Applications of Structured Recursion Schemes

Dániel Berényi[1], András Leitereg[2], Gábor Lehel[2]

[1]Wigner Research Centre for Physics, Budapest
[2]Eötvös Loránd University, Faculty of Informatics, Budapest

Abstract: Recursive structures are quite common in various areas of science, especially where expressions of a certain grammar are involved, like in natural languages, programming languages or generally in formal systems. Analysis, transformation and reasoning is closely tied to the structure of these expressions but at the same time the actual operations to be carried out at different levels of the recursive structure may vary considerably. Category theory provides a set of tools that can naturally represent and deal with exactly these constructs. In this work, we review the formulation of recursion schemes over such structures and highlight two use cases: one related to programming language compilation and another one in machine learning.

1. Introduction

Over the centuries, the study of algebra had fundamentally formed our understanding of mathematics. Starting from the solvability of equations with different degree leading to group theory, the theory of linear equation systems developing into linear algebra, and connecting concepts from geometry, number theory and combinatorics, it led to the formulation of abstract algebra and universal algebra to study abstract structures that are common in many different applications and problems arising in and outside of mathematics. During this journey, algebra played a central role in developing the formal system of symbol manipulation that helps navigate the immense net of relations and mechanize calculations. Science adopted mathematics as its language and started to describe more and more complex phenomena, that at some point led to the desire to automatize repeating tasks. Technological advancements made this possible first for arithmetic and later also for symbolic manipulations with the aid of Computer Algebra Systems. Meanwhile many other formal systems developed and lead to important advances in formal logic, type theory and programming. It was then recognized, that category theory possesses the tools to conveniently abstract over all these areas and that the concept of algebras is even more general than previously thought.

In formal systems expressions are built from a given a set of symbols, according to the rules of the grammar of the system. Expressions might have an associated type, that can help to rule out expressions which don't make sense. Types are important in typed programming languages to increase safety and reliability of software but they become crucial in computer proof assistants to prove validity or correctness of algorithms or programs. The expression structures are trees: recursive structures that at each level contain either one connective from the set of possible connectives of the grammar operating over other subexpressions or an axiom as a leaf terminating the tree. Reasoning about expressions thus involve induction over recursive types. It turns out, that such structures can be understood well in type theory by studying fixed points of parametric types. Type theory is heavily utilized in functional programming languages that

all originate from the formal system of λ-calculus developed by Alonzo Church. While λ-calculus provides the operational semantics and evaluation/simplification rules of functional programming languages, different type theories added to it give varying degrees of logical correctness guarantees to its expressions. Incidentally, λ-calculus is also the internal language of Cartesian Closed Categories in category theory, which suddenly opens the way to utilize very abstract theoretical constructions in practical computing.

In this contribution, Section 2 reviews how the recursive types of expressions of formal systems and the associated inductions (and co-inductions) are embedded in category theory in the form of fixed points of endofunctors and the associated algebras and coalgebras. These constructs then lead to the formulation of structured recursion schemes by making use of algebra homomorphisms from a special algebra, the initial algebra that is roughly speaking, the identity evaluation, that completely preserves the structure of the recursive expression. We give two examples of potential real world usage of these constructs: Section 3 outlines a compiler for a simple language implemented solely with recursion schemes, while Section 4 connects these schemes to Recursive Neural Networks in the field of machine learning.

2. Theoretical Background

First, we formulate algebras and coalgebras in the category theoretical setting. A category C is a collection of objects $X \in C$ and morphisms between objects $f: X \to Y \in C$. The algebraic structure of a category is given by the identity morphisms, which start and end in the same object: $id_X: X \to X$ and the binary operation of morphism composition ($\circ$) which is associative: $(f \circ g) \circ h = f \circ (g \circ h)$.

Structure preserving maps between categories called *functor*s play a central role in category theory. If F is a functor between two categories, $F: C \to D$ it maps identities to identities: $F(id_X) \to id_{F(X)}\ \forall X \in C$ and distribute over the composition: $F(g \circ f) = F(g) \circ F(f)\ \forall f, g \in C$. When $C = D$ we refer to them as endofunctors.

Now, an *algebra* in the category theoretical sense, or to be more precise, an F-algebra for an endofunctor denoted F over the category C is a pair of an object $X \in C$ and a morphism $\alpha: F(X) \to X$, and X is called the *carrier type* of the algebra. For example, one might consider the grammar consisting of integer constants (Integer), and the binary operation of addition ($+$), describing expressions (Expr) like: $1, (1 + 2), 1 + (2 + 3), (2 + 3), etc.$ Here, the category consists of types as objects and functions as morphisms, and then the process of calculating the result of such expressions is indeed given by an algebra, which is a function $\mathrm{Expr}(\mathrm{Integer}) \to \mathrm{Integer}$ and the carrier type of this algebra is without doubt Int.

In category theory, most constructions can be dualized and this is also the case for algebras: a *coalgebra* for an endofunctor F over the category C is a pair of an object $X \in C$ and a morphism $\alpha^*: X \to F(X)$. While algebras usually remove or destroy the structure of their input, such as the tree structure of Expr above, coalgebras create structure. As a simple example to the one above, one might devise a coalgebra breaking a given number into a sum of smaller values: $3 \to (1 + 2)$, possibly recursively: $3 \to (1 + 2) \to \big(1 + (1 + 1)\big)$.

Turning now to a universal algebraic setting, sets with algebraic structure are characterized by their signature which represents the allowed operations on that structure. For example, a monoid

over set A is given by the signature $\Sigma = 1 + A \times A$, where 1 stands for the unit element and $A \times A$ is the binary operation. Now, going back to the category theoretical setting, this signature can be considered as an endofunctor over the basis set: $\Sigma(A) = 1 + A \times A$. A matching algebra now should be a mapping: $\alpha: 1 + A \times A \to A$. If we denote the monoid operation as $\cdot$ and the unit element as a, the algebra can be decomposed into a pair of mappings: $\cdot : A \times A \to A$ and $a : 1 \to A$, or $(\cdot, a)$ for short. Similarly, a coalgebra is a mapping $\alpha^*: A \to 1 + A \times A$ that have a similar decomposition.

The duality hidden in the notation of $(\_\,,\_)$ and $\_ + \_$ is not accidental, they are representing the product and coproduct operations that are the category theoretical counterparts of the Descartes-product and the disjoint union operations on sets. From the perspective of logic, these operations represent conjunction and disjunction respectively and this leads to the programming interpretation if we now link the signature with an associated grammar: $\Sigma(A) = 1 + A \times A$ might read as "an expression over A is either the unit element *or* the binary operation applied to two other expressions". So, these endofunctors represent a choice, or a branch, and when nested into themselves they form a tree. When we'd like to evaluate such an expression tree with an algebra α, all the possible cases should be handled so it must have an evaluation rule for the unit element *and* the binary operation, hence the binary product $\alpha = (\cdot, a)$.

In the above formulation one important point should be made clearer, which leads to the abstract formulation of structured recursions. The algebra and the coalgebra operates on only one level of such a tree, when the preceding expression level is already processed. For example, in the case of the evaluator the mapping is $\alpha: F(A) \to A$, where F's recursive positions (such as $A \times A$) are single values of A and not expressions, such as $F(A), F\big(F(A)\big), \ldots$ The recursion both in the structure of the expression and in the formulation of the full evaluator is treated separately from the algebra.

Ever since the invention of the λ-calculus, we have known about the irreducible Y or fixed point combinator: an expression which, when reduced, recreates itself. This construct can be used to describe repeating patterns, as we show with the usual example of the factorial function. Given $\mathrm{fix}(f) = f(\,\mathrm{fix}(f))$, where f is a function taking a function (itself under the image of fix) as first argument, we can write:

$$\text{factorial\_prototype}: (f, n) \to \begin{cases} 1, & n = 0 \\ n \cdot f(f, n-1), & n \neq 0 \end{cases}$$

and then:

$$\mathrm{factorial}(x) \;=\; \mathrm{fix}(\,\text{factorial\_prototype}\,)(x)$$

where function partial application $(f(p))(q) \;=\; f(p, q)$ and lazy evaluation is implied. The constant branch in the prototype definition guarantees that the recursion will end at some point, and that the fixed point of the sequence $\mathrm{fix}(f) = f\big(\,\mathrm{fix}(f)\big) = f\left(f\big(\mathrm{fix}(f)\big)\right) = \cdots$ exists.

We can now use the very same trick at the type level. We need a type level version of fix, that instead of operating on functions, now will operate on functors. We would like to define the datatype representing a recursive tree, given the structure of only one level. Let one level of the expression type containing an integer *or* a pair of type a be:

$$\text{Expr\_proto(a)} = \text{Constant Integer} + \text{Branch}\ (a, a)$$

And then with the type level fixed point combinator: Fix(F) = Fix (F(Fix(F)))[1] we can write the type of our expression tree as:

$$\text{Expr} \;=\; \text{Fix (Expr\_proto)}$$

The Constant case is the base case for the recursion, since it is independent of the parameter α, therefore the fixed point exists. Instances of Expr can now be constructed:

- A single constant of value 5: Fix(Constant(5))
- A branch of two constants: Fix(Branch (Fix(Constant(3)), Fix(Constant(2))))

These constructs again can be mapped into category theory: datatypes are objects and functions between datatypes are morphisms, while parametric data types, like Expr_proto are examples of functors (in fact, endofunctors). We have seen that an algebra for an endofunctor in category C consists of the following three things: the endofunctor $F: C \to C$, an object (type, carrier type) $X \in C$ and the morphism (function) $\alpha: F(X) \to X$. In our example, F can be Expr_proto, X can be Integer and an evaluator that calculates the sums of the values in the tree can be

$$\alpha = \begin{cases} n, & \text{Constant } n \\ x + y, & \text{Branch } (x, y). \end{cases}$$

Now we will formulate the general recursive evaluator using an algebra homomorphism to the initial algebra. Let the carrier type this time be Expr = Fix (Expr_proto) itself. This implies that the corresponding algebra must perform the mapping: α_i: Expr_proto(Expr) → Expr . The function Fix happens to have exactly this signature if we replace F with Expr_proto in Fix: $F\big(\text{Fix}(F)\big) \to \text{Fix}(F)$. This algebra is a special one, called the *initial algebra*, since it does not reduce information, it keeps the whole expression tree structure intact. This observation is known as Lambek's lemma [1], that states that the initial algebra is an isomorphism. The initial object in a category is an object from which there is a unique morphism to every other object in the category. If there is an initial object, it is unique up to isomorphism. In the category of algebras, where objects are algebras, and morphisms are algebra homomorphisms, that preserve structure, the initial algebra is the initial object.

An algebra homomorphism for a given functor F must map the carrier type X and the evaluator α to another such pair: $\phi: (X_1, \alpha_1) \to (X_2, \alpha_2)$ with the additional requirement that $\phi \circ \alpha_1 = \alpha_2 \circ \phi$. On the right hand side the application of ϕ is understood as the image of it under the functor in consideration: $F(\phi)$. For the initial algebra, this leads us to the commuting diagram of Figure 1.

Now, since the inverse of the operation Fix exists, unFix: $\text{Fix}(F) \to F(\text{Fix}(F))$ which just unpacks, reveals one application of the fixed point combinator, it is possible to reverse the corresponding arrow in the diagram and arrive at the generic recursive evaluator for any functor F and algebra α:

---

[1] The syntax for type definitions is: type_constructor_name(argument) = value_constructor_name (defining expression)
The typename and the typeconstructor_name can be the same, since the first names a type, the second names a function and it should be clear from the context which is meant. For the purpose of demonstration, we might assume a Haskell like type system.

$$\phi = \alpha \circ F(\phi) \circ \text{unFix}$$

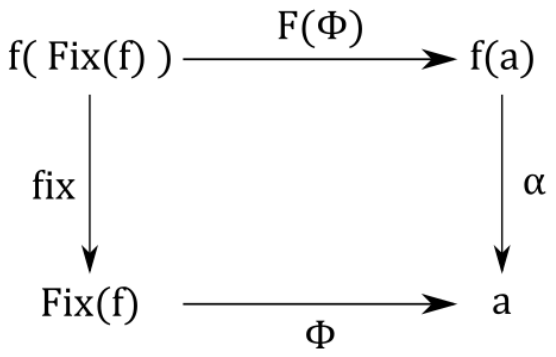

Figure 1. Commuting diagram of the algebra homomorphism from the initial algebra (fix) to another algebra (α)

If we make the dependence on the algebra explicit, we arrive at the construct called a catamorphism [2, 3]:

$$\text{cata}(\text{alg}) = \text{alg} \circ F(\text{cata}(\text{alg})) \circ \text{unFix}$$

This whole construction can be repeated in a dualized setting, and for coalgebras we can derive the anamorphism:

$$\text{ana}(\text{coalg}) = \text{fix} \circ F(\text{ana}(\text{coalg})) \circ \text{coalg}$$

One might continue by generalizing the algebras to depend not only on the actual intermediate values, but also on the previous structure, giving rise to paramorphisms [4]:

$$\text{para}(\text{ralg}) \;=\; \text{ralg} \circ F(\; p(\text{ralg})\;) \circ \text{unFix}$$

where:

$$p(ralg, x) = \big(x, \text{para}(ralg, x)\big)$$

A so-called r-algebra is a function: $ralg\colon F(Fix(F), a) \to a$ which is a generalization of the above introduced simple F-algebra: $alg\colon F(a) \to a$ where the $Fix(F)$ values are the expressions that evaluated to the a values. The dual construct utilizing r-coalgebras in a co-recursion is called an apomorphism.

These and other methods not detailed here are the prime examples of structured recursion schemes, as they clearly separate the traversal of the recursive structure from the actual functionality to be carried out on it. As the most widely used recursive structures are trees (and their special cases, lists) these schemes are frequently used in tree manipulations. In this context algebras deal with the deconstruction of structures and are used in performing bottom-up traversals, while coalgebras are used for constructing structures and top-down traversals. Several theorems can be proven about the product, composability and fusion of these schemes which are important in tree transformations, especially program transformations. In addition, the wide range of recursion schemes has been unified and simpler schemes can be understood as special cases of more general schemes [5].

In the following section we give an example application which implements a simple type checker and resource usage calculator built from solely these primitives.

3. Recursion schemes as program transformers

Modern scientific computing suffers from a hierarchy problem: a very high level domain specific description must be transformed into an efficient low-level program. Unfortunately, during these transformations much information about the problem is lost and the stages where different optimizations or decisions are carried out must be carefully chosen or otherwise the missing information might hinder their applicability. We have investigated the highly simplified problem of transforming an abstract linear algebraic expression into an efficient low level implementation [6]. One of the main tasks was to determine the amount of temporary resources required to avoid dynamic memory allocations. The source language is an embedded domain specific language (EDSL) in Haskell, which contains

- Scalar values, and arithmetic operations on scalars
- Arrays with subexpressions as elements and ArrayViews into user defined external memory.
- Function (lambda) abstraction, application and variable symbols
- Array operations:
    - map: apply a single argument function to each element of an array
    - zip: apply a two-argument function to pairs of elements of two arrays at the corresponding positions resulting in a single array
    - reduce: apply a two-argument function repeatedly to summarize all elements of an array into a single element.

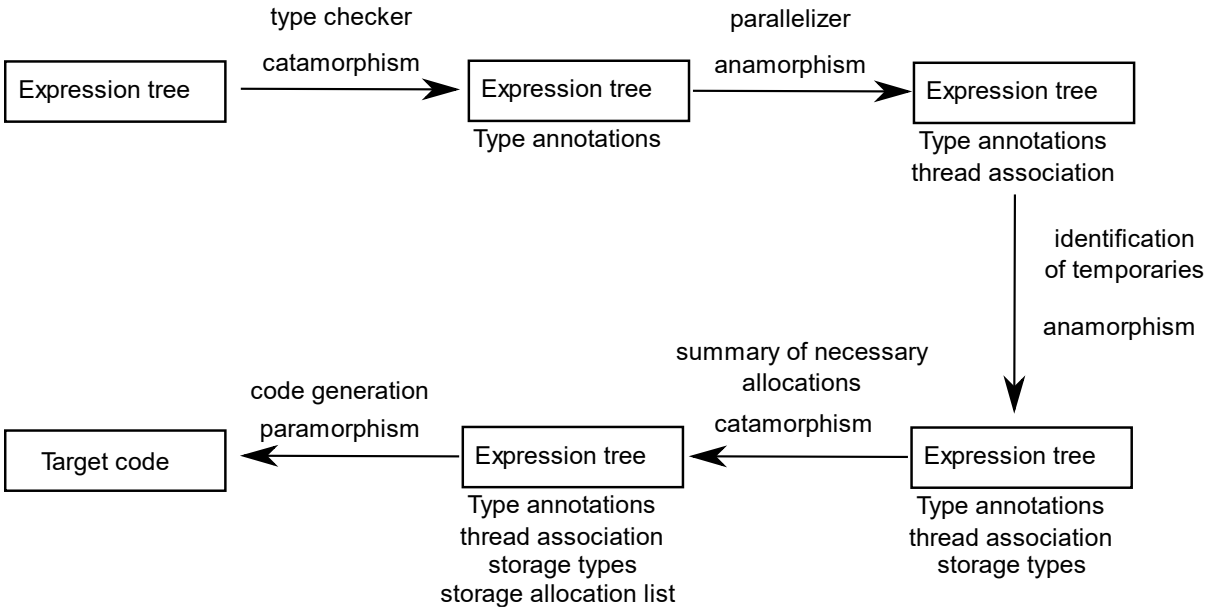

Figure 2. Flowdiagram of the structured recursion scheme based program transformer

With this EDSL, the user can build expressions from the described elements representing a tree (with a fixed point type) which will undergo a series of traversals. Each of these passes

augments the tree with annotations: values which specify some property derived from the structure of the tree as well as previous annotations. Figure 2. shows the sequence of these traversals on the tree: first a catamorphism propagates type information from the leaves (constants, views, variable symbols) upwards. Since the type system is very simple a single bottom-up traversal is sufficient to infer the types of all branches. Next, a top-down traversal determines whether it is feasible to evaluate the given expression in parallel, and if so, distributes a known number of threads between the subexpressions. In the following step the placement of the result of each operations is determined: whether it is evaluated into the memory of a parent expression, or if a new allocation is necessary. Based on the types, the number of threads and the storage targets it is now possible to determine the list of memory blocks to be preallocated. The final step is the bottom-up generation of source code in the target language in the current implementation, C++.

The power of this approach is that the program logic is reduced to the choice of the recursion scheme and the definition of the respective algebra and coalgebra. As all the tree manipulations and traversals are strongly typed, any error is immediately caught by the Haskell type system.

While in recent years compilers for functional programs have begun to use recursion schemes to simplify and optimize code, this area is the subject of active research and whether a production ready, industrial-strength compiler can be implemented entirely using recursion schemes has yet to be shown. Meanwhile, recursion schemes have recently been reinvented in a completely different emerging field: machine learning.

4. Recursion schemes in neural networks

In recent years a particular area of machine learning, that of neural networks, has gained renewed interest. This can partly be attributed to the availability of cheap and unprecedentedly powerful computing hardware in the form of Graphical Processing Units (GPUs) and the collection of enormous annotated datasets which are necessary for the training of such networks.

Briefly, Neural Networks (NNs) are compositions of analytic[2] functions with tunable parameters (weights). Together with automatic differentiation, which is an algorithm for calculating the partial derivatives appearing in the derivative of such function compositions it is possible to state minimization problems for the parameters of the neural networks with respect to a given cost function. In the most widely used supervised learning scenario, a training set is given, which is composed of pairs of input data and desired output (ground truth). The deviation of the network's output from the ground truth, as penalized by the chosen cost function, is then minimized over this training set. With optimized weights, the network can now be applied to new inputs, most likely not from the training set, to perform a desired task. In particular, NNs have excelled at image and speech recognition, classification, prediction and related tasks, in some cases even outperforming humans. The fundamental property which gives rise to their wide applicability is that they are universal approximators: a very wide class of functions can be reasonably well approximated by networks of feasible size.

[2] To be precise, the requirement is much weaker: it is enough to have a method for propagating the error through the function in the automatic differentiation step.

The main challenge of constructing a NN for a particular task lies in the choice and relative ordering of the different component functions (layer types) which are to be composed together. Each of these layers perform a particular kind of transformation on its input, such as an affine transformation, a convolution or a nonlinear transformation.

In a more abstract setting, neural networks can be thought of as functional programs composed of higher-order functions which account for the structure of a particular step, together with simpler functions which represent the specific operation to be carried out in the given part of the structure. Seen in this way, we can recognize the constructs from functional programming, which in turn have their origins in category theory.

A nice example which we can use to illustrate is the case of Recursive Neural Networks (RNNs[3])[7-9]. RNNs find application in tasks which involve repeating structure in their data, especially tree structures, such as in natural language processing or structured image processing. An RNN consists of a function with a set of weights which can be repeatedly applied to a tree structure in a bottom-up fashion, using the same weights on each level, generating the input for the next level above. At each step along the way, the network analyzes wider and wider connections inherent in the data. It was also found that sometimes the branches and leaves might require a different treatment, and thus a separate function with its own weights.

At this point the resemblance should be clear: these RNNs correspond to catamorphisms, and the set of functions applied at each level are exactly the ones which must be specified for an algebra used with the recursion scheme.

RNNs excel at parsing, sentiment analysis, paraphrase detection [10-13] and extended versions also find applications in biochemical structure prediction and analysis [14]. However, the connection to category theory has only recently been recognized, and detailed analysis in this context has not yet been carried out to the best of our knowledge.

5. Summary

In this contribution, we reviewed the concept of algebras and coalgebras in the context of category theory and how it gives rise to the formulation of structured recursion schemes over recursive structures. These schemes are the natural mechanisms for traversing over such structures, and can completely separate the desired logic from the structure of the recursion. The simplest recursion schemes which we have discussed are (algebraic) catamorphisms and (coalgebraic) anamorphisms which represent bottom-up and top-down traversal, respectively. By translating these constructions into the environment of functional programming, one gains highly generic higher-order functions which can be used to manipulate tree structures of programs. We briefly discussed how to assemble a basic compiler simply by composing these schemes, and by annotating the expression tree with intermediate results.

Finally, we have argued that these constructs are more widely applicable than mere program transformations, and have shown that certain layer types used for constructing Neural Networks can also be thought of as realizations of them. Going in the other direction, it may be possible

[3] Unfortunately, the abbreviation RNN can stand for either "Recurrent Neural Network" or "Recursive Neural Network", the former of which is special case of the latter. Here we investigate the more general recursive case.

that some of the other abstractions devised in the context of category theory may also find important applications, as new kinds of layers in Neural Networks, or in other areas of science involving structured information.

6. References

[1] J. Lambek, A fixpoint theorem for complete categories. Mathematische Zeitschrift, 103:151-161, 1968.

[2] M. Fokkinga, E. Meijer, Program Calculation Properties of Continuous Algebras, Technical report, CS-R9104, CWI, Amsterdam, Netherlands, 1991.

[3] E. Meijer, M. Fokkinga, R. Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, In Conf. Functional Programming and Computer Architecture, 124-144, Springer-Verlag, 1991.

[4] L., Meertens, Formal Aspects of Computing 4 413. 1992.

[5] R. Hinze, N. Wu, J. Gibbons, Unifying structured recursion schemes, ACM SIGPLAN Notices - ICFP '13, Vol. 48 Issue 9, pp. 209-220, 2013.

[6] A. Leitereg, Generative programming of massively parallel architectures, BSc. Thesis and "Scientific Students' Associations paper" (*in Hungarian*), Eötvös University, Department of Informatics, 2016.
The program codes and the theses available at: https://github.com/leanil/FParLin and https://github.com/leanil/LambdaGen

[7] A. Sperduti and A. Starita, Supervised neural networks for the classification of structures, IEEE Transactions on Neural Networks, vol. 8, no. 3, pp. 714–735, 1997.

[8] From machine learning to machine reasoning, L. Bottou, Machine Learning archive, Vol. 94 Issue 2, pp. 133-149. 2014.

[9] O. Irsoy, C. Cardie, Deep Recursive Neural Networks for Compositionality in Language, Advances in Neural Information Processing Systems 27, pp. 2096-2104, 2014.

[10] R. Socher, C. C.-Y. Lin, A. Y. Ng, C. D. Manning, Parsing Natural Scenes and Natural Language with Recursive Neural Networks, Proceedings of the 28th International Conference on Machine Learning, 2011.

[11] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, C. D. Manning, Semisupervised recursive autoencoders for predicting sentiment distributions. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 151–161. 2011.

[12] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2013.

[13] R. Socher, E. H. Huang, J. Pennin, C. D. Manning, A. Y. Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In Advances in Neural Information Processing Systems, pages 801–809, 2011.

[14] A. M. Bianucci, A. Micheli, A. Sperduti et al., Application of Cascade Correlation Networks for Structures to Chemistry, Applied Intelligence 12: pp. 117-146. 2000.