How Combinatory Logic Can Limit Computing Complexity

Hugolin Bergier[1,*]

[1]Regis University

Abstract. As computing capabilities are extending, the amount of source code to manage is inevitably becoming larger and more complex. No matter how hard we try, the bewildering complexity of the source code always ends up overwhelming its own creator, to the point of giving the appearance of chaos. As a solution to the cognitive complexity of source code, we are proposing to use the framework of Combinatory Logic to construct complex computational concepts that will provide a model of description of the code that is easy and intuitive to grasp. Combinatory Logic is already known as a model of computation but what we are proposing here is to use a logic of combinators and operators to reverse engineer more and more complex computational concept up from the source code. Through the two key notions of computational concept and abstract operator, we will show that this model offers a new, meaningful and simple way of expressing what the intricate code is about.

1 Introduction

In Sept. 26, 2017, an article entitled ”The Coming Software Apocalypse” [1] exposed the dangers we’re facing as software implementations are becoming increasingly complex: ”Software has enabled us to make the most intricate machines that have ever existed. And yet we have hardly noticed, because all of that complexity is packed into tiny silicon chips as millions and millions of lines of code. But just because we can’t see the complexity doesn’t mean that it has gone away (...) Just by editing the text in a file somewhere, the same hunk of silicon can become an autopilot or an inventory-control system. This flexibility is software’s miracle, and its curse. Because it can be changed cheaply, software is constantly changed; and because it’s unmoored from anything physical it tends to grow without bound. The problem is that we are attempting to build systems that are beyond our ability to intellectually manage."

As computing capabilities are extending, the amount of source code to manage is inevitably becoming larger and more complex. No matter how hard we try, the bewildering complexity of the source code always ends up overwhelming its own creator, to the point of giving the appearance of chaos. Most applications source code has become intrinsically complex for a human being to read, navigate and comprehend.

One needs to step back from the source code in order to look at the big picture. What happens if a developer tries to step back from the code? Unlike Google Maps, zooming in and out on a program source code will radically change the cognitive complexity of the content. The big picture contains simply more data and intricacy than the small picture.

The point of this paper is to propose a solution to the overwhelming cognitive complexity of source code in large applications. By *cognitive complexity*, we mean a level of intricacy in the code that makes it too difficult to grasp for a human mind.

We are proposing to use the framework of Combinatory Logic to construct complex computational concepts as a model of description of the source code that is easy and intuitive to grasp. Combinatory Logic is already known as a model of computation but what we are proposing here is to use a logic of combinators and operators to reverse engineer more and more complex computational concept up from the source code. This will lead to what we can call a vertical simplification of the computational concepts: through progressive layers of complexity, we can define high-level concepts from other (less complex) concepts without having to use the most primitive computational concepts, like incrementation or conditionals. To the vertical simplification, we will add an horizontal simplification: given that a concept in Combinatory Logic is essentially the application of an operator to an operand, we can use the combinators to extract a meaningful abstract computational operator that is simpler to grasp than the complex concept it composes.

2 Applicative Languages and Combinatory Logic

Combinatory Logic is an applicative language. While the primitive notions of application, operator and operand belong to applicative languages in general, the notion of combinator and β-reduction are specific to Combinatory Logic.

Both the notions that belong to applicative languages in general and to Combinatory Logic in particular play an important role in what follows.

*e-mail: hbergier@regis.edu

2.1 Applicative Languages

Applicative languages rely on the fundamental operation of applying an operator to an operand in order to dynamically produce a result. What distinguishes applicative languages from set-theoretical languages is that the application is considered as primitive notion and not as a notion derivable from that of set. Applicative languages study operational processes : their construction, functioning and nesting. By choosing to apprehend the application of an operator to an operand as a primitive notion, we place ourselves in a logical framework different from that which is used in most modern logic systems. There are a variety of applicative languages: the lambda-calculus of Church [2]; the combinatorial logic of Curry [3, 4] and F. Fitch [5]; the attempts of WVO Quine [6]. In Computer Science, applicative languages correspond to a whole family of programming languages called functional (among which Lisp, Haskell and ML). After having presented the foundations of applicative systems, we will focus on Combinatory Logic.

As we have just said, the primitive notion of an applicative system is a binary operation, called application. The first argument of the application is called an operator, the second argument is called an operand. The value of this operation is called result or outcome. In an application operation, an operator f is applied to its operand a to produce the result b. We can simply represent the application as follows:

$$fa$$

We call this expression an applicative expression that we can read “operator f applies to operand a”. When the operation of application is executed, a result b appears. Between the application expression and the result, we introduce a relation, designated by $>_{eval}$; we then have:

$$fa >_{eval} b$$

which reads “the operator f applies to the operand a to give the result b. "The process of getting from operation to result is called evaluation ; it is expressed by the relation '$>_{eval}$'. The evaluation is a non-symmetrical and dynamic relation:

- Non-symmetrical because it is true that f and a determine b but not the other way around.
- Dynamic since there are two successive "moments": fa does not coexist with its result b but precedes it.

Note that these two properties of the evaluation – that the set-theoretic notion of function doesn’t have – are tightly related to the properties of code execution in computer science. The execution of a program is nothing else that the evaluation of a formal expression. In that respect, the applicative principle fits very well the computing framework.

Example 1. The application “squared” can be written:

$$squared5 >_{eval} (5)2 >_{eval} 25$$

In set theory, we have:

$$squared(5) = (5)2 = 25$$

Unlike '$>_{eval}$' the relator '$=$' is both symmetrical and static. Indeed, in set theory, the function is given as a set of ordered pairs < x, y > and any image y is coexisting with the corresponding argument x.

The result of an application operation is not necessarily an operand as in the previous example. It can also be a new operator. This fundamental property of the application operation is called *repeatability*. An application operation can produce a new operator which is in turn applicable to an operand. For example, the operation squared applied to plus-three produces the new plus-three-all-squared operation. Repeatability is a very important characteristic of applicative systems in our context because, as we mentioned above, we want to be able to understand complex computational concepts as related to other computational concepts, not directly to the actual implementation.

2.2 Combinatory Logic

2.2.1 Combinators and reduction

Combinators are abstract operators that combine the predefined operators of an application system so as to form new, more complex, operators. As operators, they are applicable to their operands. The evaluation of a combinator application is called β-*reduction*. This evaluation defines the meaning of the combinator, regardless of any external interpretation. The meaning of a combinator is going to depend on the way it combines its operands through β-reduction: permutation, duplication, composition, deletion,... Thus, combinators are formal tools endowed with an intrinsic semantics [**?**]. In other words, the semantics of combinators is not defined by an external interpretation in a model. Combinators are therefore very specific operators in that they are completely reducible to their combinatorial action (that which is performed on their operands). One could say that they belong to an additional level of abstraction compared to the operators since their domain of interpretation is not given. Let us now introduce the most common elementary combinators.

2.2.2 Elementary combinators

Combinatory Logic fits particularly well the framework of Natural Deduction [7] which defines the syntactic rules of formation of the language in same times as its formal semantics through a series of rules of introduction and rules of elimination of the logical operators.

Let us take the example of the combinator of composition **B**:

Rule of elimination [**B**-e]:

1	$\mathbf{B}xyz$	
2	$x(yz)$	[**B**-e]

This deduction corresponds to the β-reduction [3]:

$$\mathbf{B}xyz \geq_{\beta} x(yz)$$

Semantically, B can be interpreted as an operator (combinator) that takes to functions (x and y in this case) and compose them together to be applied to the third operand (z). The role played by B becomes more obvious if we take usual function names for the first two operands and if use the function notation:

$$\mathbf{B}fgx \geq_\beta f(g(x))$$

If we abstract from x and use the usual notation for composition '$\circ$', we have[1]

$$\mathbf{B}fg \geq_\beta [f \circ g]$$

Rule of introduction [**B**-i]:

1	$x(yz)$	
2	$\mathbf{B}xyz$	[**B**-i]

Combinator of duplication W:

Rule of elimination [**W**-e]:

1	$\mathbf{W}xy$	
2	xyy	[**W**-e]

Rule of introduction [**W**-i]:

1	xyy	
2	$\mathbf{W}xy$	[**W**-i]

Given a rule of elimination of a combinator, the rule of introduction is straightforward. So we will only give the latter for the following combinators.

Combinator of identity I:

1	$\mathbf{I}x$	
2	x	[**I**-e]

Combinator of deletion K:

1	$\mathbf{K}xy$	
2	y	[**K**-e]

Combinators of distribution:

1	$\Phi xyzu$	
2	$x(yu)(zu)$	[Φ-e]

1	$\Psi xyzu$	
2	$x(yz)(yu)$	[Ψ-e]

1	$\mathbf{S}xyz$	
2	$xz(yz)$	[**S**-e]

In what follows, the β-reductions and β-expansions always follow a leftmost-outmost reduction strategy so we can omit to specify which combinator is introduced or eliminated (See Example 2).

[1]The brackets designate that we're not using left associativity as it is usually the case in Combinatory Logic.

Example 2.

1	$\mathbf{W}\,\mathbf{B}\,\mathbf{K}xyz$	
2	$\mathbf{B}\,\mathbf{K}\,\mathbf{K}xyz$	[**W**-e]
3	$\mathbf{K}(\mathbf{K}x)yz$	[**B**-e]
4	$\mathbf{K}xz$	[**K**-e]
5	x	[**K**-e]

Can be simplified as:

1	$\mathbf{W}\,\mathbf{B}\,\mathbf{K}xyz$
2	$\mathbf{B}\,\mathbf{K}\,\mathbf{K}xyz$
3	$\mathbf{K}(\mathbf{K}x)yz$
4	$\mathbf{K}xz$
5	x

2.3 A Complexity Threshold

Combinatory Logic is already known as a model of computation but what we are proposing here is to use a logic of combinators and operators to reverse engineer more and more complex computational concept up from the source code. This will lead to what we can call a vertical simplification of the computational concepts: through progressive layers of complexity, we can define high-level concepts from other (less complex) concepts. To the vertical simplification, we will add an horizontal simplification: given that a concept in Combinatory Logic is essentially the application of an operator to an operand, we can use the elementary combinators to extract a meaningful abstract computational operator that is simpler to grasp than the complex concept it composes.

As we go through the process of building up complex computational concepts out of simpler ones in the next section, we will explorer both ways of limiting the cognitive complexity with a particular focus on the vertical simplification. We will explore the horizontal simplification in further details in Section 5.

3 From Code to Higher-Level Concepts

This section describes the process of building up complex computational concepts in the framework of Combinatory Logic. If we had to give a name to this formal apparatus, it would be something like *computational logic of operators*. Although the approach is formal, we will not give an exhaustive account of it, particularly on the programming side where so many variants are possible. However, we will give the principles of this method for building computational concepts, illustrated with extensive examples that are aiming at: (1) showing how the method works with enough detail to be able to apply it to a larger domain ; (2) demonstrating how such computational concepts can considerably reduce the cognitive complexity of a program, i.e. how this method can make a complex piece of code much easier to grasp.

3.1 Primitive Computational Concepts

Primitive Computational Concepts are the most primitive operations of computation. Those will depend on the level of language we're abstracting from. Typically for a programming language, primitive operations will be arithmetic, boolean or memory operations such as addition, multiplication, negation, declaration or assignment. One could look at multiplication as non-primitive because built upon the more primitive concept of addition. Now, arguably, multiplication is not a complex notion for the human cognition. By *complex* notion, we do not mean the subjection notion but precisely the precise meaning that it is combining more primitive notions. Multiplication *is* such a combination with respect to addition but it is not perceived as such by people educated in arithmetic. It is truly perceived as a primitive notion.

3.1.1 Example: Incrementing

Incrementation is undoubtedly a primitive notion of computation[2].

Let *Inc* the incrementing operation which can be implemented as follow:

Example 3.

```
x = x + 1;
```

Or:

Example 4.

```
x++;
```

The operator may be applied to natural number to yield the following evaluations:

$$Inc\,1 \geq_{eval} 2$$

$$Inc\,2 \geq_{eval} 3$$

Inc is the abstract operation of incrementing. We could also express this abstraction from the second example of implementation above:

$$G \equiv \_++;$$

The operator *Inc* can be considered independently of any variable or constant: it just signifies "incrementing." As defined here, *Inc* is an *primitive computational concept* because it is defined independently of any other operator.

By itself, *Inc* is very easy to grasp. Naturally, it usually doesn't occur by itself. Incrementing is always used within a computational context, like a conditional or a loop for instance. The general interpretation of what "incrementing" means in the application domain depends on how it is combined with other primitive programming concepts. Such combination is the topic of the next section.

[2]Often presented as the successor function in computational models.

3.2 Combination of Concepts

We're now going one level up in the layers of abstraction, thus introducing the important role of the combinators with respect to our goal: organizing of the computational chaos. Combinators are operators that combine the primitive computational concepts in a certain applicative order to build more complex operations.

Let us give some examples based on the primitive operation of incrementing.

3.2.1 Example 1: Repetition

Let us consider the following snippet of code implementing an operation of incrementation repeated twice:

Example 5.

```
x = 1;
x = x++;
x = x++;
```

To express the double application of the incrementation operation, we will need:

- the combinator of duplication **W** (to express the duplication of the same operation);
- the combinator of composition **B** (to express the composition of the two operations).

The double incrementation in Example 5 can be formalized as

(Double Incrementation of 1) **W B** *Inc* 1

Indeed we have the following natural deduction:

$$\begin{array}{c|l} 1 & \mathbf{W}\ \mathbf{B}\mathit{Inc}1 \\ 2 & \mathbf{B}\mathit{Inc}\ \mathit{Inc}1 \\ 3 & \mathit{Inc}(\mathit{Inc}1) \end{array}$$

Where the levels of abstraction can be interpreted as follows:

- **W B** *Inc* 1 means "increment 1 twice". In pseudo-code:

```
x = 1;
x = x++;
x = x++;
```

- **W B** *Inc* means "increment twice". In pseudo-code, we could represent the abstraction as

```
x = _;
x = x++;
x = x++;
```

- **W B** means "apply twice". In pseudo code, we could have somehting like

```
x = _;
x = __(x);
x = __(x);
```

Note that, as we go more and more into abstract operators, it becomes harder to express the idea in source code format. And this is precisely the point: abstracting *from the source code* to express computational operations via higher-level concepts.

Moreover, beside the fact that we are operating this move toward abstraction, the fact that Combinatory Logic doesn't contain any variables x or y to express concepts makes the implementation-level representation of operators more complicated because code is highly dependent on the use of variables.

3.2.2 Conditionals

Incrementation can also be part of a conditional block:

Example 6.

```
x = 1;
if (Boolean) then{
        x++;
    }
    else{
        x++;
        x++;
    }
```

Booleans can be represented as the following combinators:

$$\mathit{True} \equiv_{def} \mathbf{K}$$
$$\mathit{False} \equiv_{def} \mathbf{C\ K}$$

And the negation defined as

$$\mathit{Not} \equiv_{def} \mathbf{B}^3\ \mathbf{C}^2\ \mathbf{I}\ \mathit{FalseTrue} \equiv \mathbf{B}^3\ \mathbf{C}^2\ \mathbf{I}\ (\mathbf{C\ K})\ \mathbf{K}$$

Indeed we have

1	Not True
2	[Not True $\equiv_{def}$ $\mathbf{B}^3\ \mathbf{C}^2\ \mathbf{I}\ (\mathbf{C\ K})\ \mathbf{K\ K}$]
3	**B B B C C I (C K) K K**
4	**B (B C) C I (C K) K K**
5	**B C (C I) (C K) K K**
6	**C (C I (C K)) K K**
7	**C I (C K) K K**
8	**I K (C K) K**
9	**K (C K) K**
10	**C K**
11	[False $\equiv_{def}$ **C K**]
12	False

And

1	Not False
2	[Not False $\equiv_{def}$ $\mathbf{B}^3\ \mathbf{C}^2\ \mathbf{I}\ (\mathbf{C\ K})\ \mathbf{K}\ (\mathbf{C\ K})$]
3	**B B B C C I (C K) K (C K)**
4	**B (B C) C I (C K) K (C K)**
5	**B C (C I) (C K) K (C K)**
6	**C (C I (C K)) K (C K)**
7	**C I (C K) (C K) K**
8	**I (C K) (C K) K**
9	**C K (C K) K**
10	**K K (C K)**
11	**K**
12	[True $\equiv_{def}$ **K**]
13	True

So we have:

$$\mathit{NotTrue} \geq_\beta \mathit{False}$$

$$\mathit{NotFalse} \geq_\beta \mathit{True}$$

We can now define the computational operation in Example 6 as:

$$\mathbf{I}\ \mathit{Boolean}\ \mathit{Inc}\ (\mathbf{W\ B}\ \mathit{Inc})\ 1$$

Assume that Boolean is *True*. We have the following deduction:

1	$\mathbf{I}\mathit{Boolean\ Inc}(\mathbf{W\ B}\mathit{Inc})1$
2	[Boolean $\equiv$ *True* $\equiv_{def}$ **K**]
3	$\mathbf{I\ K}\mathit{Inc}(\mathbf{W\ B}\mathit{Inc})1$
4	$\mathbf{K}\mathit{Inc}(\mathbf{W\ B}\mathit{Inc})1$
5	$\mathit{Inc}1$

Which formally represents the operation of incrementing 1. While if, on the contrary, Boolean is *False*:

1	$\mathbf{I}\mathit{Boolean\ Inc}(\mathbf{W\ B}\mathit{Inc})1$
2	[Boolean $\equiv$ *False* $\equiv_{def}$ **C K**]
3	$\mathbf{I\ (C\ K)}\mathit{Inc}(\mathbf{W\ B}\mathit{Inc})1$
4	$\mathbf{C\ K}\mathit{Inc}(\mathbf{W\ B}\mathit{Inc})1$
5	$\mathbf{K}(\mathbf{W\ B}\mathit{Inc})\mathit{Inc}1$
6	$\mathbf{W\ B}\mathit{Inc}1$

Which formally represents the operation of incrementing 1 twice (see Section 3.2.1).

We thus have obtained the expected behavior from Example 6: if Boolean is true, increment 1, else increment 1 twice.

We can now explore the meaning of the complex operator of conditionals:

- **I** *Boolean Inc* (**W B** *Inc*) 1 means "increment 1 once if *Boolean* is true, twice otherwise". In pseudo-code:

```
x = 1;
if (Boolean) then{
        x++;
    }
    else{
        x++;
        x++;
    }
```

- **I** *Boolean Inc* (**W B** *Inc*) means "increment once if *Boolean* is true, twice otherwise". In pseudo-code:

```
x = _;
if (Boolean) then{
        x++;
    }
    else{
        x++;
        x++;
    }
```

- **I** *Boolean* means "apply first operand if *Boolean* is true, second operand otherwise". In pseudo-code:

```
__;
if (Boolean) then{
        __;
    }
    else{
        __;
    }
```

While the concepts of *incrementation* and *double incrementation* are easy to name, the concept *incrementing-once-if-Boolean-is-true-twice-otherwise* is more complex to define as a unified notion. Although it is not impossible. One could consider that the concept of *conditioned repetition of incrementation* would be accurate, assuming that it is either one or two incrementations. Defining and naming the combined concept is a crucial step in the conceptualization of source code.

3.3 Defining Complex Concepts

We said earlier that the progressive abstraction form the source code is crucial for the comprehension of the code. We're now going to see that the key to get a progressive abstraction from the source code is to establish intermediary meaningful layers.

We just saw how we could combine primary computational concepts to form more complex computational concepts. Naming and defining concepts is what makes it possible to zoom out from the source code without increasing the cognitive complexity.

To see that more clearly, let us examine the examples we just looked at.

From the incrementation snippet...

```
x = 1;
x = x++;
```

...to the double incrementation snippet

```
x = 1;
x = x++;
x = x++;
```

...to the conditioned repetition of incrementation snippet

```
x = 1;
if (Boolean) then{
        x++;
    }
    else{
        x++;
        x++;
    }
```

we can say that we've "zoomed out" in the sense that the third snippet is clearly a superset of the second, and the second of the first. Now, Can we say that the combinatorial representation of computational concepts allow us to keep a constant level of cognitive complexity while zooming out?

If we represent the second and third snippets as direct combination of primitive computational concept, it will not appear so:

(Double Inc.)	**W B** *Inc*
(Conditioned Repet. of Inc.)	**I** *Boolean Inc* (**W B** *Inc*)

The latter is clearly more complex than the former. Actually, as we monotonically increase the complexity of the code by zooming out, so are we increasing the complexity of the corresponding combinatorial expression. To maintain a low level of complexity, we have to advantage of the progressive construction of more and more complex operators by making sure that newly constructed complex concepts are *named*:

(Double Inc.)	$Inc^2 \equiv$ **W B** *Inc*
(Conditioned Repet. of Inc.)	**I** *Boolean Inc* (Inc^2)

Now the two have a comparable level of complexity.

3.4 Higher-level Concepts

As we move up in the layers of abstraction by defining and naming more and more complex operators built upon other (less complex) operators, we are able to keep a reasonable and constant level of cognitive complexity because the less complex operators correspond to well known and well defined high-level computational concepts[3].

Let us assume that we have built complex computation concepts all the way up to the followings:

- S_b: Bubble Sort
- F^T_{100}: First element greater than 100

We can defined the following higher-level computational concept:

$$Class_{100} \equiv \mathbf{B}\ F^T_{100} S_b$$

What is the meaning of $Class_{100}$? Let us β-reduce $Class_{100}$ applied to a random array of numbers, say temperatures: *Temp*.

[3]which may be related to the application domain or not.

```
1   $Class_{100}\,Temp$
2   $\mathrm{Class}_{100} \equiv \mathbf{B}\ F^{T}_{100}\,S_b$
3   $\mathbf{B}\ \mathrm{F}^{T}_{100}\,S_b\,Temp$
4   $\mathrm{F}^{T}_{100}\,(S_b\,Temp)$
5       | $\mathrm{Temp}_S \equiv S_b\,Temp$
6   $\mathrm{F}^{T}_{100}\,Temp_S$
```

$Class_{100}$ applied to $Temp$ is β-reducible to applying F^{T}_{100} (i.e. looking for the first element greater than 100) to $Temp_S$, a sorted version of $Temp$. So $Class_{100}$ can be interpreted as "find the first element greater than 100 in the sorted version of $Temp$". In other words $Class_{100}$ is a classifying operator that separates the temperatures into two subsets: those greater than 100 and the others.

Here again, the definition of $Class_{100}$ remains of very low cognitive complexity because it rests upon the two other complex computation concepts S_b (sorting) and F^{T}_{100} (first element > 100).

3.5 Using loops to create new concepts

So far, we have seen that any computational concept is either a primitive computational concept (e.g. incrementing) or a complex computational concept recursively defined over combinators and primitive concepts. Loops, however, do not correspond to one nor the other. They create new customized concepts from primitive or complex concepts with the use of a new loop operator Σ.

To handle the loop behaviors, we need a topological semantics: a semantics of space and of movement in that space. Locations in that space are analogous to actual physical memory locations in the machine.

Let us take the example of the computation concept F^{T}_{100} (Section 3.4) which could be implemented as follows:

Example 7.

```
i := 0;
While (arrayTemp[i] < 100){
      i:=i+1;
}
z := arrayTemp[i];
```

A loop-concept is a primitive computational concept created within the program.

Σ is a special operator that generates primitive computational concepts[4] by taking as its operands a movement M and an initial position P_0 in the memory topology:

- A position in memory is any defined location in a data structure, whether it is a cell in an array, a record in a database, a field in a table or a line in a file.
- A movement is an incremental change of position in the data structure, like moving to the next/previous cell, to the next record, line or field.

[4]More precisely, those primitive computational concepts are primitive computational predicates

Note that by memory topology, we do not necessarily mean the actual physical memory location on a device but any kind of location (virtual or physical). Obviously, the encoding of those notions will depend on the encoding of the program data structure. But, in general, if we refer to the computational model of Turing machines, we can show that any incremental movement in a data structure can be reduced to scanning through a tape (array). The initial position and the movement are analogous to the base case and inductive case in recursive functions.

Example 8. In Example 7, we have:

- The incremental movement M is "moving to the next cell in the array arrayTemp[]". It is represented in the code by:
  ```
  While (arrayTemp[i] < 100){
  i:=i+1;
  }
  ```
- The initial position P_0 is "the first cell" and is represented in the code by:
  ```
  i := 0;
  While (arrayTemp[i] < 100){
  i:=i+1;
  }
  ```

As we just said, Σ is a special operator that generates primitive computational concepts by taking as its operands a movement M and an initial position P_0. In the code, Σ corresponds to the inductive application of the movement to the initial position. So the following applicative operation

$$\Sigma M P_0$$

is a primitive computational concept and we can write:

$$T \equiv_{def} \Sigma M P_0$$

T may be interpreted as the concept of "being in tempArray". In the application domain it means "being a temperature". T is a primitive computational concept[5] in the program.

From this primitive concept, we can easily build more complex concepts by adding layer of determination. Adding determination to a concept consists in increasing its intension[6] by adding new qualifying attributes to the concept. For example, the determination "blue" can be added to the concept "car" to build the new concept "blue car". As the Logic of Port Royal [8] points out, adding determinations will always increase the intension of the concept and decrease its extension (there are less blue cars than there are cars).

The operator of determination δ take as operands two concepts to form a new concept: the first operand adds determination to the second one.

[5]Predicate
[6]The intension of a concept is its meaning. Increasing the intension can be seen as "eniching" the concept.

Example 9. Using the δ operator we can add the determination blue to the concept of car:

$$\delta Blue\, Car$$

Primitive loop concepts add determination through conditional within the loop block: first, through the loop guard; second, through the additional conditionals in the loop block. In Example 7, determination is added through the loop guard itself:

Example 10. In the following snippet of code

```
i := 0;
While (arrayTemp[i]      100){
    i:=i+1;
}
z := arrayTemp[i];
```

The guard is adding the determination "first greater than 100".

Let us call F_{100} the boolean concept associated with the while loop guard in Example 7. F_{100} means "the first greater than 100" (it doesn't say the "first what") and is represented in the code by:
While (arrayTemp[i] **< 100**){

We already defined T the primitive loop concept. So we have

$$F_{100}^{T} \equiv_{def} \delta F_{100}\, T$$

Meaning "From the temperature array" with the additional determination "greater than 100", i.e "the first temperature greater than 100." We've now formally constructed, from the primitive loop concept T, the complex concept F_{100}^{T} mentioned in Section 3.4.

The guard can add all kind of determination to the primitive loop concept but it can also add the empty concept determination $\delta\varepsilon \equiv \mathbf{I}$ like in the following snippet:

```
i := 0;
While (arrayTemp[i] != NULL){
    i:=i+1;
}
```

Assuming that NULL is the value of the first empty cell, the only complex concept here will be:

$$T' \equiv_{def} \delta\varepsilon T \equiv T$$

When the loop guard is empty, the conditionals within the loop block can generate more complex concepts through additional determination. We will only show an example here without getting into the details[7]. Consider the following snippet of code with an empty determination in the loop guard and a conditional inside the loop to check if the value is grater than 100:

```
i := 0;
While (arrayTemp[i] != NULL){
    if arrayTemp[i] > 100{
        array2.add(arrayTemp[i]);
    }
    i:=i+1;
}
```

The additional determination is given by the conditional guard. Note that such a conditional in the loop block is not going to select only the first greater than 100 (F_{100}) but *all the values* greater than 100: A_{100}. We thus have the following new complex concept:

$$A_{100}^{T} \equiv_{def} \delta A_1 00(\delta\varepsilon T) \equiv A_1 00T$$

The complex concept A_{100}^{T} means "all the temperatures greater than 100".

4 Computational concepts and cognitive complexity

We've already hinted to the fact that the cognitive complexity of combinatorial computational concepts is close to invariant under increasing scope of source code (what we've called "zooming out" from the source code), just like the complexity of a map doesn't increase as the user zooms out of a certain place on Google Map.

With the capability to express loop computational concepts, we can now make this point more clear.

Suppose the following snippet of code:

```
if Boolean then{
    i := 0;
    While (arrayTemp[i] < 100){
        i:=i+1;
        }
    z := arrayTemp[i]
    z++;
    z++;
    }
    else{
    i := 0;
    While (arrayTemp[i] < 100){
        i:=i+1;
        }
    z := arrayTemp[i]
    z++;
    }
}
```

We can use the primitive concepts, loop concepts and combinators we've examined so far to define the corresponding complex concept:
(C_1) $\quad X \equiv_{def} \mathbf{I}\, Boolean\, (Inc^2\, F_{100}^{T})(Inc\, F_{100}^{T})$

Which can be expressed as "the conditioned double incrementation of the first temperature greater than 0." This high-level computational concept X combines other (less) complex concepts:
$(C_2.1)$ $\quad F_{100}^{T} \equiv_{def} \delta F_{100} T$
$(C_2.2)$ $\quad Inc^2 \equiv_{def} \mathbf{W}\,\mathbf{B}\, Inc$

[7] See subsequent paper for more on loop concepts.

$(C_3) \qquad T \equiv_{def} \Sigma MP_0$

We can define a tree of computational concepts[8] where the root is the highest level and most complex concept (the complete scope of the code) and the leafs are the primitive computational concepts as defined previously. Going up and down the tree corresponds to zooming in and zooming out on the source code. Just like on Google Map, the zooming in and out doesn't mean an decrease or increase in cognitive complexity.

As a matter of fact, each of the definition above (C_1, $C_2.1$, $C_2.2$, C_3) are of *comparable level of cognitive complexity* despite their different size of scope on the source code. The combinatorial abstraction into computational concepts is robust with respect to cognitive complexity.

5 Abstract Computational Operators

We have said earlier (Section 2.3) that there are two ways to reduce the cognitive complexity of a high-level concept expressed in Combinatory Logic that we can call the vertical and horizontal way:

- Define and name intermediary concepts that are contained in the complex concept;
- Define and name abstract operators of this concept.

We've extensively looked at the vertical way in Section 3. We've also briefly mentioned some aspects of the second, as in the following:

- **W B** *Inc* 1 means "increment 1 twice".
- **W B** *Inc* means "increment twice".
- **W B** means "apply twice".

Although this is not a complex one, this small example can help us clarify what we mean by: *defining and naming abstract operators of a complex concept can reduce the cognitive complexity of that concept*.

Let us remember that a combinatorial expression is assuming left associativity and is, in essence, the application of an operator to an operand. If we can divide a combinatorial expression in two parts, we have now an abstract operator applied to an operand. Those two concepts might be easier to grasp than the whole. This is, in some sense, a divide-and-conquer strategy for reducing the cognitive complexity of the expression.

As explained above **W B** *Inc* 1 contains the abstract operator **W B** *Inc* which means "increment twice". So we can divide the complex computational concept into two parts: Increment Twice | 1. We simply need to understand "1" and "increment twice" instead of the whole complex concept. If **W B** *Inc* ("increment twice") is still too complex to grasp, we can divide it again into two part as the above abstraction suggests: Apply Twice | Increment. Each of those concepts are simpler to grasp than the application of one to the other.

[8]Analogous to the computational tree in recursive theory [9]. Although, those computational tree do not have the capacity to reduce multiple levels of function because recursive functions don't include combinators. For example, S(S(S(0))) is irreducibly on 4 levels in the tree.

The power of Combinatory Logic lies in the capacity of reorganizing the combination of operators (hence the name) in order to build meaningful abstract operators. Without the **W** and **B** combinators, **W B** *Inc* 1 would be expressed as Inc(Inc(1)). This latter expression cannot separate the concept of incrementing from the concept of applying twice because the two are entangled.

This powerful feature of combinatory logic will find a better illustration in the complex computational concept X:
$X \equiv_{def} \mathbf{I}\, Boolean\, (Inc^2\, F^T_{100})(Inc\, F^T_{100})$

As it is defined now, there is no much interest in abstracting... It would be just like reducing the scope of the code...

We have the following β-expansion:

1	$[X \equiv_{def} \mathbf{I}\ Boolean\, (Inc^2\, F^T_{100})(Inc\, F^T_{100})]$
2	$\mathbf{I}$ Boolean $(\text{Inc}^2\, F^T_{100})(Inc\, F^T_{100})$
3	$\Phi(\mathbf{I}$ Boolean$)$ $\text{Inc}^2 Inc F^T_{100}$
4	$\mathbf{B}\,\Phi(\mathbf{I}$ Boolean$)$ $\text{Inc}^2 Inc F^T_{100}$
5	$[X_1 \equiv_{def} \mathbf{B}\ \Phi\ (\mathbf{I}\ Boolean)\ Inc^2 Inc]$
6	$[X \equiv_{def} X_1 F^T_{100}]$

We can now describe the complex concept X as the application of:

X_1: "The conditioned doubling of the incrementation" (meaning, incrementation is doubled only if *Boolean* is true)
onto:

F^T_{100}: "The first temperature greater than 100."

Here again, X_1 can be further abstracted if needed.

Note that, although we can now focus on X_1 instead of X (thanks to the combinators **B** and Φ), this is not a matter of zooming into a subset of the initial code. It truly is an abstraction. In the source code, the abstract operator X_1 could be represented as:

```
if Boolean then{
    z := ____
    z++;
    z++;
    }
    else{
    z := ____
    z++;
    }
}
```

Abstract computational operators shed a new light on the understanding of a complex computational concept as the application of a simpler operator to a simpler operand. The Combinators make it possible to build the most simple and meaningful operator/operand separation.

6 Conclusion

The starting point of this paper was to propose a solution to the overwhelming cognitive complexity of source code in large applications. By *cognitive complexity*, we mean a level of intricacy in the code that makes it very difficult to grasp for a human mind.

We have proposed to use the framework of Combinatory Logic for constructing complex computational concepts as a simple model of description of the source code of a program. By "simple model of description" we mean a model of description that doesn't increase in cognitive complexity as the complexity or size of the code increases.

In this paper, we have given the foundations of this formal method by defining the primary notions of primitive computational concepts, primitive loop concept, operation of determination and complex computational concept.

Through a series of examples, we have shown that the applicative framework of Combinatory Logic allows for setting a complexity threshold by defining intermediary concepts that are contained in the more complex concepts. As complex computational concepts are combined (using combinators) into even more complex concepts, the complexity of the former doesn't have to be included into the latter as it has been already ontologically defined.

Finally, we have explored an alternative way of reducing the cognitive complexity of a computational concept: abstract computational operator. Using combinators and the fundamental idea that a complex computational concept is the application of a simpler operator to a simpler operand, we can now divide a complex concept into two simpler ones in order to better grasp the complex application as a whole.

References

[1] S.b.J. Somers, *The coming software apocalypse* (2017), `https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/`

[2] A. Church, Noûs **7**, 24 (1973)

[3] H.B. Curry, R. Feys, W. Craig, *Combinatory logic* (North-Holland Publishing Company, 1958)

[4] H. Curry, *Outlines of a formalist philosophy of mathematics* (North-Holland, 1970)

[5] F.B. Fitch, *Elements of combinatory logic* (Yale University Press, 1974)

[6] W.V. Quine, *Word and Object* (M.I.T. Press, 1960)

[7] D. Jean-Pierre, G. Gaëll, B. Sauzay, *Logique combinatoire et lambda-calcul: des logiques d'opérateurs* (Cépaduès-éditions, 2016)

[8] A. Arnauld, P. Nicole, *La logique ou L'art de penser* ... (1741)

[9] P. Odifreddi, *Classical recursion theory. the theory of functions and sets of natural numbers* (1999)

[10] H. Bergier, D. Jean-Pierre, M.D. Popelard, T.G. Pavel, A. Vaetus, L. Peridy, Ph.D. thesis (2016)

[11] D.J. P., *Langages applicatifs, langues naturelles, et cognition* (Hermès, 1990)

[12] F. Maribel, *Models of computation: an introduction to computability theory* (Springer, 2009)

[13] D.S.O. KROENING, *DECISION PROCEDURES: an algorithmic point of view* (SPRINGER, 2019)

[14] M. Sipser, *Introduction to the theory of computation* (Thomson Course Technology, 2006)

[15] A. Vaetus, D. Jean-Pierre, Ph.D. thesis (2001)

[16] T. Biggerstaff, B. Mitbander, D. Webster, Proceedings of 15th International Conference on Software Engineering (1993)