Bringing Algebraic Hierarchical Decompositions to Concatenative Functional Languages

ATTILA EGRI-NAGY, Akita International University, Japan

Programming languages tend to evolve over time to use more and more concepts from theoretical computer science. Still, there is a gap between programming and pure mathematics. Not all theoretical results have realized their promising applications. The algebraic decomposition of finite state automata (Krohn-Rhodes Theory) constructs an emulating hierarchical structure from simpler components for any computing device. These decompositions provide ways to understand and control computational processes, but so far the applications were limited to theoretical investigations. Here, we study how to *apply algebraic decompositions to programming languages*. We use recent results on generalizing the algebraic theory to the categorical level (from semigroups to semigroupoids) and work with the special class of concatenative functional programming languages. As a first application of semigroupoid decompositions, we start to design a family of programming languages with an explicit semigroupoid representation.

1 Introduction

Bringing mathematics and programming closer is not a new idea. The development of Fortran, one of the earliest successful languages, started with the question "…can a machine translate a sufficiently rich mathematical language into a sufficiently economical program …?"[4]. However, the emphasis was on expressing scientific computation, not on the mathematical properties of the language itself. Two decades later [3], it was further suggested that programming should stay close to the mathematical notion of a function, opening a line of research leading to today's functional languages. *Category theory*, the study of abstract functions [2], naturally got connected to computer science [6, 26]. In the Categorical Abstract Machine [9] this connection is made explicit.

Not all mathematics relevant to computing found a way to applications. One prominent example is Krohn-Rhodes Theory [22], which is about series-parallel decompositions of automata. The theory brings together several algebraic topics: *What are the basic building blocks of algebraic structures? When are two structures similar (homomorphic), or the same (isomorphic)? What computation can be expressed in a structure, and what cannot?* There are promising applications across many fields [27], but the strictly automata theoretic formulations are not flexible enough for those. One particular decomposition algorithm have been translated to categories[29], but it lacked software implementation. However, recent advances opened up the space of feasible applications [13–15]. Leveraging this new opportunity, we directly apply semigroupoid decompositions to programming languages.

What are the potential benefits of semigroupoid decompositions of programmign languages? They can appear on two levels:

(1) language design, and
(2) problem solving.

First, a semigroupoid representation of a programming language is a mathematical object, so it gives a precise view of the structure of the language and allows comparisons between languages. It gives a finer measure than the binary decision of computational universality (Turing-complete or not).

Second, we would like to distinguish between, and classify different solutions for a fixed problem in the same programming language. Studying multiple solutions can be used for establishing correctness, improving efficiency, communication and teaching purposes or simply for aesthetic reasons. We conjecture that *different algorithmic solutions can be characterized by the different decompositions of the programming language*. For instance, we would normally distinguish between recursion on a sequential collection, or folding (reducing) the same collection, since the realized computational processes are different. On the other hand, we can argue that they are equivalent, as folding can be implemented by recursion. We would like to clarify and decide questions like this on the algebraic level.

Structure of the paper

Section 2 introduces the algebraic hierarchical decompositions of automata, and draws a parallel between the motivating example of the Rubik's Cube analysis and the envisioned language decompositions. Section 3 precisely defines semigroupoids and sketches the recently developed decomposition algorithm, which enables the proposed application of programming language decompositions. Section 4 is a brief survey of languages with the required programming language features. The search homes in on concatenative functional languages. Section 5 contains the preliminary work for programming language decompositions. We close with brief conclusions and outline the next stage of this project.

2 Hierarchical Automata Decompositions

2.1 Decomposing Automata

Finite state automata (FSA) are fundamental tools in computer science [20], used in digital circuit design and compiler construction. Without added structure for language recognition, they are essentially *discrete dynamical systems*. Thus, they can model biological systems and intelligent agents. Algebraically, we define an automaton as a *transformation semigroup* (X, S), where X is a finite set of states and S is a semigroup of total functions $X \to X$. This form allows us to use tools from abstract algebra, most notably *hierarchical decompositions*, which identify basic building blocks and combine them in a network where control information flows in one direction only.

It is a common technique for understanding something by representing it as another object, a more familiar one, or one with a better structure (see Figure 1). In mathematics, we call these homomorphisms, structure-preserving maps. For FSA, we have *emulation*: an emulating automaton can carry out at least as much computation as the emulated one. *Decomposition* is a special form of emulation: we build an automaton from simpler ones with hierarchical (one-way) connections between the components. The seminal result, the Krohn-Rhodes Theorem [22] states that we can always create such decompositions. These are easy to manipulate *cognitive tools* for

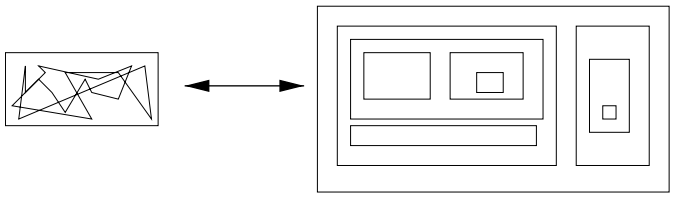

Fig. 1. Schematic explanation of hierarchical decompositions. On the left we have the original system with a complicated structure. For understanding its dynamics, we build another system (on the right) with a hierarchical structure. The decomposition is potentially bigger than the original system. The hierarchical structure *emulates* the original: we can ask how to do an original action in the hierarchical form. The arrow pointing from the decomposition is *interpretation*: we can ask what is the meaning (in the original system) of a hierarchical action.

understanding the structure and behaviour of automata, promising a wide range of applications [27].

2.2 Understanding the Rubik's Cube

The motivation for studying programming languages through algebraic decompositions methodology partially came from a previous successful application of algebraic decompositions. The symmetry group of the Rubik's Cube contains all the possibilities, the total space of arrangements a player can explore. It is natural to use computer algebra to study permutation puzzles [21], and the hierarchical decompositions are particularly good for expressing solution strategies [12]. A decomposition of the cube group corresponds to one particular strategy for solving. The standard beginner method starts with making the cross on the white face of the cube. We locate the white edge faces and move them to their positions. This is a smaller puzzle inside the Rubik's Cube. We only need to think about the four white edge pieces, and ignore the others. Once the cross is done, we do not move them any more, so we are left with a smaller group, the stabilizer group of the cross. We still move temporarily the edges of the cross, but every operation sequence will move them back. This way we decomposed the problem into two. Then, we can iterate this process. Other methods choose different first goal and have accordingly different stabilizer groups. As a more concrete example from [12], the smaller 2×2 cube (Pocket Cube) can be solved by an overly systematic algorithm represented by the wreath product decomposition:

$$S_8 \wr C_3 \wr S_7 \wr C_3 \wr S_6 \wr C_3 \wr S_5 \wr C_3 \wr S_4 \wr C_3 \wr S_3 \wr C_3 \wr C_2 \wr C_3,$$

meaning that first we solve the position of a corner regardless of other parts. The symmetric group S_8 at the top level encodes this. Then we fix the orientation of that corner by working in C_3, the cyclic group of order 3, and iterate the process for the remaining corners. Another strategy is to move all the corners to the correct positions, and deal with their orientations at the same time:

$$S_8 \wr \left(\prod_{i=1}^{7} C_3 \right).$$

Given a particular arrangement, the different decompositions will give different sequences of moves to solve the cube. We are interested in these different ways of understanding the puzzle, not necessarily in the quickest solution.

2.3 Understanding Programming Languages and Programs

Similar to the arrangement space of the Rubik's Cube, the *program space* of a language is the set of all (syntactically valid) programs written in it. Of course, this is a vast space, and unlike the puzzle, it is infinite. It is more useful to think about the *solution space* of a particular computational problem (see also Figure 3). For a given input-output pair we would like to know what is the set of all distinct solutions. The basic task of sorting a sequence is an immediate example, as we have several ways of doing it with different performance characteristics. Then, as a second layer of differences, we have stylistic variations (beyond naming). Sometimes individual programmers can be recognized by the style of code they write, based on the preferences for certain built-in functions and the selections of coding patterns. The differences can be on the level of implementation or of the algorithm specification. Informally we can say that computations can differ by their intermediate results, applied operations, modular structure, or by any combination of these, shaped by language primitives, performance constraints or the programmers' experience level.

How can we detect and characterize these differences in a mathematically precise way? Traditionally, these questions are discussed in *formal semantics* (e.g., [18]). Our working hypotheses is that the algebraic decompositions of the underlying programming language gives that tool. As the Rubik's Cube examples suggests, the solution space spanned by problem solving strategies, different coding styles can be characterized by the space of algebraic decompositions (Figure 2).

3 Semigroupoids

To benefit from the algebraic decompositions, we need to have an algebraic representation. The FSA representation is too low-level for representing programs directly. In particular, programming languages have types to determine what computation is admissible in given contexts. Therefore, we need to introduce typing to automata, i.e., we generalize them to abstraction level of category theory. Fortunately, this generalization has already been done in [15] for simplifying decomposition algorithms.

3.1 Definition

Informally, we can define semigroupoids by saying that they are categories, but with the condition for identity arrows for each object removed. Alternatively, we can start with semigroups, and restrict composition, giving rise to types. To fix the notation, we give a complete definition.

Definition 3.1. A *semigroupoid* consists of *objects*, *arrows* between the objects, and an *associative composition* operation on the arrows. For a semigroupoid $\mathcal{S}$ we use the following notation.
objects The set of objects is denoted by $\mathrm{Ob}(\mathcal{S})$. For objects we use letters σ, τ, ρ. We also call these (formal) *types*.

application domain	mathematical structure	meaning of decompositions
Rubik's Cube	symmetry group	strategies for solutions
program space	semigroupoid	problem solving techniques, coding styles

Fig. 2. The parallel between decompositions of the symmetry group of the Rubik's Cube and the decompositions of the program space of a programming language.

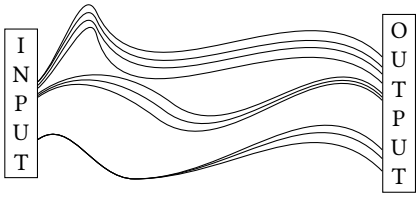

Fig. 3. The algorithmic solution space is the set of computer programs solving a computational problem, i.e. realizing a function described by input-output mappings. This schematic diagram emphasizes the space-like nature. Some solutions are closer to each other, others are more different. As programs are complex structures, we cannot measure distance with a single number, but by algebraic structures.

arrows The total set of arrows is seldom mentioned. Instead, for an object pair (σ, τ) we talk about the set of arrows between those objects, denoted by $\mathcal{S}(\sigma, \tau)$. It can be empty, or contain one or more arrows. An arrow $a \in \mathcal{S}(\sigma, \tau)$ has *domain* $\mathrm{dom}(a) = \sigma$, and *codomain* $\mathrm{cod}(a) = \tau$ (also called *source* and *target*). Alternatively, we can write $a : (\sigma, \tau)$, which reads as 'a has type (σ, τ)'.
composition Two arrows a and b are *composable* if $\mathrm{cod}(a) = \mathrm{dom}(b)$. The composition is denoted by concatenation $(a, b) \mapsto ab$, which type-checks as $a : (\sigma, \tau)$, $b : (\tau, \rho)$, and $ab : (\sigma, \rho)$. Here is a diagram for composition.

$$\sigma \xrightarrow{a} \tau \xrightarrow{b} \rho \qquad \sigma \xrightarrow[ab]{} \rho$$

Note that we compose on the right as in automata theory, not on the left as in category theory, where we would say $b \circ a$ (reading as b after a).
associativity Composition should satisfy the *associativity* condition: $a(bc) = (ab)c$. Consequently, the composite abc is a well-defined arrow of type $(\mathrm{dom}(a), \mathrm{cod}(c))$.

Example 3.2 (Two-type semigroupoid [14])**.** Fig. 4 shows a semigroupoid with two types σ, τ. Morphisms a, b are of type $\sigma^{\circlearrowleft}$, c, d, e are of type $\overrightarrow{\sigma\tau}$, and f is of type $\tau^{\circlearrowleft}$.

An initial semigroupoid exploration have been carried out [14], finding that there can be several type structures for the same abstract semigroupoid (partially defined semigroup multiplication tables), and this underlying type structure (arrow-type semigroupoid) has central importance in studying semigroupoids.

In algebra, we often define structures by giving a set of *generators*, i.e., a set of elements that yield the whole structure when compose them systematically. For semigroupoids, the generators are arrows, forming a directed graph. We produce the generated semigroupoid by taking the transitive closure of this graph.

3.2 Decompositions

The Covering Lemma method [13] creates a two level decomposition of a semigroup based on a surjective morphism. The method has been generalized to semigroupoids [15] to make sure that the second level is also of the same kind of algebraic structure, since applying the algorithm strictly to semigroups does not necessarily give a semigroup on the second level.

The underlying idea of the decomposition algorithm is simple: we create an approximation on the top level, and recover the lost information on the bottom level. Changes on the top level imply changes at the bottom, but not vice-versa. Thus, the structure is hierarchical. The algorithm to create this two level decomposition has three steps.
collapse A surjective morphism $\varphi : \mathcal{S} \to \mathcal{T}$ is the input of the decomposition algorithm. The map φ can take several arrows to a single arrow, hence the name collapsing. The image $\mathcal{T}$ is the top level of the hierarchy.
copy For each arrow in $a \in \mathcal{T}$, we copy its preimage (everything that maps to arrow a). That preimage form a bottom level component, recovering all the forgotten information in φ.
compress Some of the bottom level components are the same (isomorphic), so we can apply compression: store a representative once and the mappings to each copy.

Our goal is to apply this method to programming languages, thus we need to create a semigroupoid representation of a language.

4 Decomposing Languages

In software engineering we do not work with algebraic structures directly. We write programs in a high-level language. Consequently, there is a gap between theory and practice. We aim to bridge this gap by *developing an experimental programming language with a semigroupoid structure.*

First, we have to find a suitable syntax. Figure 5 shows the three possibilities of writing operations and their operands. The postfix, *concatenative*, notation matches the sequential composition notation of semigroupoids (see Figure 4). It is arguable, that changing between these notations is a trivial translation. For instance, balanced parentheses encode the behaviour of a stack. After all, there is one abstract tree structure behind the same arithmetic expression. However, there are other language constructs (e.g., variable declarations), that are not that trivial to translate, and we want to keep the mathematical model naturally close to the semigroupoid structure.

4.1 Relevant Languages

4.1.1 FORTH. The language FORTH [24] was the first concatenative language (though the term 'concatenative' did not exist that time). The purpose was to have a language almost as efficient as

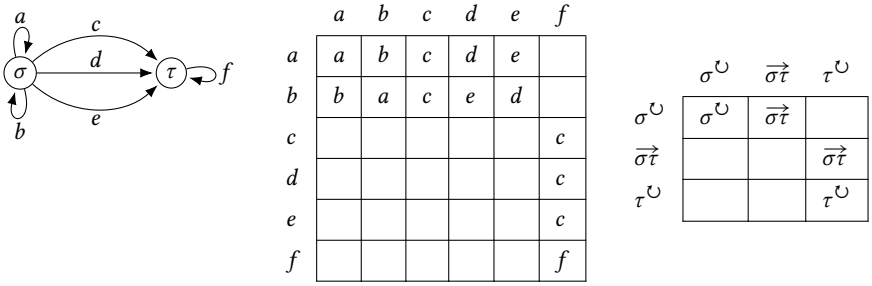

Fig. 4. A semigroupoid with two objects and six arrows. Diagram of objects and arrows on the left, the corresponding composition table in the middle, and the simplified composition table only with the arrow types on the right. The sequence $abaacff$ is typed as the domains match the codomains for each pair in the sequence. However, $bdef$ is not valid d is not composable with e.

infix: mathematics, most languages
`(2*3)*(3+4)`
prefix (Lisp-like):
`(* (* 2 3) (+ 3 4))`
postfix (stack-based):
`2 3 * 3 4 + *`

Fig. 5. The three different ways of writing operators and their operands. Mathematics uses the infix notation. Perhaps due to its familiarity, most programming languages use the same. Languages in the Lisp family use prefix notation. This allows to extend the operators to n-ary, e.g., `(+ 1 2 3)`. It also has a uniform structure for arithmetic an function calls. Finally stack-based languages use postfix notation. This does not even need parentheses, but the arity of operators are fixed.

assembly, but using English words defined by the user. The words denote either machine code programs, or sequences of other words. The compilation process is simple, it follows the concatenation of words. Here is a classic conceptual example from [7].

```
: WASHER  WASH SPIN RINSE SPIN ;
```

The new word `WASHER` is defined as the sequence of four other words – the symbols `:` and `;` are also words.

To ensure communication between the machine code fragments (the words) a stack is used instead of formal parameter lists of functions. To call a function, one has to put input parameters on the stack. A program entirely consists of subroutine calls, which is called *threaded code*. This allows very short program representations, since the code is just a sequence of word identifiers referring to machine code fragments. The execution speed is faster than interpreted languages [8], but not as fast as register allocation optimized native code.

4.1.2 Lisp. Although it is a language with prefix operators, we need to mention Lisp for two reasons. First, the idea of *quoting*, i.e., treating programs as data [23]. Second, its 'axiomatic' nature: building the language from a few primitives[17].

Quoting allows us to treat a piece of code, represented as a list, as a data structure. Here are two expressions from a modern Lisp, Clojure [19].

```
(count (map inc [1 2 3 4 5]))
(count '(map inc [1 2 3 4 5])
```

The first line evaluates to 5, after mapping the increment function over the collection. The second results in 3, as the code for incrementing numbers has three parts. Code is data, thus it can be modified by the program itself. Originally, this was thought to be an important ingredient for artificial intelligence.

Here, we are more interested in the idea of defining a languages by a few operations. These are the 'generators' of the language. For example,

```
last == (comp first reverse)
```

therefore we do not need to define an operator for getting the last element of a list, if we can reverse the list and take the first element.

4.1.3 Joy. The language Joy[28] uses the same stack based operation as FORTH, but it has a radically different design philosophy, being purely functional. It does indeed come from philosophy [1]. Unfortunately, most of the material written about Joy remained unpublished (available only by individuals mirroring the original homepage), and the implementations are not usable any more (due to old C standard). However, the key ideas are amply described in the draft papers.

The concatenation of two programs denotes the composition of the functions defined by the two programs. For example, the following program

```
[1 2 3 4]  [dup *]  map
```

produces the list `[1 4 9 16]` by pushing a list to the stack, then another one containing code and then applying the higher order function `map`. Essentially, putting code into a list quotes that code. Programs are built of three different types of components.

- **literals** values that push themselves to the stack, e.g., `0`,1, `"mango"`
- **operators** n-ary operators that take n values from the stack and push some result, e.g., `+`, `swap`
- **combinators** require quoted programs on the stack, e.g., `apply`

Why a stack? Every function has a single argument, the stack. Alternatively, we could consider the whole memory content of a computer as the state. Any operation would be a function of a global state to another global state. Function composition would revert

back to the 'typeless' semigroup multiplication. This shows that computation should be 'localized' to some extent. As another viewpoint, we can say that computation is resource-aware in the sense that variables can used only once, as in linear logic [5]. For example, the squaring function need to duplicate x to be able compute x^2.

4.1.4 Factor. A modern and full-featured concatenative language is Factor [25]. Among other features, it adds variable declarations for efficiency purposes. These are beyond the scope of this paper, but it is worth noting that concatenative languages can have practical applications.

5 A minimal semigroupoid language

It is easy to implement a concatenative programming language. Parsing is the same as tokenizing, and the whole language can be implemented by a lookup table from tokens to stack-to-stack unary functions. We implemented a minimal language con-cat [10] for the constructions below. We differ from Joy in that we need to consider the list symbols [and] as separate tokens. Reading [pushes an *open* list to the stack, which consumes (quotes) everything that comes after that until the closing], which closes the list, leaving it at the top of the stack.

5.1 Typing rules

Objects in a semigroupoid (category) can have many interpretations. The simplest way to see them as 'pegs' to put the morphisms on. This view is compatible with considering them as types.

In a stack-based language, the inputs of an operator are on the stack, so their numbers and data types determine the applicability of the operator. This is often described in a *stack effect*, e.g., for $f(x, y) = z$:

```
(x y -- z)
```

We will use the stack state as types, the objects of the semigroupoid.

5.2 Generating sets – Partial Order of Sublanguages

We will design and implement a succession of increasingly more capable concatenative languages, and systematically study their decompositions, i.e., adding the language elements one by one, and see what they generate.

Example 5.1. The `swap` operator has the effect of exchanging the top of the stack and the element below it. The code `swap swap` will leave the stack as it was before, so it is the same as identity `id` operator. Thus `swap` generates the semigroupoid with the composition table below.

	`id`	`swap`
`id`	`id`	`swap`
`swap`	`swap`	`id`

This is $\mathbb{Z}_2$, the cyclic group of order 2, essentially a modulo-2 counter.

5.3 Dealing with infinity

The Krohn-Rhodes Theory is primarily a result for *finite* semigroups. In contrast, the computation space of a programming language is infinite. There are two strategies to deal with this discrepancy. We can have a set of atomic programs that generate only a finite program space. Or generalize the decomposition algorithms to the infinite case. For a specific decomposition algorithm this is possible [16]. Also, the Covering Lemma method does not require finiteness. However, there is no software implementation for the infinite case. Therefore, the decomposition needs to be done 'manually'. It is expected that the systematic analysis of finite sublanguages will give enough understanding for carrying out a proof for the infinite case.

There are several places where infinity comes into a programming language.

- **numbers** Digital computers are discrete devices, but integers $\mathbb{Z}$, or even just the natural numbers $\mathbb{N}$ are infinite sets.
- **stack** We can push some data item to the stack, and then another one. There is no inherent limit in pushing, thus the stack appears to be infinite.
- **collections** Aggregate data types do not have size limits. We can always add an element to a list, or put a collection into an other one (nesting).
- **quoting** In principle, we can quote any expression, even an already quoted one. When quoting is represented as a list, repeated quoting is the same as a nested data structure.

Numbers are easy to deal with. We can apply finite arithmetic, modular calculations. This is a natural choice for computers. An 8-bit processor can only work with values from 0 to 255. On the algebra side, the rings representing finite arithmetic, called quotient, or residue class rings, has been decomposed hierarchically [11]. In other words, the language generated by `0,1,...,n-1,+,*` has been studied mathematically.

The finiteness of the stack is a fact for all hardware implementations. It can be of flexible size, e.g., by letting it grow backwards at the top of the heap, but then heap size makes it finite. Handling the stack overflow error in the semigroupoid could be problematic. Do we block pushing to the stack, or allow the bottom of the stack erased? We can have a boolean flag included in the stack data structure, and thus in the type definition in the semigroupoid. However, finite stack does not imply finite programming language. Also, artificial restriction on the language, e.g., limiting the program size to fixed number of instructions, can easily violate associativity. Without associativity, the whole algebraic framework is inapplicable. Thus, we need to consider finite homomorphic images of infinite languages.

5.4 Binary Addition Language

Here, we construct a very minimal language, only for binary addition. The stack size is limited to 2. There are two number literals `0` and `1`, and one numerical operator `+` for addition. Starting from an empty stack, the program `1 0 +` would push `1`, then `0` to the stack, then `+` pop those values, and push their sum `1` to the stack. The program `0 1 0` is not valid, as we have a stack overflow. Also, the program `+` is not valid due to a stack underflow error. Figure 6 shows a set of generator arrows for this limited language. It also shows how compression can be applied to the semigroupoid.

The corresponding semigroupoid is infinite. It is easy to see that once the literal `0` is pushed to the stack, the program `0 +` will act as

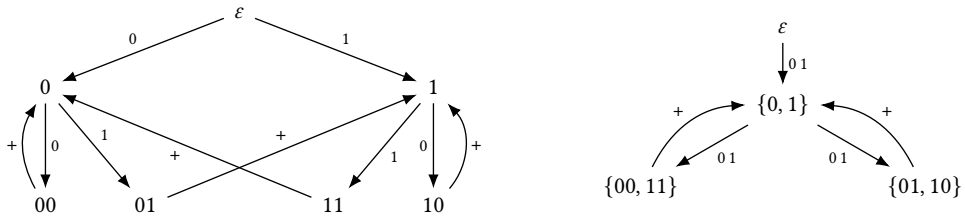

Fig. 6. On the left, generator arrows for binary addition stack size 2 language. The types (objects) are the possible states of the stack. For example, 01 means that first 0, then 1 was pushed to the stack. The empty stack is represented by ε. The type ε does not have an identity, so we really need a semigroupoid to model this language. The generated semigroupoid is infinite. The missing arrows can be found by the transitive closure of the underlying graph. On the left, the compressed form.

an identity on that stack state. We can repeat `0 +` infinitely many times. Similar loops of the generator arrows can be contracted to other types (except the empty stack), i.e., we can find directed paths starting from a type (a stack-state) that returns back to the same type.

How can we turn this structure into a finite semigroupoid? For the Covering Lemma method, we need to give a surjective morphism. We can try to make the first component to be a finite semigroupoid. Thus, the decomposition divides the infinite into two parts, one finite, and another infinite. We might implement the identity on the stack state of containing a single `0`, but we could say that we are not interested in the details, just the fact there is an identity operation on that type. On the top level of the hierarchy, we record the existence of the identity arrow, and on the second level we include all the infinite ways to realize that arrow.

One possible finite image is the *arrow-type semigroupoid* from [14]. That surjective morphism sends the set of arrows between two types to a single arrow. In a way, this gives a primitive type-checker. The second level of the decomposition then would comprise of all these lost details, namely all the possible programs that take the source type to the target type.

Another way to have a finite representation is to consider the transformations of the global state space. In Figure 6, we use the same label for several arrows, though in the semigroupoid those arrows are different, and the connection is external (made on the semantic level). If we take the stack states in a fixed order ε, 0, 1, 00, 01, 11, 10 (just top to bottom, left to right enumeration in the diagram), and encode by their index,

ε	0	1	00	01	11	10
0	1	2	3	4	5	6

then + defines the mappings $3 \mapsto 1$, $4 \mapsto 2$, $5 \mapsto 1$, and $6 \mapsto 2$. For the remaining states it is undefined, since the + operator requires two number literals on the stack. We get the partial transformations $+[\_, \_, \_, 1, 2, 1, 2]$, $\mathtt{0}[1, 3, 6, \_, \_, \_, \_]$, $1[2, 4, 5\_, \_, \_, \_,]$. This gives a partial transformation representation, which can easily be translated into a transformation representation (by adding an extra sink state to represent the undefined state). They generate a semigroup with 21 elements. When doing a complete (holonomy) decomposition, we get $\mathbb{Z}_2$ among the components, coming from the binary addition. It is always possible to have such a partial transformation representation, but the state space grows fast.

6 Conclusion

We described a way to bring the algebraic hierarchical decompositions to programming languages. This is the first step in the long-term project of building a programming language, piece by piece, maintaining an explicit algebraic understanding. This plan requires modifications in automata theory (generalizing the algebraic representation from semigroups to semigroupoids) and restricting to the special class of concatenative languages to match the linear syntax of automata input words. The main ideas can be summarized by the following points.

- Concatenative languages have the same syntax as semigroupoid composition.
- Stack effects define types (objects of the semigroupoid), giving categorical semantics.
- We can manage infinity by mapping an infinite structure down to a finite one.

These enable us to feed a programming language into the decomposition algorithms, giving a new type of mathematical understanding of computer programs. The next task is the define a concatenative language in a *modular* way. Since a full featured language is too big to decompose, we need to add literals, operators and combinators one by one as generators. The source code for this project is available at [10].

Acknowledgments

This project was funded in part by the Kakenhi grant 22K00015 by the Japan Society for the Promotion of Science (JSPS), titled 'On progressing human understanding in the shadow of superhuman deep learning artificial intelligence entities' (Grant-in-Aid for Scientific Research type C, https://kaken.nii.ac.jp/grant/KAKENHI-PROJECT-22K00015/).

References

[1] Stevan Apter. 2004. A Conversation with Manfred von Thun. *VECTOR, Journal of the British APL Association* 20, 3 (2004). https://archive.vector.org.uk/art10000350.
[2] S. Awodey. 2010. *Category Theory.* OUP Oxford.
[3] John Backus. 1978. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM* 21, 8 (Aug. 1978), 613–641. doi:10.1145/359576.359579

[4] John Backus. 1978. The history of Fortran I, II, and III. *ACM Sigplan Notices* 13, 8 (1978), 165–180.

[5] Henry G Baker. 1994. Linear logic and permutation stacks-the Forth shall be first. *ACM Sigarch Computer Architecture News* 22, 1 (1994), 34–43.

[6] M. Barr and C. Wells. 1995. *Category Theory for Computing Science.* Number v. 1 in Prentice-Hall international series in computer science. Prentice Hall.

[7] Leo Brodie. 1987. *Starting Forth.* Prentice-Hall, Inc.

[8] L. Brodie. 2004. *Thinking Forth.* Punchy Publishing. https://www.forth.com/forth-books/

[9] G. Cousineau, P.-L. Curien, and M. Mauny. 1987. The categorical abstract machine. *Science of Computer Programming* 8, 2 (1987), 173–202. doi:10.1016/0167-6423(87)90020-7

[10] Attila Egri-Nagy. 2025. *CONCAT functional concatenative programming language with an explicit semigroupoid representations.* codeberg.org/egri-nagy/con-cat.

[11] A. Egri-Nagy and C. L. Nehaniv. 2007. Finite Residue Class Rings of Integers Modulo n from the Viewpoint of Global Semigroup Theory. In *Proceedings of International Conference on Semigroups and Languages 2005 In Honour of the 65th birthday of Donald B. McAlister*. World Scientific Press, 66–83.

[12] Attila Egri-Nagy and Chrystopher L. Nehaniv. 2015. Computational Understanding and Manipulation of Symmetries. In *Artificial Life and Computational Intelligence*, Stephan K. Chalup, Alan D. Blair, and Marcus Randall (Eds.). Springer International Publishing, Cham, 17–30.

[13] Attila Egri-Nagy and Chrystopher L. Nehaniv. 2024. From Relation to Emulation and Interpretation: Computer Algebra Implementation of the Covering Lemma for Finite Transformation Semigroups. In *Implementation and Application of Automata*, Szilárd Zsolt Fazekas (Ed.). Springer Nature Switzerland, Cham, 138–152. doi:10.1007/978-3-031-71112-1_10

[14] Attila Egri-Nagy and Chrystopher L. Nehaniv. 2025. Computational Exploration of Finite Semigroupoids. arXiv:2509.00837 [cs.FL] https://arxiv.org/abs/2509.00837 miniKanren 2025 Workshop at ICPFL/SPLASH'25.

[15] Attila Egri-Nagy and Chrystopher L. Nehaniv. 2025. Representation Independent Decompositions of Computation. arXiv:2504.04660 [math.GR] https://arxiv.org/abs/2504.04660

[16] Gillian Z. Elston and Chrystopher L. Nehaniv. 2002. Holonomy Embedding of Arbitrary Stable Semigroups. *International Journal of Algebra and Computation* 12, 6 (2002), 791–810.

[17] Paul Graham. 2002. The Roots of Lisp. https://www.paulgraham.com/rootsoflisp.html.

[18] C.A. Gunter. 1992. *Semantics of Programming Languages: Structures and Techniques.* MIT Press.

[19] Rich Hickey. 2020. A History of Clojure. *Proc. ACM Program. Lang.* 4, HOPL, Article 71 (June 2020), 46 pages. doi:10.1145/3386321

[20] John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

[21] D. Joyner. 2008. *Adventures in Group Theory: Rubik's Cube, Merlin's Machine, and Other Mathematical Toys.* Johns Hopkins University Press.

[22] Kenneth Krohn and John Rhodes. 1965. Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines. *Trans. Amer. Math. Soc.* 116 (April 1965), 450–464.

[23] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. doi:10.1145/367177.367199

[24] Charles H Moore. 1974. FORTH: a new way to program a mini computer. *Astronomy and Astrophysics Supplement, Vol. 15, p. 497* 15 (1974), 497.

[25] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. 2010. Factor: A dynamic stack-based programming language. *Acm Sigplan Notices* 45, 12 (2010), 43–58.

[26] B.C. Pierce. 1991. *Basic Category Theory for Computer Scientists.* MIT Press.

[27] John Rhodes. 2009. *Applications of Automata Theory and Algebra via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games.* World Scientific Press. Foreword by Morris W. Hirsch, edited by Chrystopher L. Nehaniv (Original version: UC Berkeley, Mathematics Library, 1971).

[28] Manfred Von Thun and Reuben Thomas. 2001. Joy: Forth's functional cousin. In *Proceedings of the 17th EuroForth Conference.*

[29] Charles Wells. 1980. A Krohn-Rhodes Theorem for categories. *Journal of Algebra* 64 (1980), 37–45. doi:10.1016/0021-8693(80)90130-1