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

Combinatory Logic is an elegant and powerful logical theory that is used in computer science as a theoretical model of 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 Moses Schönfinkel (Schönfinkel, 1924) and Haskell Curry (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).

Introduction

A model of Combinatory Logic is an algebraic structure implementing combinators in a non-trivial way. Such a model is called Combinatory Alge-
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. We refer to these general terms as Combinatory Terms (Bimbó, 2012, p. 2).

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 \( Y \cdot X \) with the property that \( Y \cdot X = X \cdot (Y \cdot X) \), for any term \( X \) of Combinatory Logic, is thus something like a solution to the problem. You can apply that solution as many times as deemed necessary and the problem remains stable and confined.

When we encounter the problem of how to test a piece of software \( X \), and we have a test suite \( Y \cdot X \) 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 \( g = f \cdot x \) on a variable \( y \). Thus, \( f(x, y) = g(y) = (f \cdot x) \cdot y = f \cdot x \cdot y \); 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 & Feys, 1958) and (Curry, et al., 1972). Such algebras are closed under a combination operation \( M \cdot N \) for all terms of the algebra \( M, N \); and two distinct Combinators \( S \) and \( K \) can be defined with the following properties:

\[
K \cdot P \cdot Q = P \tag{1}
\]

and

\[
S \cdot P \cdot Q \cdot R = P \cdot Q \cdot (P \cdot R) \tag{2}
\]

where \( P, Q, R \) are elements in the combinatory algebra\(^1\).

\(^1\)The use of variables named \( P, Q, R \) is borrowed from Engeler (Engeler, 2020).
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). We present here only a few of them.

Identity

The identity combinator is defined as

\[ I := S \cdot K \cdot K \]  

Indeed, \( I \cdot M = S \cdot K \cdot K \cdot M = K \cdot M \cdot (K \cdot M) = M \). 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 \( L_x \) for a variable \( x \) as follows:

\[ L_x \cdot x = I \]  
\[ L_x \cdot M = K \cdot M \text{ if } M \text{ different from } x \]  
\[ L_x \cdot M \cdot N = S \cdot L_x \cdot M \cdot (L_x \cdot N) \]

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 \( L_x \) 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 \( (\lambda x. M) \cdot N \).

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

The Fixpoint Combinator

Given any combinatory term \( Z \), the Fixpoint Combinator \( Y \) generates a combinatory term \( Y \cdot Z \), called Fixpoint of \( Z \), that fulfills \( Y \cdot Z = Z \cdot (Y \cdot Z) \).

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 \( Y \cdot Z \) is a solution for the problem \( Z \). For more details, consult Fehlmann (Fehlmann, 2016).

Using Lambda Calculus notation, the fixpoint combinator can be written as

\[
Y := \lambda f. (\lambda x. f \bullet (x \bullet x)) \bullet (\lambda x. f \bullet (x \bullet x))
\]  

(7)

For a reference see Barendregt (Barendregt, 1984),

Translating (7) into an S-K term proves possible but becomes a bit lengthy.

By applying (6), (5):

\[
\lambda x. f \bullet (x \bullet x) = S \circ \lambda x. f \circ \lambda x. x \bullet x
\]

\[
\lambda x. f = K \circ f
\]

Then applying (6) and (4):

\[
\lambda x. x \bullet x = (S \circ \lambda x. x \circ \lambda x. x)
\]

\[
= (S \circ \lambda x. x)
\]

this yields

\[
\lambda x. f \bullet (x \bullet x) = S \circ (K \circ f) \circ (S \circ \lambda x)
\]

and therefore

\[
Y = \lambda f. (\lambda x. f \bullet (x \bullet x)) \bullet (\lambda x. f \bullet (x \bullet x))
\]

\[
= \lambda f. (S \circ (K \circ f) \circ (S \circ \lambda x)) \bullet (S \circ (K \circ f) \circ (S \circ \lambda x))
\]

\[
= S \circ (S \circ (\lambda f. S \circ (K \circ f))) \circ (\lambda f. S \circ \lambda x) \circ (S \circ (\lambda f. S \circ (K \circ f))) \circ (\lambda f. S \circ \lambda x)
\]

Now solving the remaining \( \lambda \)-terms:

\[
\lambda f. S \circ (K \circ f) = S \circ \lambda f. S \circ \lambda f. K \circ f
\]

\[
= S \circ (K \circ S) \circ (S \circ \lambda f. K \circ \lambda f. f)
\]

\[
= S \circ (K \circ S) \circ (S \circ (K \circ K) \circ \lambda f)
\]

considering

\[
\lambda f. S = K \circ S
\]

and

\[
\lambda f. K \circ f = S \circ \lambda f. K \circ \lambda f. f = S \circ (K \circ K) \circ \lambda f.
\]

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

\[
\lambda f. S \circ \lambda f. K \circ \lambda f. f = S \circ (K \circ K) \circ \lambda f.
\]

applying again (6) and (5).

Putting things together:

\[
Y = \lambda f. (\lambda x. f \bullet (x \bullet x)) \bullet (\lambda x. f \bullet (x \bullet x))
\]

\[
= S \circ (S \circ (K \circ S) \circ (S \circ (K \circ K) \circ \lambda f) \circ (K \circ f) \circ (S \circ (K \circ S) \circ (K \circ f) \circ (S \circ (K \circ K) \circ \lambda f)))
\]
Applying $\text{Y}$ to any combinator term $Z$ now explicitly transports $Z$ on the top of the formula and keeps the rest of the structure of $\text{Y}$ such that $\text{Y}$ can be applied repeatedly. This exercise should give the reader an impression how Combinatory Logic works.

Applying the fixpoint combinator $\text{Y}$ to some combinator $Z$ in the Lambda-style is much simpler:

$$
\lambda f. (\lambda x. f \bullet (x \bullet x)) \bullet ((\lambda x. f \bullet (x \bullet x)) \bullet Z) = (\lambda x. Z \bullet (x \bullet x)) \bullet ((\lambda x. Z \bullet (x \bullet x)) \bullet Z)
$$

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.

There are simpler forms of the Fixpoint Combinator, e.g.,

$$
\text{Y}' = S \bullet (K \bullet (S \bullet I \bullet I)) \bullet (S \bullet (S \bullet (K \bullet S) \bullet K) \bullet (K \bullet (S \bullet I \bullet I)))
$$

Proof is left to the reader.

When applying $\text{Y}$, or $\text{Y}'$, or any other equivalent fixpoint combinator to a combinator 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 (Potter, 2004). For more combinators in Combinatory Logic, see e.g., Zachos (Zachos, 1978) and Bimbò (Bimbò, 2012).

**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 $a_1, ..., a_m, b$ be arrow terms. Then

$$\{a_1, ..., a_m\} \rightarrow b$$  \hspace{1cm} (9)

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 $a_1, ..., a_m$ represent test data, the term $b$ is the known expected response of the test case (9). 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 (9). The formal, recursive, definition of the Graph Model as a power set, in set-theoretical language, is given in equation (10):

$$\mathcal{G}_0(\mathcal{L}) = \mathcal{L}$$
$$\mathcal{G}_{n+1}(\mathcal{L}) = \mathcal{G}_n(\mathcal{L}) \cup \{\{a_1, ..., a_m\} \rightarrow b | a_1, ..., a_m, b \in \mathcal{G}_n(\mathcal{L}), m \in \mathbb{N}\}$$  \hspace{1cm} (10)

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

$$\mathcal{G}(\mathcal{L}) = \bigcup_{n \in \mathbb{N}} \mathcal{G}_n(\mathcal{L})$$  \hspace{1cm} (11)

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:

\[ M \cdot N = \{ c | \exists \{b_1, b_2, ..., b_m\} \rightarrow c \in M; b_i \in N \} \]  

(12)

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.
- \( a_i \) for a singleton set of arrow terms: \( a_i = \{a\} \) for an arrow term \( a \).
- \( \emptyset \) for the empty set, such as in the arrow term \( \emptyset \rightarrow a \).
- \( a_i \cup b_j \) 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

\[ a_{i,j} = a_{i,1} \cup a_{i,2} \cup ... \cup a_{i,j} \cup ... \cup a_{i,m} = \bigcup_{k=1}^{m} a_{i,k} \]  

(13)

In terms of these conventions, \( (x_i \rightarrow y)_j \) 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 \( x_i \rightarrow y \).

With this notation, the application rule for \( M \) and \( N \) now reads:

\[ M \cdot N = \{(b_i \rightarrow a) \cdot (b_j) \} = \{a | \exists b_i \rightarrow a \in M; b_i \subset N \} \]  

(14)

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).

- \( I = (a_1 \rightarrow a) \) is the Identification; i.e., \( (a_1 \rightarrow a) \cdot (b) = (b) \)
\[ K = (a_1 \rightarrow \emptyset \rightarrow a) \] selects the 1st argument:
\[ K \bullet (b) \bullet (c) = ((b_1 \rightarrow \emptyset \rightarrow b) \bullet (b)) \bullet (c) = (\emptyset \rightarrow b) \bullet (c) = (b) \]
\[ KI = (\emptyset \rightarrow a_1 \rightarrow a) \] selects the 2nd argument:
\[ KI \bullet (b) \bullet (c) = ((\emptyset \rightarrow c_1 \rightarrow c) \bullet (b)) \bullet (c) = (c_1 \rightarrow c) \bullet (c) = (c) \]
\[ S = \left( (a_i \rightarrow (b_j \rightarrow c)) \rightarrow (d_k \rightarrow b) \rightarrow (a_i \cup b_{j,i} \rightarrow c) \right) \]

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

The proof that the arrow terms’ definition of \( S \) fulfills equation (2) is somewhat more complex. Readers interested in that proof are referred to Engeler (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 combinator (Barendregt, 1977). See also in Fehlmann (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 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 \( a_1 \rightarrow b \) or \( \emptyset \rightarrow b \). This reduces the model to become the trivial one.

As an example, see Bimbó (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 last years’ paper of the authors (Fehlmann & Kranich, May 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 his-
Its elements may be visualized as a directed graph, whose nodes indicate the firing of a neuron. As before, we consider \( G(\mathcal{A}) \), constructed as in (11). The elements of \( G(\mathcal{A}) \) 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 (14), we have an algebraic structure; the application representing brain functions, interpreted as thoughts.

**The Fixpoint Combinator in the Neural Algebra**

The fixpoint combinator \( \mathbf{Y} \) 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 \( \mathbf{Y} \) 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 \( \mathbf{C} \cdot X = X \). 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

\[
X_{i+1} = \mathbf{C} \cdot X_i, i \in \mathbb{N}
\]  

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

Attractors are ordered by inclusion (15), 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. Eventually, this ultimate solution is empty.

The controlling operator is closely linked to the fixpoint operator $Y$; however, if $X$ has a solution $C \cdot X$, then $X$ is of the form $X = Y \cdot Z$ for some suitable cascade $Z$. Thus, not all combinator $X$ 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 alike one of the basic or the fixpoint combinators but is more of a prescription, how to find suitable attractors. Engeler (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, 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. 321). 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 digi-
talization and to software-intensive 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 last
years’ paper (Fehlmann & Kranich, May 2020). The data movements induce
a sizing valuation on this algebra by counting the number of data move-
ments executed once per test case.

When we speak of test cases, we always intend a suitable data move-
ment 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 asser-
tions 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. Elements of $\mathcal{L}$ are descriptions of the system status at a cer-
tain moment. In the sequel, the arrow term $a_i \rightarrow b$ together with its associat-
ed data movement map represents a test case, that, given test data $a_i$,
yields $b$ as the expected, correct result.

Engines providing machine learning and artificial intelligence are in-
cluded in such a “program” (Fehlmann, Jan. 2020, p. 85ff). If $a_i \rightarrow b$ is a
test case, $a_i \subseteq \mathcal{L}$ 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.

Combining Tests

The definition (14) looks somewhat counter-intuitive for combining
tests. To apply one test case to another, it is required that the result of appli-
cation contains all the full test cases providing the response sought.

A more intuitive approach would only require test cases providing such
a response meeting the required controls. The existential quantifier would
then guarantee that there is a test case yielding such response. When accept-
ing the Axiom of Choice in its traditional form, that does not look like a
problem. However, this seemingly more intuitive approach would immedi-
ately lead to a contradiction to Turing’s halting problem (Turing, 1937).
Consult last years’ paper (Fehlmann & Kranich, May 2020) for more infor-
mation of this observation.

The intuitionistic, or constructive, variant of the Axiom of Choice re-
quires 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 $M \cdot N$ 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 intuitionism and programming.

**Combination Limitations**

Combining tests in a Combinatory Algebra is unlimited indeed because there is no typing involved that governs applicability. By (14), you can combine test cases across test stories as deemed appropriate; all that counts are the data movements. The only restriction is, not all combinations of test cases yield an executable new test case. It must be possible to combine the data movement maps associated to the test cases as well; thus, if these movements do not overlap between two test cases, they do not yield a valid and executable test case.

We therefore silently agree that combinations of tests only are considered if their data movement maps overlap somehow.

**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 $Cfp(a^0_i \rightarrow b^0)$, where $a^0_i \in G_0(\mathcal{L})$ and $b^0 \in G_0(\mathcal{L})$ are arrow terms of level 0; i.e., assertions about the state of the program. $Cfp(a^0_i \rightarrow b^0)$ is the number of unique data movements touched when executing the test case $a^0_i \rightarrow b^0$. This is the recursion base.

Then the following equations (16) recursively define the size of tests:

$$
|a| = 0 \text{ for } a \in G_0(\mathcal{L})
$$

$$
|a_i \rightarrow b| = Cfp(a_i \rightarrow b) \text{ for } a_i \in G_0(\mathcal{L}) \text{ and } b \in G_0(\mathcal{L})
$$

$$
|c_i \rightarrow d| = \sum_i |c_i| + |d| \text{ for all test cases } c_i \text{ and } d
$$

(16)

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 (16) to the combinators $S, K, I$, and $Y$ 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 our recent book (Fehlmann, Jan. 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 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. For an example, consult last year’s paper (Fehlmann & Kranich, May 2020).

Attractors

While the fixpoint combinator $Y$ 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 quite like 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 & Kranich, 2014) – a more theoretical answer should include at least the requirement that attractors cover functionality. This leads to the notion of Convergence Gap, explained in (Fehlmann, Jan. 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 $\lfloor (a_i \rightarrow b)_j \rfloor$ of all test cases $(a_i \rightarrow b)_j$ belonging to some test story $T_k$ and referring to some user story, or FUR $U_l$. This yields a matrix:

$$A = \lfloor (a_i \rightarrow b)_j \rfloor_{l,k}$$ (17)
The indices of the matrix run over integers \( l, k \in \mathbb{N} \).

The transfer function \( A \) maps test stories to user stories, and we call it the Test Coverage Matrix because you can assess how good test stories cover user stories.

Let user stories be prioritized, say by some Goal Profile \( y \). The goal profile characterizes priorities by a unit vector in the space of the alternatives under consideration. Then the transfer function \( A \) can be applied to a Solution Profile \( x \), describing the importance of the test stories, and \( Ax \) is the result of applying \( A \) to this solution profile. Obviously \( Ax \neq y \); however, the difference \( \| x - y \| \) is interesting. If this difference is small, then the solution profile \( x \) 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 \( y_A \) be the Principal Eigenvector of \( AA^\top \), solving the eigenvalue problem (18) for some \( \lambda \in \mathbb{R} \).

\[
AA^\top y_A = \lambda y_A
\] (18)

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

\[
Convergence\ Gap = \| y - y_A \| \quad (19)
\]

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 \( A_0, A_1, A_2, \ldots \) 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 and control, meaning that for a certain \( \varepsilon > 0 \) and all attractors \( A_i \) holds:

\[
Convergence\ Gap(A_i) = \| y_i - y_A \| < \varepsilon \quad (20)
\]

Thus, our constructor \( C \) must construct an ascendant series \( A_0, A_1, A_2, \ldots \) such that both (20) and (21) holds:

\[
A_{i+1} = C \cdot A_i, \ i \in \mathbb{N}
\]

\[
A_i \subseteq A_{i+1}, \ i \in \mathbb{N} \quad (21)
\]

The constructor \( C \) therefore is an intelligence search in a wide range of potential attractors, keeping the convergence gap small enough.
We call such a series of attractors *Bound*, 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 $A = \left(\left\lfloor (a_i \rightarrow b_j) \right\rfloor\right)_{l,k}$, the total test size of $A$ is

$$|A| = \sum_{l,k} \left\lfloor (a_i \rightarrow b_j) \right\rfloor_{l,k}, l,k \in \mathbb{N} \tag{22}$$

If $A_0 \subseteq A_1 \subseteq A_2 \subseteq \ldots$, then $|A_0| \leq |A_1| \leq |A_2| \leq \ldots$ also holds.

We can use combination of tests, 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 $A$. 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.

Therefore, constructing a suitable constructor $C$ 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.

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. However, 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**

Aristotle's *Mental Completion* reflects his understanding of recursion as a mentally completed inductive definition of a concept (Engeler, 2020). Following him, we identified constructors as a general prescription for constructing attractors that serve as approximations to solutions for problems. We have shown how such constructions look in the case of the Algebra of
Tests, although not in this paper but in its predecessors (Fehlmann & Kranich, May 2020), and in our recent book (Fehlmann, Jan. 2020). We linked attractors to fixpoint operators, and all this in a very practical setting, with potentially high economic impact.

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.

So, why don’t we educate our young engineers in basic sciences? Once they have mastered that, they can apply the basic findings to any applied technical or scientific area they care for.

Open Questions

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 promised us in the early years of this century (Wolfram, 2002)? 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?

Less philosophical is the open and quite practical question, how actual constructors look for software and system testing, whether there are some general rules to follow, besides Combinatory Logic, or if every testing domain requires its own constructors and attractor series.

There is also a mathematical question to solve, namely how we can speed up the selection of the right cells in the test coverage matrix that needs enhancements by new test cases. This is related to sensitivity analysis (Fehlmann & Kranich, 2020) for QFD matrices. If a practical solution exists, this could help us constructing straight attractors instead of searching around.

However, such a solution is not yet known to the authors.

Bibliography


