Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

Text Processing Chains: Getting Help from Typed Applicative Systems

Marie Anastacio, Ismaïl Biskri
LAMIA-Université du Québec à Trois-Rivières
{Marie.Anastacio ; Ismail.Biskri}@uqtr.ca

Abstract

A processing chain must be the result of a discovery process that requires constant back and forth between theoretical description of the solution, software implementation, testing and refinement of the theoretical description in the light of the results of experimentation. This process is iterative. Some projects based on this philosophy have seen the light in the last years. However, they lack flexibility and formal foundations. The model we propose has strong logical foundations. It allows rapid prototyping and supports a maximal re-use and composition of existing modules.

Introduction

"Language and text processing" is a broad field of research including retrieval, classification, and information's analysis. As Web is a big source of information, this field can have a lot of implications on several sectors of society. Compared to the quick expansion of data quantity, the evolution of their analysis is too slow and insufficient.

A big challenge in these fields is the multitude of disciplines needed to go further. So, experts of different domains need to work together.

In the literature about data-mining and text-mining, many projects aim to allow the creation of complex processing chains. Aladin (Seffah, Meunier, 1995), D2K/T2K (Downie & al., 2005), RapidMiner (Mierswa et al., 2006), Knime (Warr, 2007) and WEKA (Witten et Al., 2011) use processing chains for language engineering, Gate (Cunningham et Al., 2002) use it for linguistic analysis. The processing chains are widely used, but the solutions previously mentioned suffer from limitations. They are strongly bonded to their specific platforms and programming languages. To take the best advantage of them, the user needs to have knowledges about the developed software and sometimes about programming language.

In previous papers (Biskri et al., 2015; 2013), we presented a flexible and modular architecture for text processing. Each task is represented by a function that is independent from the others. Inspired by Applicative and Combinatory Categorial Grammar (ACCG) (Biskri, Desclés, 1997), our model is composed of rules based on combinatory logic and typed applicative systems. As some rules of the model were limited to functions taking one or two inputs, the current paper presents the extension of this model to functions with any number of inputs.

Combinatory Logic and Typed Applicative Systems

Combinatory logic was introduced with the work of Schönfinkel in 1924 and later extended by Curry and Feys (Curry, Feys, 1958; Hindley, Seldin, 2008). The notion of combinators was introduced with the purpose of bringing a logical solution to some paradoxes, such as Russel's Paradox, and to eliminate the need of the variables in order to avoid variable telescoping. Combinators are abstract operators. They act as functions over argument within an operator-operand structure. Each rule is represented by a unique rule called β-reduction; which defines the equivalence between the logical expression without combinator and the one with combinator. Elementary combinators can be associated to others to create complex combinators. Our model uses only the four elementary combinators, whose notations and β-reductions are shown in the table below.

Combinator	Role	β-Reduction
B	Composition	**B** x y z → x (y z)
C	Permutation	**C** x z y → x y z
S	Distributive composition	**S** x y u → x u (y u)
W	Duplication	**W** x y → x y y

B, **C**, **S** and **W** are elementary combinators.

The composition combinator **B** combines two operators x and y together and form the complex operator **B** x y that acts on an operand z.

The permutation combinator **C** uses an operator x in order to build the complex operator **C**x that acts on the same two operands as x but in reverse order.

The composition combinator **S** distributes an operand u to two operators x and y.

The duplication combinator **W** takes an operator x that acts on the same operand twice and form the complex operator **W**x that acts on this operand only once.

We can combine those combinators together to create complex combinators. For example, we could have an expression such as "**S B C** x y z u v". Its global action is determined by the successive application of its elementary combinators (first **S** secondly **B** and finally **C**).

S B C x y z u v
B x (**C** x) y z u v
x (**C** x y) z u v
x x z y u v

The resulting expression, without combinators, is called a normal form. This form, according to Church-Rosser theorem, is unique.

Two forms of complex combinators got their own notation. The power and the distance of an existing combinator.

Let χ be a combinator. The power of a combinator, written as a superscript, represents the number of times its action must be applied. It is defined by $\chi^1 = \chi$ and $\chi^n = \mathbf{B}\ \chi\ \chi^{n-1}$. For example, the action of the expression $\mathbf{W}^2$ a b would be:

$\mathbf{W}^2$ a b
B W W a b
W (**W** a) b
W a b b
a b b b

The distance of a combinator, written as a subscript, represent the number of steps its action is postponed. It is defined by $\chi_0 = \chi$ and $\chi_n = \mathbf{B}^n\ \chi$. For example, the action of the expression $\mathbf{C}_2$ a b c d e will be:

$\mathbf{C}_2$ a b c d e
$\mathbf{B}^2$ **C** a b c d e
C (a b c) d e
a b c e d

For our model, we represent the type of our combinatorial expressions with notations taken from applicative systems.

Applicative systems represent functions with any number of inputs and one output. To each operand is associated a type and each function has an applicative type. These applicative type starts with "F". Types are defined as follows:

1. Basic types are types.
2. If x and y are types, Fxy is a type.

For example, if x and y are types, a function having an x typed input and returning an y typed operand will be of type Fxy. A function having two x typed inputs and returning a y typed operand will be of type FxFxy.

This can also be read as: a function taking an x type input and giving back a function taking an x type input and returning an y type output.

We use this notation to represent a combinatorial expression type. For example, if w is an operand that takes two inputs of types x and y and returning a z typed output, his type is FxFyz. When we apply the **C** combinator to it, we form a new operand **C**w of type FyFxz.

Formal Model

Our model refers to programs as modules. They are organized in series and as such they form processing chains. A module acts like a mathematical function that takes several arguments, process them and return an output. We are not interested in the internal programming of the modules but only in their representation as functions and how they are organized to create processing chains.

A processing chain must respect two rules:

1. The chain must contain at least one module
2. The chain must be syntactically correct
3.

The semantic aspect is user's responsibility.

Our model tends to answer two questions:

- Given a set of modules, what are the allowable arrangements that lead to coherent processing chains?
- Given a coherent processing chain, how can we automate as much as possible its assessment?

Concretely, a module applies an operation to one or many entities and returns a new entity. We can therefore assign an applicative type to it.

We note the module named M1 of type Fxy as follow: [M1 : Fxy], and represent it as in fig. 1 . A processing chain is the representation of the order of application of several modules on their inputs. To be valid, the type of an input must be the same as the output linked to it (fig. 2). It also can be seen as a module itself as it has inputs and output (fig. 3).

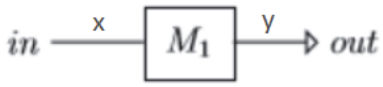

Figure 1 – Module schematisation

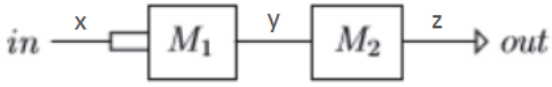

Figure 2 – Valid chain of two modules in series

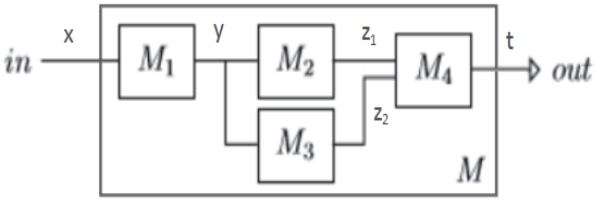

Figure 3 – Processing chain as a new module

Our model allows the reduction of a processing chain to this unique module representation. The combinatory logic keeps the execution order and the rules take type in account to check the syntactic correctness. To reduce a chain, we only need the module list, their type, and their execution order.

The previous model has some limitations. It doesn't manage with modules with a limited number of inputs. Let us show these rules :

APPLICATIVE RULE	[X : x] + [M1 : Fxy] ---------- [Y : y]
COMPOSITION RULE	[M1 : Fxy] + [M2 : Fyz] ----------**B** [**B** M2 M1 : Fxz]
DISTRIBUTIVE COMPOSITION RULE	[M1 : Fxy] + [M2 : FxFyz] ----------**S** [**S** M2 M1 : Fxy]
PERMUTATION RULE	[M1 : FxFyz] ----------**C** [**C** M1 : FyFxz]
DUPLICATION RULE	[M1 : FxFxy] ----------**W** [**W** M1 : Fxy]

The above rules are only the core set of the previous model. We will extend the rules so they can be applied to any number of inputs. To do this we'll need some new notations.

[M1 : $Fx_1...Fx_ny$] is a module M1 with n inputs of different types, input in place "i" is of type x_i, and an output of type y.

[M1 : $(Fx)^ny$] is a module M1 with n inputs of type x and an output of type y.

Composition rule

[M1 : $Fx_1...Fx_ny$] + [M2 : Fyz]
---------- $\mathbf{B^n}$
[$\mathbf{B^n}$M2M1 : $Fx_1...Fx_nz$]

The composition rule is used when two modules are in series (as in fig. 2). If M1 has n inputs, the power of the B combinator is n. For these rules, the inputs number of M2 can be more than one.

Duplication rule

[M1 : $(Fx)^ny$]
---------- $\mathbf{W^n}$
[$\mathbf{W^n}$M1 : Fxy]

The duplication rule transforms a module with n identical inputs to a module with only one input. It can be applied only if the chain give the same value to each of its inputs (fig. 4).

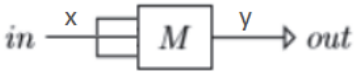

Figure 4 – Module getting a single value in its three inputs

Permutation rule

[M1 : $Fx_1...Fx_ny$]
---------- **C**
[$\mathbf{C_{p-1}}(\mathbf{C_p}(\ldots(\mathbf{C_{m-2}}M1))) : Fx_1...Fx_{p-1}Fx_mFx_p...Fx_{m-1}Fx_{m+1}...Fx_ny$]

The permutation rule allows to change the order of inputs. It takes the input at position m and moves it to the position p, with p<m. It's used to reorganize input to make the other rules applicable.

Application of the Approach

In this section, we will show how the rules given in the previous section are applied and illustrate the reduction of a processing chain with an example.

Let us consider the linear connection of two modules (fig. 2). The module [M1 : FxFxy] applies on two identical inputs of type x and yield an output of type y. The module [M2 : Fyz] applies on this output to yield an output of type z. This chain is expressed by the expression: [M1 : FxFxy] + [M2 : Fyz]. The composition rule can be applied and returns the complex module [$\mathbf{B^2}$ M2 M1 : FxFxy]. If the type of M1 output and M2 output where not the same, we could not have applied the composition rule. So, the application of the rules is a proof of syntactic correctness of the chain.

The module [**B²** M2 M1 : FxFxy] can be reduced a second time with the duplication rule. It is reduced to the complex module [**W** (**B²** M2 M1) : Fxy].

The permutation rule allows to reorganise the inputs of a module to apply another rule. Let M be a module with four inputs of types x, y, z and x and an output of type t : [M : FxFyFzFxt]. Let X be the value given to the first and fourth inputs (fig. 5-a). If the fourth was in second position, we could apply the duplication rule to M. So, we want to move the fourth input to second position. The permutation rule returns the complex module [**C₁** (**C₂** M) : FxFxFyFzt] (fig. 5-b). on this new module, the duplication rule can be applied to get a complex module [**W** (**C₁** (**C₂** M)) : FxFyFzt] (fig 5-c).

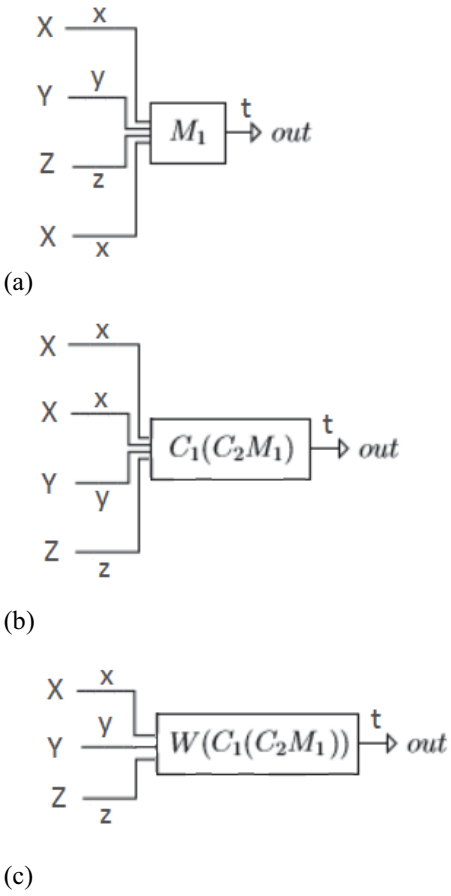

Figure 5 – inputs reorganisation

Let us now give the analysis of a somewhat complex processing chain (fig. 6). This chain is a combination of five modules.

- M1 of type FxFyz
- M2 of type Fzx
- M3 of type Fzx
- M4 of type Fzy
- M5 of type FxFxFyt

To reduce this chain, we will start with the last module and process from left to right. So we start with [M5 : FxFxFyt]. His first input takes the output of [M2 : Fzx]. The composition rule gives a new complex module [**B** M5 M2 : FzFxFyt] (fig. 7). This new module and [M3 : Fzx] can be reduced with the distributive composition rule to get the module [**S** (**B** M5 M2) M3 : FzFyt]. (fig. 8). The first input of this module can be reduced with the composition rule to get a new module [**B²** (**S** (**B** M5 M2) M3) M1 : FxFyFyt]. (fig. 9)

To reduce this module with [M4 : Fzy] we want to use the composition rule. But to apply it, M4 output must be the first input of our module. We use the permutation rule to reorganise the inputs and got a new module [**C** (**C₂** (**B²** (**S** (**B** M5 M2) M3) M1)) : FyFxFyt] (fig. 10). Finally, we can apply the combination rule that returns the module [**B** (**C** (**C₂** (**B²** (**S** (**B** M5 M2) M3) M1))) M4 : FzFxFyt]. As we have only one module, and no other rule can be applied, the processing chain is reduced.

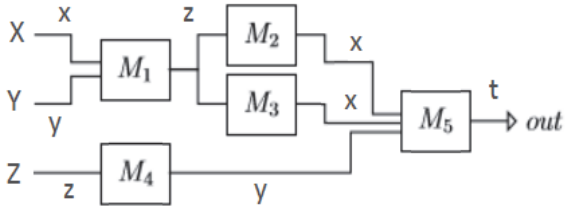

Figure 6 – A complex processing chain

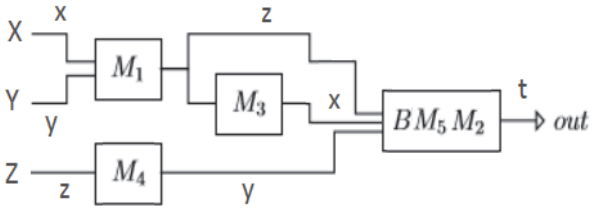

Figure 7 – Reduction step 1

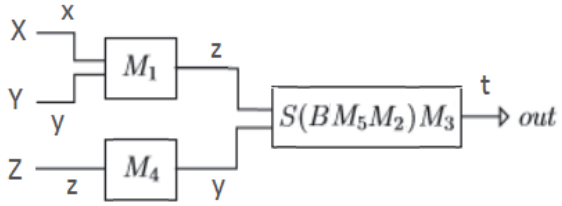

Figure 8 – Reduction step 2

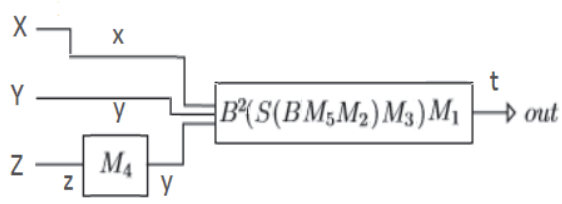

Figure 9 - Reduction step 3

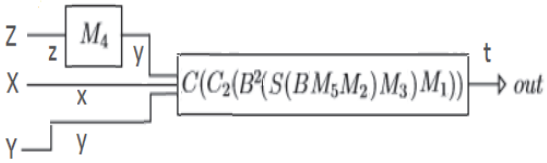

Figure 10 – Reduction step 4

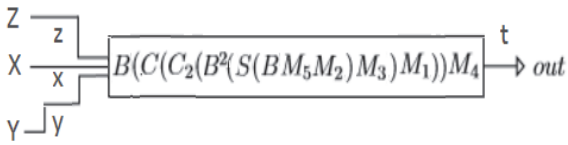

Figure 11 – Reduction last step

As it has been completely reduced, the processing chain is considered as syntactically correct. Its combinatory expression is: **B** (**C** (**C**$_2$ (**B**^{2} (**S** (**B** M5 M2) M3) M1))) M4. Using combinatory logic-reductions, we can get the normal form of this expression.

> **B** (**C** (**C**$_2$ (**B**^{2} (**S** (**B** M5 M2) M3) M1))) M4 Z X Y
>
> **C** (**C**$_2$ (**B**^{2} (**S** (**B** M5 M2) M3) M1)) (M4 Z) X Y
>
> **C**$_2$ (**B**^{2} (**S** (**B** M5 M2) M3) M1) X (M4 Z) Y
>
> **B**^{2} (**S** (**B** M5 M2) M3) M1 X Y (M4 Z)
>
> **S** (**B** M5 M2) M3 (M1 X Y) (M4 Z)
>
> **B** M5 M2 (M1 X Y) (M3 (M1 X Y)) (M4 Z)
>
> M5 (M2 (M1 X Y)) (M3 (M1 X Y)) (M4 Z)

This form contains the order of application of modules on their inputs (X, Y and Z).

Implementation and Experimentations

A prototype of the theoretical model was implemented. The rules are implemented in a F# library and a testing software in C# language. To implement our model we had to know if in some cases there were two or more applicable rules.

As the permutation rule is applicable to any module with more than one input, it is considered separately. This rule is only used to prepare the module to apply another rule. For instance, it can be used to group identical inputs in the beginning to let us apply the duplication rule.

There is two cases of ambiguity. In the first case, we can apply the duplication rule or the composition rule (fig. 12). For this case, we will chose to apply the duplication rule first because the resulting expression is shorter. In the second case, we can apply the composition rule or the distributive composition rule (fig. 13). For the same reason, we will apply the distributive composition rule first.

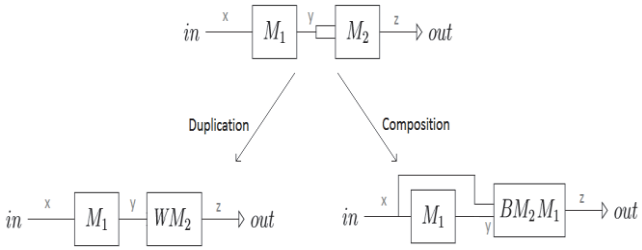

Figure 12 – Ambiguity between composition and duplication

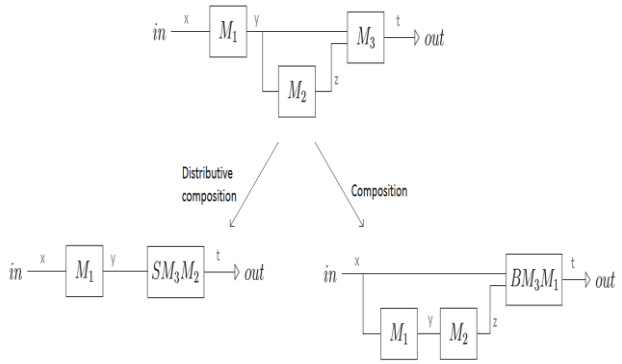

Figure 13 – Ambiguity between composition and distributive composition

The library implements two reduction approaches.

The first approach is the same as the one used in our previous example. The last module of the chain is considered as "current module". The different rules are tested on it and the applicable one is applied. The resulting module is the new current module, and so on. This approach is faster and human readable.

The second approach considers all modules simultaneously and reduces every reduceable module until none is reduceable anymore. This approach is slower but, as it reduces every valid sub chain of the processing chain, it can locate the invalid links inside the chain.

Each approach has been tested on 22 processing chains containing 9 syntactically incorrect chains and 13 correct

chains. The results are shown in table 1 and some of the reduced chains are shown in figure 14.

	REDUCED	NOT REDUCED
VALID CHAIN	13	0
INVALID CHAIN	0	9

Table 1 – Results of reduction

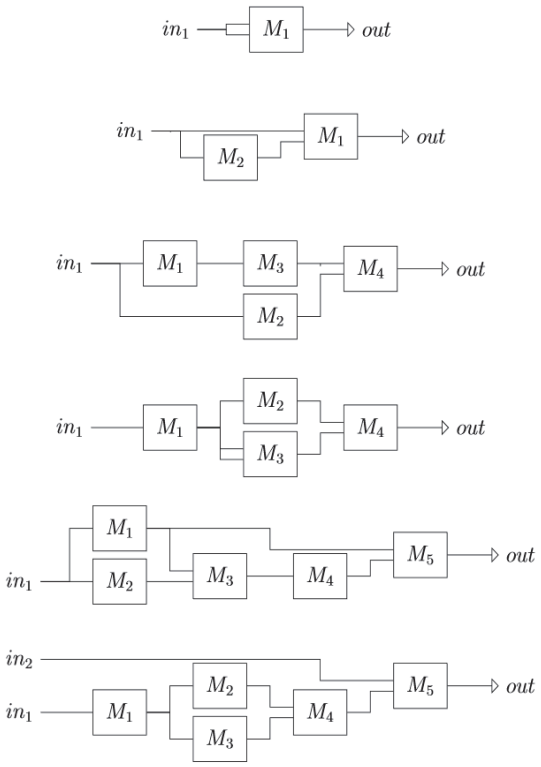

Figure 14 – Set of reduced chains

Conclusion

The need for flexible, adaptable, consistent and easy-to-use tools and platforms is essential. But many challenges are yet to be solved. The user stays in center of its experience and he can change his mind. The flexibility of the tools is really important when it happens. Without it, user needs to constantly go back and forth between theoretical description of the solution, software implementation, testing and refinement of the theoretical description in light of experimentation results. The model that we propose allows rapid prototyping and support a maximal re-use and composition of existing modules. It also ensures a firm compositionality of the different modules in the different processing chains.

Text analysis is only a modality of the theoretical framework developed here. It is possible to adapt this work to other types of data from various disciplines.

References

Biskri, I., Desclés, J.P., 1997. "Applicative and Combinatory Categorial Grammar (from syntax to functional semantics)", In Recent Advances in Natural Language Processing. John Benjamins Publishing Company.

Biskri, I., Anastacio, M., Joly, A., Amar Bensaber, B., 2015. "A Typed Applicative System for a Language and Text Processing Engineering". In International Journal of Innovation in Digital Ecosystems. Elsevier.

Biskri, I., Anastacio, M., Joly, A., Amar Bensaber, B., 2013. "Integration of Sequence of Computational Modules Dedicated to Text Analysis: a Combinatory Typed Approach". In proceedings of AAAI-FLAIRS'13, St-Pete Beach.

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V. 2002. "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications". Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02). Philadelphia.

Downie, J. S., Unsworth, J., Yu, B., Tcheng, D., Rockwell, G., and Ramsay, S. J., 2005. A revolutionary approach to humanities computing: tools development and the D2K datamining framework. In Proceedings of the 17th Joint International Conference of ACH/ALLC.

Curry, B. H., Feys, R., 1958. Combinatory logic, Vol. I, North-Holland.

Hindley, J. R., Seldin, J. P. 2008. Lambda-calculus and Combinators, an Introduction. Cambridge University Press.

Mierswa, I., Wurst, M., Klinkemberg, R., Scholz, M. and Euler, T. 2006. "YALE: Rapid Prototyping for Complex Data Mining Tasks". In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2006), ACM Press.

Seffah, A., Meunier, J.G. 1995. "ALADIN : Un atelier orienté objet pour l'analyse et la lecture de Textes assistée par ordinaleur". International Conferencence on Statistics and Texts. Rome.

Shaumyan, S. K. 1998. Two Paradigms of Linguistics: The Semiotic Versus Non-Semiotic Paradigm. In Web Journal of Formal, Computational and Cognitive Linguistics.

Warr, A. W. 2007. Integration, analysis and collaboration. An Update on Workflow and Pipelining in cheminformatics. Strand Life Sciences.

Witten, I., Frank, E. and Hall, M., 2011. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers.