing the test, and the right-hand side describing the response of the test case. Software testing is the key to digitalization and to software-intense products that perform safety-critical tasks.

Test cases are a mapping of arrow terms onto *Data Movement Maps*. Data movement maps model the software under test by identifying the data groups moved by the software, based on the ISO standard 19761 COSMIC (ISO/IEC 19761 2011). This has been explained in more detail in Fehlmann (2020). The data movements induce a sizing valuation on this algebra by counting the number of data movements executed per test case.

When we speak of test cases, we always intend a suitable data movement map with it; thus, the same arrow term can be mapped to several data movement maps, counting as separate test cases.

State Assertions

For our Test Algebra, we now assume $\mathcal{L}$ to be the set of all state assertions for a given program. We use the term "program" but mean a system that might consist of coded software, services, or anything yielding results electronically. Learning machines also are "programs" in that sense even if it is not the code that implements learning, rather the learned knowledge itself. Elements of $\mathcal{L}$ are descriptions of the system status, or the knowledge such as system has, at a certain moment. In the sequel, the arrow term $a_i \to b$ together with its associated data movement map represents a test case, that, given test data a_i, yields b as the expected, correct result (Fehlmann 2020, p. 85ff).

If $a_i \to b$ is a test case, $a_i \subset \mathcal{L}$ specifies a set of test data that holds before executing the test, and $b \in \mathcal{L}$ the state of the program after execution. The finite set a_i represents the states before execution of possible unrelated threads of the program, or services involved.

Testing Complex Systems

Usually, unit tests that ensure the proper functioning of software modules are available because they originate from the software development process (Junit Team 1997ff). The integration of modules and components and building systems of systems, or other complex products, requires many more tests, among them end-to-end tests that cause huge efforts. Most often, the time slots available for testing are used up to accommodate additional or forgotten user requirements. Consequently, with respect to functionality, the more important tests become when creating complex products, the less tests are executed, by lack of time and resources. Attempts to execute tests automatically do not address the lack of good test cases for complex products. There are the test cases that need to be created

automatically. This approach is called *Autonomous Real-time Testing*, to point out that testing effort always must remain limited. It addresses the problem how to automatically create test cases by Artificial Intelligence, namely by generating test cases using Combinatory Logic and selecting the relevant ones using ISO/IEC 16355 (ISO/IEC 16355-1 2015). The approach is explained in Fehlmann (2020).

Combining Tests

The definition (13) explains how to combine test cases. To apply one set M of test cases to another N, it is required that for testing the assertion a, test cases $b_i \subset N$ exist such that $(b_i \to a)$ in M has effectively been tested. Consult last the paper of Fehlmann and Kranich (2020) for more information about the existential quantifier in (13).

The intuitionistic, or constructive, variant of the Axiom of Choice requires not only the existence of test providing valid test data as response, but construction instructions for the existence of such tests, respectively the related test cases. This means that it is not enough to know the existence of tests, but you need to know how to construct them. This is possibly the reason why test automation has proven to be so hard.

And for those who consider such reasoning too theoretical, let us provide a rather practical argument: programmers who want to set up test concatenation $M \bullet N$ for automatic testing, need access to the test cases in N that provide the responses needed for M, for combining M with N. Otherwise, combining tests is unsafe or cannot be automated. Thus, with the combinatory algebra of arrow terms, mathematical logic meets both intuitionism and programming.

Combination Limitations

Combining tests in a Combinatory Algebra is unlimited indeed because there is no typing involved that governs applicability. By (13), you can combine test cases across test stories as deemed appropriate; all that counts are that the test cases remain executable. This means that two test cases must not only be linked by its assertions, but also executable code must exist that combine these two test cases. In terms of software, two data movement maps representing the test case executions must exist that overlap.

The Size of Tests

For a testing framework, we need to be able to measure the size of tests. The standard ISO/IEC 19761 COSMIC for measuring functional size serves as measuring method. The functional size of the associated data movement map is the size of a test case, denoted by $Cfp(a_i^0 \to b^0)$, where $a_i^0 \in \mathcal{G}_0(\mathcal{L})$ and $b^0 \in \mathcal{G}_0(\mathcal{L})$ are arrow terms of level 0; i.e., assertions about the state of the program. $Cfp(a_i^0 \to b^0)$ is the number of unique data movements touched when executing the test case $a_i^0 \to b^0$. This is the recursion base.

Then the following equations (15) recursively define the size of tests:

$$\begin{gathered} \lfloor a \rfloor = 0 \text{ for } a \in \mathcal{G}_0(\mathcal{L}) \\ \lfloor a_i \to b \rfloor = Cfp(a_i \to b) \text{ for } a_i \subset \mathcal{G}_0(\mathcal{L}) \text{ and } b \in \mathcal{G}_0(\mathcal{L}) \\ \lfloor c_i \to d \rfloor = \sum_i \lfloor c_i \rfloor + \lfloor d \rfloor \text{ for all test cases } c_i \text{ and } d \end{gathered} \tag{15}$$

The definition holds for all arrow terms in the algebra of tests.

The addition does not take into consideration whether data movements are unique; thus, the size of two test cases is always the sum of the sizes. When speaking about tests, we do not use the term knowledge base for sets of arrow terms, but rather *Test Story* for a set of test cases. Test stories typically share a common intent, or business value.

The Functional Size of Combinators

Applying the definition (15) to the combinators $\boldsymbol{S}, \boldsymbol{K}, \boldsymbol{I}$, and $\boldsymbol{Y}$ yields an infinite size for each of them, because the arrow term sets are infinite. This is conformant to the observation that when expressing these combinators as terms in the Lambda calculus, they are closed insofar as they do not contain free variables nor constants.

Autonomous Real-time Testing

In Fehlmann (2020), we coined the term *Autonomous Real-time Testing* (ART) to describe software tests that are

- Executed automatically in a system during operations, or when pausing operations;
- Started from a base test using recombination and other operations of combinatory algebra by adding autonomously generated test cases;
- Controlled by transfer functions assuring relevance for users' values.

In previous papers and the book referenced about, we have explained how to keep the growth of test cases under control, using the *Convergence Gap* as a hash. The convergence gap in transfer functions measures the gap between the needs – of the customer, the user, certification authority, or else – and the achieved test coverage. Consult the paper of Fehlmann and Kranich (2020).

Attractors

While the fixpoint combinator $\boldsymbol{Y}$ works as above on sets of test cases, in most cases, it returns infinite tests as "solutions" – something not too practical. However, we can construct attractors for neural algebra, approximating the infinite testing set, as good as we wish. This creates a new problem for us, namely, to assess: when is testing good enough?

While good practices can provide answers – e.g., by looking at the remaining defect rate (Fehlmann and Kranich 2014) – a more theoretical answer should include at least the requirement that attractors cover functionality. That is the significance of the *Convergence Gap*, explained in Fehlmann (2020, p. 10).

Let U_l denote a finite set of user stories, and T_k another set of test stories, usually somewhat larger than the set of user stories. The matrix $U_l \otimes T_k$ maps test stories to user stories and becomes a transfer function, if each cell contains the size $\lfloor (a_i \to b)_j \rfloor$ of all test cases $(a_i \to b)_j$ belonging to some test story T_k and referring to some user story, or FUR U_l. This yields a matrix:

$$\boldsymbol{A} = \left(\lfloor (a_i \to b)_j \rfloor\right)_{l,k} \tag{16}$$

The indices of the matrix run over integers $l, k \in \mathbb{N}$.

The transfer function $\boldsymbol{A}$ maps test stories to user stories, and we call it *Test Coverage Matrix* because you can assess how good test stories cover user stories.

Let user stories be prioritized, say by some *Goal Profile* $\boldsymbol{y}$. The goal profile characterizes priorities by a unit vector in the space of the alternatives under consideration. Then the transfer function $\boldsymbol{A}$ can be applied to a *Solution Profile* $\boldsymbol{x}$, describing the importance of the test stories, and $\boldsymbol{Ax}$ is the result of applying $\boldsymbol{A}$ to this solution profile. Obviously $\boldsymbol{Ax} \neq \boldsymbol{y}$; however, the difference $\|\boldsymbol{x} - \boldsymbol{y}\|$ is interesting. If this difference is small, then the solution profile $\boldsymbol{x}$ represents an optimum selection of test stories, meaning that tests cover what is relevant to the user's goal profile.

Optimum solution profiles can be calculated using the eigenvector method (Fehlmann 2016, p. 34). Let $\boldsymbol{y_A}$ be the *Principal Eigenvector* of $\boldsymbol{AA}^{\mathsf{T}}$, solving the eigenvalue problem (17) for some $\lambda \in \mathbb{R}$.

$$\boldsymbol{AA}^{\mathsf{T}}\boldsymbol{y_A} = \lambda \boldsymbol{y_A} \tag{17}$$

The principal eigenvector $\boldsymbol{y_A}$ is called the *Achieved Profile* of the transfer function $\boldsymbol{A}$. Both, $\boldsymbol{y}$ and $\boldsymbol{y_A}$ are *Profiles*. This means, their vector length $\|\boldsymbol{y}\| = 1$ respectively $\|\boldsymbol{y_A}\| = 1$ are both one, where $\|...\|$ represents the *Euclidean Norm* for vectors. The difference between a goal profile and an achieved profile is called *Convergence Gap*:

$$Convergence\ Gap = \|\boldsymbol{y} - \boldsymbol{y_A}\| \tag{18}$$

The convergence gap is a metric that measures how well a transfer function explains the observed profile with suitable controls. The controls are the test stories; the observed profile compares with the goal profile of the user stories' relevance for the user of the software or the system. Note that computing the achieved profile is very often not straightforward, as it is in our case where we can make use of simple linear algebra.

We can now construct attractors as a series $\boldsymbol{A_0}, \boldsymbol{A_1}, \boldsymbol{A_2}, ...$ of test coverage matrices that approximate the test suite that we need to cover our functional

requirements. However, the attractors must all keep the convergence gap und control, meaning that for a certain $\varepsilon > 0$ and all attractors $\boldsymbol{A}_i$ holds:

$$Convergence\ Gap(\boldsymbol{A}_i) = \left\|\boldsymbol{y}_i - \boldsymbol{y}_{\boldsymbol{A}_i}\right\| < \varepsilon \tag{19}$$

Thus, our constructor $\mathbf{C}$ must construct an ascendant series $\boldsymbol{A_0}, \boldsymbol{A_1}, \boldsymbol{A_2}, \ldots$ such that both (19) and (20) holds:

$$\begin{gathered} \boldsymbol{A}_{i+1} = \mathbf{C} \bullet \boldsymbol{A}_i, i \in \mathbb{N} \\ \boldsymbol{A_i} \subseteq \boldsymbol{A}_{i+1}, i \in \mathbb{N} \end{gathered} \tag{20}$$

The constructor $\mathbf{C}$ therefore is an intelligent search in a wide range of potential attractors, keeping the convergence gap small enough. Such a series of attractors is called *bound*, namely by the convergence gap. In fact, bound attractors constitute a formal way to solve all kind of issues normally tackled by Artificial Intelligence. The hash functions used for measuring the convergence gap, might be considerably more complicated than in the case of test size.

Optimum Test Size

For a test coverage matrix $\boldsymbol{A} = \left(\left\lfloor (a_i \to b)_j \right\rfloor\right)_{l,k}$, the total test size of $\boldsymbol{A}$ is

$$\lfloor \boldsymbol{A} \rfloor = \sum_{l,k} \left(\left\lfloor (a_i \to b)_j \right\rfloor\right)_{l,k}\ l, k \in \mathbb{N} \tag{21}$$

If $\boldsymbol{A}_0 \subseteq \boldsymbol{A}_1 \subseteq \boldsymbol{A}_2 \subseteq \cdots$, then $\lfloor \boldsymbol{A_0} \rfloor \leq \lfloor \boldsymbol{A_1} \rfloor \leq \lfloor \boldsymbol{A_2} \rfloor \leq \cdots$ also holds.

Combinations of tests can be used, as well as applying any special combinator such as projection or substitution to generate new test cases.

Bound attractors build up a test suite by adding more tests to the test coverage matrix $\boldsymbol{A}$. The convergence gap must not necessarily decrease. In contrary, adding more tests can spoil the convergence gap, for instance if some test story gains too much weight and inflate the respective user stories' achieved profile.

Constructing a suitable constructor $\mathbf{C}$ is all but simple, nor straightforward, because adding more tests does not solve a problem. In view of executing such tests on a machine, the number of tests must not only remain finite but also limited to some manageable number. Thus, the question how to select relevant test stories out of the many possible combinations must be answered. An answer is proposed in Fehlmann and Kranich (2019). Moreover, based on sensitivity analysis for linear matrices, the authors will present sample attractors for software and system testing at the upcoming (2022) ATINER's conference on Information Technology & Computer Science (Fehlmann and Kranich 2022). Sensitivity analysis speeds up the selection of the new test cases. Thus, it seems that the problem of effectively create autonomous tests for large and complex systems can be solved.

For practical applications, combining unit tests from related domains such as steering control of an autonomous vehicle with weather forecast is always feasible

to construct attractors for a system test of this autonomous vehicle. The convergence gap enforces that such test combinations cover the full range of test cases relating to both steering control, and process weather forecast services; otherwise, some test stories would grow beyond limits.

There is an optimum number of attractors delivering enough tests to fit the test intensity required by the user of the system, and test size that still can be executed within a limited time frame. Computing that optimum is an important task for product management, and, depending upon safety and privacy criticality, must be carefully chosen to make such a product acceptable for the society.

Conclusions

It has been shown that Aristotle's *Mental Completion* leads to feasible solutions for actual challenges and problems. His understanding of recursion as a mentally completed inductive definition of a concept (Engeler 2020) allows developing the techniques necessary for testing modern, complex systems of systems, including cyber-physical systems that impact people's life and health. Following Engeler, we identified constructors as a general prescription for constructing attractors that serve as approximations to solutions for problems. Effective methods and algorithms exist for such constructions in the algebra of tests, as shown in Fehlmann and Kranich (2020), and in Fehlmann (2020). Linking attractors to fixpoint operators, in a very practical setting, has potentially a high economic impact in the *Fourth Industrial Revolution* (Schwaab 2017), in the realm of cyber-physicals systems such as autonomous cars, intelligent medical instruments, virtual reality, and more.

Open Questions

Why did Aristotle not invent relations? Because he had no use for them (Engeler 2020, p. 12). Euclidean geometry went without relations. So why do we not yet know combinators for software testing? Because we are probably just now finding out what they could be good for?

The authors are currently developing ideas how actual constructors look for software and system testing, as well as for Artificial Intelligence. However, whether there are some general rules to follow, besides Combinatory Logic, or if every testing domain requires its own constructors and attractor series, remains open.

Obviously, there are more open questions than we can mention here. Maybe this is a step toward the *New Kind of Science* that Stephen Wolfram (2002) promised us in the early years of this century? Is the approach presented in this paper potentially fruitful not only to Artificial Intelligence, Neuroscience, and system testing? What else could we describe by a constructor and by attractors? Thus, better understanding what we are doing, and why?

Why are we doing theoretical stuff like logic and other basic sciences? Maybe the answer is because this is the way to new business models and more efficient progress in applied sciences? Probably the only sure way? Because otherwise you get lost in the jungle? Without hope for finding an exit.

Do we offer young engineers enough education in basic sciences? Once they have mastered that, they can apply the basic findings to any applied technical or scientific area they care for.

Acknowledgments

Thanks to Erwin Engeler who sent his former student the enjoyable paper about Aristotle' Relations that he wrote as a gift to his 90th birthday and for all the investigations into the Graph Model that he did. I hope we were able to share this with our reviewers and readers.

Also, many thanks to all who contributed to this paper by pointing to weaknesses and confusions. Special thanks to the reviewers who contributed with their comments much for improving this paper.

References

Akao Y (1990) *Quality function deployment - Integrating customer requirements into product design*. Portland, OR: Productivity Press.

Aristoteles (367-344 BCE) *Organon*. Übersetzt von Julius von Kirschmann, Hofenberg ed. Berlin: Andronikos von Rhodos.

Barendregt HP (1977) The type-free lambda-calculus. In J Barwise (ed.), *Handbook of Mathematical Logic*, 1091–1132. Amsterdam: North Holland.

Barendregt HP (1984) The lambda calculus – Its syntax and semantics. In *Studies in Logic and the Foundations of Mathematics*. Amsterdam: North-Holland.

Bimbó K (2012) *Combinatory logic - Pure, applied and typed*. Boca Raton, FL: CRC Press.

Church A (1941) The calculi of lambda conversion. In *Annals of Mathematical Studies* no. 6. Princeton University Press.

Curry H (1930) Grundlagen der kombinatorischen Logik. (Basics of combinatory logic). *American Journal of Mathematics* 52(3): 509–536.

Curry H, Feys R (1958) *Combinatory logic*, volume I. Amsterdam: North-Holland.

Curry H, Hindley J, Seldin J (1972) *Combinatory logic*, volume II. Amsterdam: North-Holland.

Deming W (1986) *Out of the Crisis*. Center for Advanced Engineering Study ed. Boston, MA: Massachusetts Institute of Technology.

Engeler E (1981) Algebras and Combinators. *Algebra Universalis* 13(Dec): 389–392.

Engeler E (1995) *The combinatory programme*. Basel, Switzerland: Birkhäuser.

Engeler E (2019) Neural algebra on "how does the brain think?" *Theoretical Computer Science* 777(Apr): 296–307.

Engeler E (2020) *Aristotle' relations: an interpretation in combinatory logic*. arXiv: History and Overview.

Fehlmann TM (1981) *Theorie und Anwendung des Graphmodells der Kombinatorischen Logik*. (Theory and application of the graph model of combinatory logic). Zürich, CH: ETH Dissertation 3140-01.

Fehlmann TM (2001) QFD as algebra of combinators. In *8th International QFD Symposium, ISQFD 2001*. Tokyo, Japan.

Fehlmann TM (2016) *Managing complexity – Uncover the mysteries with six sigma transfer functions*. Berlin, Germany: Logos Press.

Fehlmann TM (2020) *Autonomous real-time testing – Testing artificial intelligence and other complex systems*. Berlin, Germany: Logos Press.

Fehlmann TM, Kranich E (2014) *Exponentially Weighted Moving Average (EWMA) prediction in the software development process*. Rotterdam, NL: IWSM Mensura.

Fehlmann TM, Kranich E (2019) Testing artificial intelligence by customers' needs. *Athens Journal of Sciences* 6(4): 265–286.

Fehlmann TM, Kranich E (2020) Intuitionism and computer science – Why computer scientists do not like the axiom of choice. *Athens Journal of Sciences* 7(3): 143–158.

Fehlmann TM, Kranich E (2022) A sensitivity analysis procedure for matrix-based transfer functions. *Athens Journal of Sciences* (proposed).

ISO 16355-1 (2015) *Applications of statistical and related methods to new technology and product development process - part 1: general principles and perspectives of Quality Function Deployment (QFD)*. Geneva, Switzerland: ISO TC 69/SC 8/WG 2 N 14.

ISO/IEC 19761 (2011) *Software engineering – COSMIC: a functional size measurement method*. Geneva, Switzerland: ISO/IEC JTC 1/SC 7.

Ishikawa K, Loftus JH (1990) *Introduction to quality control*. Tokyo: 3A Corporation.

JUnit Team (1997ff) The 5th major version of the programmer-friendly testing framework for Java and the JVM. Retrieved from: https://junit.org/. [Accessed 28 January 2022]

Potter MD (2004) *Set theory and its philosophy*. Oxford, UK: Oxford University Press.

Schönfinkel M (1924) Über die Bausteine der mathematischen Logik. (About the building blocks of mathematical logic). *Mathematische Annalen* 92(3–4): 305–316.

Schwaab K (2017) *The fourth industrial revolution*. First Edition. New York: World Economic Forum.

Turing A (1937) On computable numbers, with an application to the Entscheidungs problem. In *Proceedings of the London Mathematical Society* 42(Series 2): 230–265.

Wolfram S (2002) *A new kind of science*. First Edition. Champaign, Illinois: Wolfram Media.

Zachos E (1978) *Kombinatorische Logik und S-Terme*. (Combinatorial logic and S-terms). Zurich: ETH Dissertation 6214.