FINITE-STATE REPRESENTATION OF GEOMETRIC TRANSDUCTIONS

DRAFT

Emmanuel Roche
Clover.AI
roche@cloverai.net

June 8, 2023

ABSTRACT

We introduce a formalism describing certain transformations of sets of labeled segments. These transformations are mappings expressed in terms of geometric relationships between these segments. These transformations, which we call geometric transductions, can be represented with finite-state transducers or bimachines. Such geometric transductions are the foundation of a new family of natural language parsing algorithms.

Keywords Finite-State Machine · Natural Language Parsing

1 Introduction

In the following work, we will revisit the old question of how to represent the syntax of a natural language, as linguists discuss it in an attempt to describe it formally. This question can be traced back to the earliest days of Natural Language Processing. However, this work differs from the original attempts in important ways. Earlier work usually follows a programming language compiler view which three main points can characterize:

1. **(Language Family Specification)** One must identify and define the family of formal languages best adapted to the task. In the case of programming languages, this could be LR(1) languages. In the case of natural language, it could be context-free grammars or tree-adjoining grammars, among many others. This first foundational point had been a key focus of natural language processing until deep learning emerged.
2. **(Rule Set)** One needs to define the set of rules that, together with the formal language family, describes the formal language completely. In the case of a programming language, this could be a Yacc file or any kind of syntax declaration file. In the case of natural language, this could be a set of rules, or any kind of declarative format, itself defined as a programming language. Once the type of rule is defined, this second point, the writing of rules, is meant to be the domain of linguists who would actualize a particular language, English, for instance, from the abstract grammar to the grammar specific to that language. In some instances, these grammar rules can be either fully learned or adjusted through training.
3. **(Parsing Algorithm)** Once the first two points are settled, a parsing algorithm becomes necessary. This algorithm will take the rules of the previous point and an input statement or sentence and produce some kind of formal representation of that sentence. In the case of programming languages, this could be a parse tree or compiled code. In the case of natural language, this takes a great variety of forms, including syntactic trees and dependency representations.

Within that paradigm, the most important theoretical work fits within the first point, which tries to address two constraints: make the family of languages as close as possible to natural language (powerful enough to describe it) with the additional requirements that rules of the second point can be expressed as naturally as possible by linguists. Thereby, the second point fits within the realm of linguistics and linguists whose task is to embed a particular natural language, English, for instance. Finally, the third point, the parsing algorithm, becomes a pure algorithm problem where complexity and efficiency are addressed.

Taking as a starting point the linguistic theory of Lexicon-Grammar, as described, for instance, in the trilogy from Maurice Gross (Gross, 1968, 1975, 1986), we observed that how linguists talk about syntax was far removed from what formal framework allowed them to express. The approach described here can also be viewed as an attempt to bridge that gap.

This work may feel alien to the current partitioners of NLP where, what is now called symbolic processing, has been overshadowed, maybe "eradicated" is a better word, by advances in deep learning. In fact, the latter has been able to produce results in such tasks as automatic translations so far beyond what earlier symbolic processing could achieve that the latter has, to use a somewhat militaristic metaphor, been seemingly obliterated by deep learning. Within that view, it is not only symbolic processing that has become obsolete but the entire field of descriptive linguistics as well, as far as syntax is concerned. Transformers seem content to distill the entire body of necessary linguistic knowledge to the following statement: "Words depend on context (the transformer layer), and fuzzy hierarchy is important (the fact that there are multiple layers)." In particular, concepts such as verbs, nouns, and part-of-speech have become irrelevant. That view, often implicit and sometimes explicit, however empirically successful it has been, leaves many questions unanswered. For instance:

- Large Language Models (LLMs) combine knowledge of language with knowledge of the world. For instance, they cannot, so far at least, produce a model of language acquired by a six-year-old without ingesting such large corpora as the entire Wikipedia. Is there a way to better separate what language is from what is knowledge and produce a model of language much smaller than successful LLMs? By much smaller, we mean several orders of magnitude smaller.
- Is there such a thing as syntax (or grammar, we use the terms interchangeably)? Can the type of knowledge described by linguists be formally described into a system powerful enough to analyze any sentence?

The vast successes of deep learning have also pushed some to posit these successes come from the fact that deep learning is much more biologically plausible. It should be noted, however, that symbolic processing is also biologically plausible, as demonstrated by the existence of language itself. In fact, language is, in at least some sense of term, symbolic, and there is no reason that this symbolicity is not itself part of the plausible biologic processing of language.

There is no reason why symbolic processing and deep learning are exclusive, as a better understanding of symbolic structure could lead to better architectures.

Succinctly, our initial steps will be the following:

- Initial description of $\gamma-$spaces that represent a state of parsing and of $\gamma-$transformations that describe a transition between these states (this present document).
- Extension of $\gamma-$spaces to distinguish certainty and uncertainty in parsing and description of how parsing could be viewed as a propagation of certainty
- Description of how this applies to particular sentences covering a wide range of linguistic phenomena
- The relationship between general syntax and syntax specific to particular domains (i.e. local grammars)
- Experimental validation on a substantial part of English.

2 Prior Work

The use of formal languages to describe natural language can be traced back to, and probably defines, the starting point of computational linguistics, or natural language processing. Following Chomsky's original rule rewriting system, designing formal languages to describe linguistic information has a long tradition. Early famous examples include Winograd's PROGRAMMAR (Winograd, 1972) , PART-II (Shieber, 1984) , Tree-Adjoining Grammars (Joshi and Schabes, 1992) or INTEX (Silberztein, 1993) among many others.

The evolution of these formalisms also required the design of parsing algorithm. We cannot review the history of parsing here (see Jurafsky and Martin (2019) for an overview of dependency parsers for instance). We will only mention that applying the geometric transductions described here is related to earlier parsing techniques relying on finite-state transducers and bimachines (Roche, 1993; Yli-Jyra, 2005; Gamallo and Garcia, 2018). Finite-State Transducers have been used in natural language processing beyond parsing with packages like OpenFST (Allauzen et al., 2007) using a large variety of algorithms designed for natural language processing (Mohri, 2000).

3 Introduction to $\gamma-$spaces and $\gamma-$transformations

Figure 1 below shows set of labeled segments. In this figure, only the X axis is meaningful; how these rectangles are arranged along the Y axis is only provided for visual clarity. This set is a set of triples (s, l, e) where s and e are integers such that $s < e$ and the label l is an element of some finite alphabet Σ. We call these triples "labeled segments", or just "segments". For instance, the segment of the lowest left corner of is represented by the triple $(0, a, 2)$ where 0 is the starting position, 2 the ending position and a the label. The segment just above it is the triple $(1, e, 3)$. We call this type of set a one-dimensional $\gamma-$ space (γ for 'geometric') and denote it $1 - \gamma-$space. Because we will only discuss one-dimensional spaces in this document, we will simply denote it $\gamma-$space.

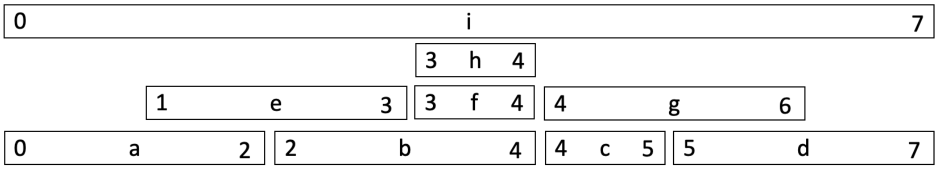


Figure 1: S_1: A $\gamma-$space example

We are interested certain kinds of mappings between $\gamma-$spaces. This type of mapping, which we will call $\gamma-$transformations, is defined by a set of rules in which each rule is a pair pattern/transformation. The pattern describes the condition under which a transformation is applied.

The following is an example of a pattern expressed in plain English:

A segment labelled 'f' (segment 'f' for short) aligned with a segment 'h', left aligned with a segment 'b' which should precede a segment 'c' and follow a segment 'a'

We express formally the same condition with the following pattern expression p_1, using prefix binary operators (we use the vertical bar | to separate the two arguments of the operator):

$$p_1 = (\cdot\, a \,|\, (\cdot\, (>\sim \,|\, b\,|(||\, f\,|h\,))\,|\, c))$$

The three operators used in this expression are:

- $\cdot$: *next-to* or *concatenation*: indicates that the starting range of the second operand is the same as the ending range of the first one.
- $>\sim|$: *strictily-greater-right-aligned* : indicates that the second operand should have a starting point strictly larger than the starting point of the first operand and that the ending points of both operands should be equal.
- $||$: *aligned*: indicates that both operands share the same starting and ending points.

In our example S_1,

$$(||\, f \,|\, h\,)$$

indicates a segment 'f' aligned with a segment 'h'. We use the prefix notation for the operator (||, 'aligned', with the vertical bar as an argument separator. 'Aligned' means that both segments share the same starting and ending positions. Segments 'f' and 'h' of S_1 of Figure 1 are clearly aligned in that sense.

Then,

$$(>\sim \,|\, b\,|(||\, f\,|h\,))$$

indicates that segment 'f' should be 'right-aligned' with 'b' and strictly smaller. This means that the ending point of 'f' should be the same as the ending point of 'b' and that the starting point of 'f' should be strictly larger than the starting point of 'b'. When an operand is not a segment but a complex operand, the condition is applied on the first segment (recursively) of that operand. In other words, in the case above, as far as the $>\sim|$ is concerned, only segments 'b' and 'f' are considered and 'h' can be ignored ('h' is not ignored in the previous $(||\, f \,|\, h\,)$ operation). Again, S_1 matches this condition.

then

$$(\cdot\,(>\sim \mid\ b\,|(||\,f\,|h\,))\mid c)$$

indicates that a segment ’c’ needs to be following immediately the segment ’b’.

Finally,

$$(\cdot\, a \mid (\cdot\,(>\sim \mid\ b\,|(||\,f\,|h\,))\mid c))$$

indicates that ’b’ comes just after ’a’.

The next-to operation, which we also call the concatenation operation, is the default operation. It can be represented with the operator $\cdot$ or implied by just putting the two operands next to each other. For instance, $(\cdot\ a|b)$ can be written ab. In addition, because it is associative, parentheses can be omitted. p_1 can then be simplified into:

$$p_1 = a\,(>\sim \mid\ b\,|(||\,f\,|h\,))\,c$$

Since S_1 verifies this condition, we can say that the pattern matching of p_1 is successful or that *"p_1 matches S_1"* or $S_1 =\sim p_1$ following programming language notations such as Perl.

In the case where there is no ambiguity, we will also use the infix notation:

$$p_1 = a\,(\ b >\sim |\,(\,f\,||h\,))\,c$$

as it is often easier on the eye.

Once a pattern is defined, we can define a set of segment creations and deletions based on that pattern. For instance, the following transformation expression:

$$t_1 = a\,(+j\,(>\sim |\,b\,|\,(||\,f\,|\,(-h)\,)\,)\,c)$$

indicates that, in the case of the pattern p_1, a new segment ’j’, spanning ’b’ and ’c’, needs to be created and that the segment ’h’ needs to be deleted. The result of that transformation is the space S_2 of Figure 2 .

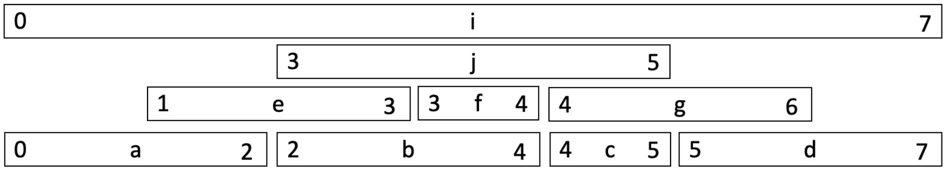


Figure 2: $S_2 = \widetilde{t_1}(S_1)$

We will see that we can apply a set of such rules efficiently by representing them with Finite-State Transducers (FST), thereby allowing us to scale the number of rules being applies at once. In fact, million of rules can be represented by a single FST of reasonable size. We call the FST representing such transformation a $\gamma-$ transducer ($\gamma-$FST). We will only consider here one-dimensional geometric constraints (relationships between segments), but it is also possible to consider higher dimensional spaces; for instance considering the geometric relations between rectangles in two-dimensional spaces, and so forth.

In addition to $\gamma-$ transformations, we will also consider the case in which a given transformation is applied sequentially on the same space. For instance, for a $\gamma-$FST T, we will consider $T \circ T\ ...\ T(\gamma) = T^n(\gamma)$. In particular, we are interested in the fixed points of such transformations.

As mentioned in the introduction, this formalism is related to parsing of natural language sentences. We will allude to this relation at the end of this document.

4 Definitions

4.1 Notations

For convenience, we introduce the notations before their definitions:

Notation	Meaning
Σ	finite alphabet
$\mathbb{L}(\Sigma) \subset \mathbb{Z} \times \Sigma \times \mathbb{Z}$	Set of all labeled segments on Σ
$S_1, S_i \subset \mathbb{L}(\Sigma)$	$\gamma-$spaces
$\mathbb{G}(\Sigma) \subset \mathcal{P}(\mathbb{L}(\Sigma))$	set of $\gamma-$spaces on an alphabet Σ
$L_1(\Sigma)$	formal language of pattern matching expressions
$L_2(\Sigma) \supset L_1(\Sigma)$	formal language of transformation expressions
$t_1, t_2 \in L_2(\Sigma)$	transformation expressions, elements of $L_2(\Sigma)$
$\mathbb{T}(\Sigma) \subset \mathbb{G}(\Sigma)^{\mathbb{G}(\Sigma)}$	The set of $\gamma-$space transformations, also called geometric transductions, on $\mathbb{G}(\Sigma)$
$\widetilde{t_1}, \widetilde{t_2} \in \mathbb{T}(\Sigma)$	$\gamma-$space transformations on $\mathbb{G}(\Sigma)$
T_1, T_2 s.t. $\widetilde{T_1}, \widetilde{T_2} \in \mathbb{T}(\Sigma)$	finite-state transducers representing a $\gamma-$space transformation

4.2 Labeled Segment

Following formal languages conventions, we will consider a finite alphabet Σ. Its elements will be called letters or labels.

Definition 1 (Labeled Segment) *Given a finite alphabet Σ, a* labeled segment *is a triple (s, l, e) where s and e are integers such that $s < e$ and $l \in \Sigma$. l is called the label of that segment and s and e the starting and ending positions of that segment respectively. (s, e) is called the range of that segment.*

In Figure 1, $(3, b, 4)$ and $(4, g, 6)$ are two examples of such segments.

We will use the following notation: given a segment $\sigma = (\alpha, \beta, \delta)$, we write the starting position of that segment $\sigma.s = \alpha$, the ending position $\sigma.e = \delta$ and the label $\sigma.l = \beta$.

We denote by $\mathbb{L}(\Sigma)$ the set of all labeled segments on Σ. $\mathbb{L}(\Sigma)$ is strict a subset of $\mathbb{Z} \times \Sigma \times \mathbb{Z}$ ($\mathbb{Z}$ being the set of integers) because of the constraint of ending positions being strictly larger than starting positions.

4.3 $\gamma-$ Space

Definition 2 ($\gamma-$ Space) *Given an alphabet Σ, a $\gamma-$ space is a finite set of labeled intervals on that alphabet.*

A gamma space is therefore any finite subset of $\mathbb{L}(\Sigma)$. $\mathbb{G}(\Sigma)$ denotes the set of $\gamma-$spaces on Σ. We have $\mathbb{G}(\Sigma) \subset \mathcal{P}(\mathbb{L}(\Sigma))$ where $\mathcal{P}(\mathbb{L}(\Sigma))$ is the powerset of $\mathbb{L}(\Sigma)$.

For instance, Figure 1 represents the following $\gamma-$space:

$$S_1 = \{(0, a, 2), (2, b, 4), (4, c, 5), (5, d, 7), (1, e, 3), (3, f, 4), (4, g, 6), (3, h, 4), (0, i, 7)\}$$

It is important to note that while all segments of this particular space have different labels, there is no requirent that this be the case. This is done solely for convenience reasons (to make it easier to refer to a particular segment within the present discussion).

4.4 Relative Position Operators

Given a $\gamma-$space and two labelled segments σ_1 and σ_2 in that space, we will now define a set of relative position operators. Each operator will state some information about how σ_2 is positioned with respect to σ_1. We call this set of operators OP. Each operator is therefore a mapping between $\mathbb{L}(\Sigma) \times \mathbb{L}(\Sigma)$ and $\{0, 1\}$. This set is defined in the table below:

Operators (OP)	Name	Definition (true iff)
$\cdot$	**next-to**	$\sigma_1.e = \sigma_2.s$
$\cdot*$	**following**	$\sigma_1.e \leq \sigma_2.s$
$\Vert$	**aligned**	$\sigma_1.s = \sigma_2.s$ and $\sigma_1.e = \sigma_2.e$
$>$	**strictly-larger**	$\sigma_1.s \leq \sigma_2.s$ and $\sigma_1.e \geq \sigma_2.e$ and ($\sigma_1.s < \sigma_2.s$ or $\sigma_1.e > \sigma_2.e$)
$>\sim\mid$	**strictly-larger-right-aligned**	$\sigma_1.s < \sigma_2.s$ and $\sigma_1.e = \sigma_2.e$
$>\mid\sim$	**strictly-larger-left-aligned**	$\sigma_1.s = \sigma_2.s$ and $\sigma_1.e > \sigma_2.e$
$<$	**strictly-shorter**	$\sigma_1.s \geq \sigma_2.s$ and $\sigma_1.e \leq \sigma_2.e$ and ($\sigma_1.s > \sigma_2.s$ or $\sigma_1.e < \sigma_2.e$)
$<\sim\mid$	**strictly-shorter-right-aligned**	$\sigma_1.s > \sigma_2.s$ and $\sigma_1.e = \sigma_2.e$
$<\mid\sim$	**strictly-shorter-left-aligned**	$\sigma_1.s = \sigma_2.e$ and $\sigma_1.e < \sigma_2.e$
$\geq$	**larger**	$\sigma_1.s \leq \sigma_2.s$ and $\sigma_1.e \geq \sigma_2.e$
$\geq\sim\mid$	**larger-right-aligned**	$\sigma_1.s \leq \sigma_2.s$ and $\sigma_1.e = \sigma_2.e$
$\geq\mid\sim$	**larger-left-aligned**	$\sigma_1.s = \sigma_2.s$ and $\sigma_1.e \geq \sigma_2.e$
$\leq$	**shorter**	$\sigma_1.s \geq \sigma_2.s$ and $\sigma_2.e \leq \sigma_2.e$
$\leq\sim\mid$	**shorter-right-aligned**	$\sigma_1.s \geq \sigma_2.s$ and $\sigma_1.e = \sigma_2.e$
$\leq\mid\sim$	**shorter-left-aligned**	$\sigma_1.s = \sigma_2.s$ and $\sigma_1.e \leq \sigma_2.e$
$\sim\mid$	**right-aligned**	$\sigma_1.e = \sigma_2.e$
$\mid\sim$	**left-aligned**	$\sigma_1.s = \sigma_2.s$
$\times$	**cross**	$(\sigma_1.s < \sigma_2.s$ and $\sigma_1.e > \sigma_2.s$ and $\sigma_1.e < \sigma_2.e)$ OR $(\sigma_1.s > \sigma_2.s$ and $\sigma_1.s < \sigma_2.e$ and $\sigma_1.e > \sigma_2.e)$
$\cdot\times$	**cross-first**	$\sigma_1.s < \sigma_2.s$ and $\sigma_1.e > \sigma_2.s$ and $\sigma_1.e < \sigma_2.e$
$\times\cdot$	**cross-next**	$\sigma_1.s > \sigma_2.s$ and $\sigma_1.s < \sigma_2.e$ and $\sigma_1.e > \sigma_2.e$

In the space S_1 of Figure 1, here are a few examples of true values:

- $(3, f, 4) \cdot (4, g, 6)$
- $(3, f, 4)||(3, h, 4)$
- $(4, g, 6) > | \sim (4, c, 5)$

Figure 3 shows, in a more intuitive way, examples of relative position operators.

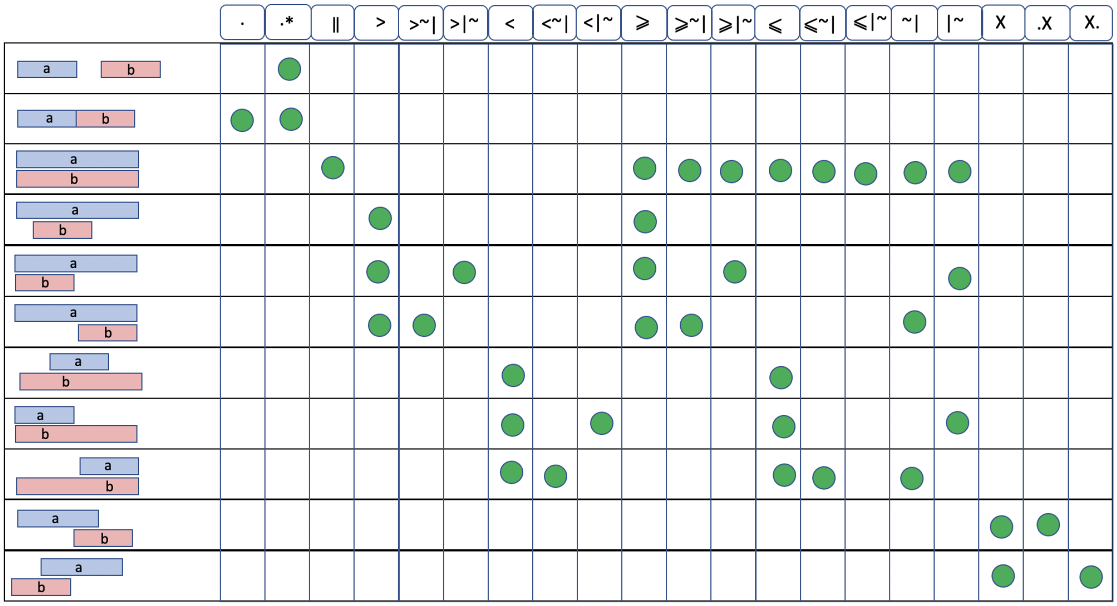


Figure 3: Example of Operators

We should immediately remark that OP is neither minimal nor maximal. In other words, it is possible to define other operators and it is also possible to work with a subset of OP. We have chosen this set for convenience (as will be shown later) and for the ease with which they are to visualize intuitively.

4.5 Syntactic Definition of Patterns

Let us consider again the pattern p_1:

$$p_1 = a\ (>\sim \mid\ b \mid (|| \, f \mid h\,)\,)\, c$$

It can be loosely expressed by the following sentence: *match a 'a' followed by an interval 'b' with two parallel intervals, labeled 'h' and 'f' respectively, right-aligned to itself followed by a 'c'*. Applied to S_1 of Figure 1, its set of matches is a single match. That match is a subset of S_1. The set of matches of p_1 on S_1 is:

$$\{\{(0,a,2),(2,b,4),(3,f,4),(3,h,4),(4,c,5)\}\}$$

In order to define that matching operation formally, we define patterns such as p_1 syntactically.

We first define the following set:

$$\overline{OP} = \{\ \text{'(op's.t. op} \in \text{OP}\} \cup \{\text{'|', ')'}\}$$

We then define the language $L_1 \subset (\Sigma \cup \overline{OP})^*$ as the context-free language defined by grammar G_1 of Figure 4 where 'label' is any element of Σ. As expected, $p_1 \in L_1$.

```
(1) pat :- label
(2) pat :- (op pat | pat )
(3) pat :- pat pat
```

Figure 4: G_1: Grammar Defining L_1: Pattern Expressions

4.6 Pattern Matching Semantic

Now that we have defined patterns and spaces, we need to define pattern maching. Pattern matching will be the operation that, given an input $\gamma-$space $S \in \mathbb{G}(\Sigma)$ and a pattern expression $p \in L_1(\Sigma)$, returns a set called match-set(S, p) defined recursively as follows:

Definition 3 (match-set for pat $:-$label) *If $a \in \Sigma$ and $S \in \mathbb{G}(\Sigma)$, then*

$$\textit{match-set}(S, a) = \{((s,e), \{(s,a,e)\}) \textit{ s.t. } (s,a,e) \in S\}$$

which simply means that matching a label returns as matches all the segments with that label with the corresponding range. For instance, for S_1 of Figure 1,

$$\text{match-set}(S_1, f) = \{((3,4), \{(3,f,4)\})\}$$

Definition 4 (match-set for $(\cdot p_1|p_2)$) *If $p_1, p_2 \in L_1(\Sigma)$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{match-set}(S, (\cdot p_1|p_2)) = \textit{match-set}(S, p_1\, p_2) =$$
$$\{((s_1,e_2), M_1 \cup M_2) \textit{ such that there is}$$
$$((s_1,e_1), M_1) \in \textit{match-set}(S, p_1) \textit{ and } ((s_2,e_2), M_2) \in \textit{match-set}(S, p_2) \textit{ with } e_1 = s_2\}$$

For instance, for the same space S_1:

$$\text{match-set}(S_1, fg) = \{((3,6), \{(3,f,4),(4,g,6)\})\}$$

Definition 5 (match-set for $(||p_1|p_2)$) *If $p_1, p_2 \in L_1(\Sigma)$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{match-set}(S, (||p_1|p_2)) =$$
$$\{((s_1,e_1), M_1 \cup M_2) \textit{ such that there is}$$
$$((s_1,e_1), M_1) \in \textit{match-set}(S, p_1) \textit{ and } ((s_1,e_1), M_2) \in \textit{match-set}(S, p_2)\}$$

For instance

$$\text{match-set}(S_1, (||f|h)) = \{((3,4), \{(3,f,4)), (3,h,4)\})\}$$

Definition 6 (match-set for $(>\sim \mid p_1|p_2)$) *If $p_1, p_2 \in L_1(\Sigma)$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{match-set}(S, (>\sim \mid p_1|p_2)) =$$

$$\{((s_1, e_1), M_1 \cup M_2) \textit{ such that there is}$$

$$((s_1, e_1), M_1) \in \textit{match-set}(S, p_1) \textit{ and } ((s_2, e_2), M_2) \in \textit{match-set}(S, p_2) \textit{ with } e_1 = e_2 \textit{ and } s_1 < s_2\}$$

For instance

$$\text{match-set}(S_1, (>\sim \mid b|\,f)) = \{((2,4), \{(2,b,4), (3,f,4)\})\}$$

The other operators can be defined in similar way according to the definitions of the table of Section 4.4. The match of p_1 on S_1 can now be expressed as

$$\text{match-set}(S_1, p_1) = \{((0,5), \{(0,a,2), (2,b,4), (3,f,4), (3,h,4), (4,c,5)\})\}$$

4.7 Transformation Expression

In order to define transformations, mappings that modify $\gamma-$spaces, we first need to define transformation expressions.

Transformation expressions are defined by the expressions generated by the grammar G_2 of Figure 5. We call $L_2(\Sigma)$, or simply L_2 when Σ is known, the formal language generated by G_2.

```
(1) tpat :- label
(2) tpat :- (op tpat | tpat )
(3) tpat :- tpat tpat
(4) tpat :- (- tpat )
(5) tpat :- (+ label tpat )
```

Figure 5: G_2 where label $\in \Sigma$: Grammar for Transformation Expressions

Since G_2 contains all the rules of G_1, $L_1(\Sigma) \subset L_2(\Sigma)$. For instance, we can define a transformation expression $t_1 \in L_2(\Sigma)$ based on the previously defined pattern p_1:

$$t_1 = a\,(+j\,(>\sim \mid b \mid (||\,f \mid (-h)\,)\,)\,c)$$

t_1 can be seen as an extension of p_1 through the addition of two operations: (1) removing h and (2) adding at segment j starting at b and ending at c.

We will see how t_1 defines a mapping $\widetilde{t_1} : \mathbb{G}(\Sigma) \to \mathbb{G}(\Sigma)$ which we call a $\gamma-$*space transformation*. For instance, $\widetilde{t_1}$ transforms S_1 into S_2 displayed on Figure 2.

$$S_2 = \widetilde{t_1}(S_1)$$

4.8 Applying Transformations

For any transformation $t \in L_2(\Sigma)$, we can obtain the underlying pattern $p \in L_1(\Sigma)$ by removing all the '+' and '-' operations. We call this mapping p_{2to1} the projection from L_2 to L_1. Note that $p_{2to1}^2 = p_{2to1}$ and that p_{2to1} is the identity on $L_1(\Sigma)$. With that definition, we observe that $p_1 = p_{2to1}(t_1)$. This allows us to extend the definition of match-set to transformations in the following way:

Definition 7 (match-set for $t \in L_2(\Sigma)$) *If $t \in L_2(\Sigma)$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{match-set}(S, t) = \textit{match-set}(S, p_{2to1}(t))$$

For each transformation expression t in $L_2(\Sigma)$ and a given space $S \in \mathbb{G}(\Sigma)$, we will introduce an intermediary concept called modif-set(S, t) which will be a set of quadruples (r, A, D, M) where r is a range and A, D, M are subsets of S called the Addition, Deletion and Matching Sets respectively.

For instance, we will see that:

$$\text{modif-set}(S_1, t_1) = \{((0, 5), \{(2, j, 5)\}, \{(3, h, 4)\}, \{(0, a, 2), (2, b, 4), (3, f, 4), (3, h, 4), (4, c, 5)\})\}$$

We will now defined modif-set(S, t) recursively:

Definition 8 (modification set for tpat :- label) *if $a \in \Sigma$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{modif-set}(S, a) = \{(r, \emptyset, \emptyset, M) \textit{ s.t. } (r, M) \in \textit{match-set}(S, a)\}$$

Definition 9 (modificationset for tpat :- (op tpat , tpat)) *Let $t = (op\ t_1|t_2) \in L_2(\Sigma)$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\begin{gathered}\textit{modif-set}(S, (op\ t_1|t_2)) = \{(r, A_1 \cup A_2, D_1 \cup D_2, M) \textit{ s.t. } (r, M) \in \textit{match-set}(S, (op\ t_1|t_2)) \\ \textit{and there is} \\ (r_1, A_1, D_1, M_1) \in \textit{modif-set}(S, t_1) \textit{ and } (r_2, A_2, D_2, M_2) \in \textit{modif-set}(S, t_2)\}\end{gathered}$$

Definition 10 (modificationset for tpat :- (- tpat)) *If $t_1 \in L_2$ and $S \in \mathbb{G}(\Sigma)$ then,*

$$\textit{modif-set}(S, (-t_1)) = \{(r_1, A_1, D_1 \cup M_1, M_1) \textit{ s.t. } (r_1, A_1, D_1, M_1) \in \textit{modif-set}(S, t_1)\}$$

Definition 11 (modificationset for tpat :- (+ label tpat)) *Let $t_1 \in L_2(\Sigma)$, $a \in \Sigma$ and $S \in \mathbb{G}(\Sigma)$ then*

$$\textit{modif-set}(S, (+a\, t_1)) = \{((s_1, e_1), A_1 \cup \{(s_1, a, e_1)\}, D_1, M_1) \textit{ s.t. } ((s_1, e_1), A_1, D_1, M_1) \in \textit{modif-set}(S, t_1)\}$$

Given an expression representing a transformation expression t, and a $\gamma-$space S, we can define the transformation of that expression on this graph as:

Definition 12 ($\gamma-$Space Transformation of a Transformation Expression) *Given an alphabet Σ, a transformation expression $t \in L_2(\Sigma)$, $\widetilde{t} : \mathbb{G}(\Sigma) \to \mathbb{G}(\Sigma)$ is the mapping such that for each $S \in \mathbb{G}(\Sigma)$:*

$$\widetilde{t}(S) = (S - \bigcup_{(r,A,D,M) \in \textit{modif-set}(S,t)} D) \cup \bigcup_{(r,A,D,M) \in \textit{modif-set}(S,t)} A$$

In other words, we take all the modifications of t and remove all the segments that appear in any deletion set and then add all the segments that appear in addition set.

Since there is no ambiguity, we will often write $t(S)$ instead of $\widetilde{t}(S)$.

With this definition, $S_2 = \widetilde{t_1}(S_1)$, or simply $S_2 = t_1(S_1)$.

We could use the alternative formula:

$$(S \cup \bigcup_{(r,A,D,M) \in \text{modif-set}(S,t)} A) - \bigcup_{(r,A,D,M) \in \text{modif-set}(S,t)} D$$

and it would change the definition (the second formula allows to remove segments that have just been added). However, it wouldn't change its applications in what will follow.

Definition 13 ($\gamma-$Space Transformation of a Set of Transformation Expressions) *Given an alphabet Σ, a set τ of transformation expressions, $\widetilde{\tau} : \mathbb{G}(\Sigma) \to \mathbb{G}(\Sigma)$ is the mapping such that for each $S \in \mathbb{G}(\Sigma)$:*

$$\widetilde{\tau}(S) = (S - \bigcup_{t \in \tau} \bigcup_{(r,A,D,M) \in \textit{modif-set}(S,t)} D) \cup \bigcup_{t \in \tau} \bigcup_{(r,A,D,M) \in \textit{modif-set}(S,t)} A$$

Definition 14 ($\gamma-$Space Transformation) *Given Σ, a $\gamma-$Space Transformation is a mapping $m : \mathbb{G}(\Sigma) \to \mathbb{G}(\Sigma)$ such that there exist a set of transformation expressions $\tau \subset L_2(\Sigma)$ with $m = \widetilde{\tau}$.*

The set of all $\gamma-$Space transformations on Σ is denoted $\mathbb{T}(\Sigma)$.

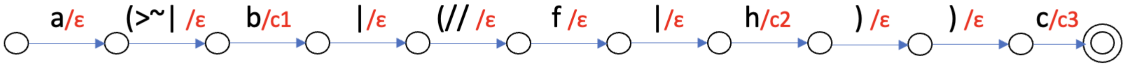
5 Geometric Transductions, $\gamma-$Space Transformations with Finite-State Transducers

What makes these transformations expression powerful is that a very large number of transformation expressions can be applied in parallel to one input space. Given a set τ of transformation expressions, we first transform each expression $t \in \tau$ into a Finite-State Transducer (FST) T_t and then combine all of these into a single FST $T = \bigcup_{t\in\tau} T_t$. Once T available, for any new space $S \in \mathbb{G}(\Sigma)$, we apply an algorithm which we call Trans$-\gamma$ such that $\text{Trans} - \gamma(T, S) = \widetilde{\tau}(S)$.

Let us consider again the transformation expression t_1:

$$t_1 = a\,(+j\,(>\sim \mid b \mid (\|\, f \mid (-h)\,)\,)\,c)$$

which transforms S_1 into S_2 of Figure 2. We will convert t_1 into the FST T_1 of Figure 6. The input symbols are elements of $\Sigma \cup \overline{OP}$ and the output symbols are sequences of bytecodes to be then executed by a virtual machine. In this example, $c1$, $c2$ and $c3$ are the non-empty bytecode sequences.



code segment index	bytecode segment
c_1	(START_CREATE_BLOCK, j, 1)
c_2	(DELETE_BLOCK, ϵ, ϵ)
c_3	(END_CREATE_BLOCK, 1, ϵ)

Figure 6: T_1 representing t_1 and the bytecode sequences

The algorithm Trans$-\gamma$ works in two steps. The first step resembles a traditional FST matching algorithm with some shallow pushdown operations and produces as output a sequence of bytecodes; each bytecode being, as is common in the compilation of many programming language, a 3-uple of the shape $(\text{operand}, \text{argument } 1, \text{argument } 2)$. This sequence of bytecode, together with the matching segments, represents a program which is then executed in the second step. In our example, when T_1 is applied to S_1, the output is the program P_1 of Figure 7. Note that, the matching segments are part of the program, hence, on Figure 7, the program P_1 is constituted by both columns.

matching segment	bytecode
$(0, a, 2)$	(START_CREATE_BLOCK, j, 1)
$(3, h, 4)$	(DELETE_BLOCK, ϵ, ϵ)
$(4, c, 5)$	(END_CREATE_BLOCK, 1, ϵ)

Figure 7: P_1: Program to be run on $\gamma-$VM

Running this program produces the following addition and deletion sets:

$$A = \{(2, j, 5)\} \text{ and } D = \{(3, h, 4)\}$$

Finally, we get

$$\widetilde{T_1}(S_1) = \widetilde{t_1}(S_1) = S_2 = (S - D) \cup A$$

5.1 Preliminary Symbols

Let us introduce the symbols necessary to further describe the algorithm.

We will call C is the set of possible virtual machine bytecodes such as the one already seen for T_1 on Figure 6. C contains the of triples of the following shape:

opcode	arg1	arg2
START_CREATE_BLOCK	label	new block ID
END_CREATE_BLOCK	ϵ	ϵ
DELETE_BLOCK	new block ID	ϵ

With C defined, we can define the FST T to be equal to $((\Sigma \cup \overline{OP}), C^*, X, i, F, E)$ where

- $\Sigma \cup \overline{OP}$ is the input alphabet
- C^* is the output alphabet, i.e. the set of, possibly empty, sequences of bytecodes from C
- X is the set of states
- $i \in X$ is the initial state
- $F \subset X$ is the set of final states
- $E \subset X \times (\Sigma \cup \overline{OP}) \times C^* \times X$ is the set of transitions

The first step of the algorithm, described in Appendix A, will build an FST B equal to $(S, C^*, Z, i_B, F_B, E_B)$ where:

- S is the input alphabet. In other words, the each input label of B will be a segment of S
- C^* is the output alphabet. Every output label of B will be a possibly empty sequence of bytecodes
- $Z \subset X \times \mathscr{F}^+$ is the set of states where $\mathscr{F}^+$ is a non empty sequence of frames. We define the set of possible frames $\mathscr{F}$ below.
- $i_B \in Z$ is the intial state
- $F_B \subset Z$ is the set of final states
- $E_B \subset Z \times S \times C^* \times Z$ is the set of transitions

Each Frame $\phi \in \mathscr{F}$ is a t-uple where the first element is a symbol $y \in Y$ as listed below and the second element is a possibly empty sequence of integers. In other words $\mathscr{F} \subset Y \times \mathbb{Z}^*$. We will write each frame in the following way $[y, n_1, n_2, ..., n_k]$. For instance, [flow, $1, 2$] and [out] are possible frames.

Y (symbols)	description
out	outside of any match
flow	within the flow of string pattern matching
flowrange	within the flow with range constraint
flowmax	within the flow with max right constraint
smallRightAlign1	Smaller-Right-Align operator $<\sim|$, first argument
smallRightAlign2	Smaller-Right-Align operator $<\sim|$, second argument
align1	Align operator $||$, first argument
align2	Align operator $||$, second argument

5.2 Trace of Trans$-\gamma$ on Examples T_1 and S_1

In order to make the algorithm Trans-γ more concrete, we follow one example step-by-step: we take T_1 of Figure 6 and S_1 of Figure 1 as inputs.

The trace is given here:

i of z	$z \in Z \subset X \times \mathscr{F}^+$	transition of T	segment of S	transition of B
0	$(0, [\text{out}])$	$(0, a, \epsilon, 1)$	$(0, a, 2)$	$(0, (0, a, 2), \epsilon, 1)$
1	$(1, [\text{flow}, 0, 2])$	$(1, (<\sim\|, \epsilon, 2)$		$(1, \epsilon, \epsilon, 2)$
2	$(2, [\text{flow}, 2, 2][\text{smallRightAlign1}][\text{flow}, 0, 2])$	$(2, b, c_1, 3)$	$(2, b, 4)$	$(2, (2, b, 4), c_1, 3)$
3	$(3, [\text{flow}, 2, 4][\text{smallRightAlign1}][\text{flow}, 0, 2])$	$(3, \|, \epsilon, 4)$		$(3, \epsilon, \epsilon, 4)$
4	$(4, [\text{flowrange}, 3, 4][\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(4, (//, \epsilon, 5)$		$(4, \epsilon, \epsilon, 5)$
5	$(5, [\text{flowrange}, 3, 4][\text{align1}][\text{flowrange}, 3, 4]$ $[\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(5, f, \epsilon, 6)$	$(3, f, 4)$	$(5, (3, f, 4), \epsilon, 6)$
6	$(6, [\text{flowmax}, 3, 4, 4][\text{align1}][\text{flowrange}, 3, 4])$ $[\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(6, \|, \epsilon, 7)$		$(6, \epsilon, \epsilon, 7)$
7	$(7, [\text{flowmax}, 3, 3, 4][\text{align2}, 3, 4][\text{flowrange}, 3, 4])$ $[\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(7, h, c_2, 8)$	$(3, h, 4)$	$(7, (3, h, 4), c_2, 8)$
8	$(8, [\text{flowmax}, 3, 4, 4][\text{align2}, 3, 4][\text{flowrange}, 3, 4])$ $[\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(8,), \epsilon, 9)$		$(8, \epsilon, \epsilon, 9)$
9	$(9, [\text{flowmax}, 3, 4, 4][\text{smallRightAlign2}, 2, 4][\text{flow}, 0, 2])$	$(9,), \epsilon, 10)$		$(9, \epsilon, \epsilon, 10)$
10	$(10, [\text{flow}, 0, 4])$	$(10, c, c_3, 11)$	$(4, c, 5)$	$(10, (4, c, 5), c_3, 11)$
11	$(11, [\text{flow}, 0, 5])$			

Here are some more explanations for each step:

0. We initialize the process with the initial state of B is $(0, [\text{out}])$ where 0 refers to the initial state of T and the first frame [out] where out $\in Y$ indicates that no matching has yet occurred. Running through CODE-OUT, after matching the segment $[0, a, 2]$, the new computational state is generated: $[\text{flow}, 0, 2]$ indicates that we are within the default pattern matching phase, indicated by the symbol flow, with a match that started at position 0 and has now reached position 2
1. the symbol $(<\sim|$ from the transition starting at state 1 of T_1 is read. This symbol indicates that we are now in the process of matching the first argument of a $<\sim|$, that is 'strictly-smaller-left-align', operator. The deepest frame, $[\text{flow}, 0, 2]$, stores what has already been matched. The second frame [SmallRightAlign1] indicates that we are in the process of matching the first argument of that operator. The last frame $[\text{flow}, 2, 2]$ indicates that we are starting a new regular match at position 2
2. after reading the symbol b in the transition from T_1 and the segment $(2, b, 4)$ from S_1, $[\text{flow}, 2, 2]$ becomes $[\text{flow}, 2, 4]$ indicating that the current match that started as 2 is now at position 4.
3. after reading the separator $|$, we transition to the second argument of the current operator which is indicated by the frame $[\text{SmallRightAlign2}, 2, 4]$. In that frame, 2 and 4 indicate the range of the first argument. The top frame $[\text{flowrange}, 3, 4]$ indicates that we start a regular match that has to happen between 3 and 4. In fact, it has to end at 4 because of the semantic of 'left-aligned' and it has to start strictly after 2 because of the semantic of 'strictly-smaller'.
4. after reading the symbol $(//$ from T_1, we prepare matching the first argument of yet another operator: the 'align' operator. We add the frame [align1] to indicate that fact. We also know that this operator will need to happen with the current range constraint which is indicated by the top $[\text{flowrange}, 3, 4]$.
5. we read the segment $(3, f, 4)$ which transforms the top level flowrange into a frame $[\text{flowmax}, 3, 4, 4]$ where the three numbers respectively represent the starting position of the current match, the ending position of the current match and the maximum allowed ending position. All other frames are left unchanged.
6. reading $|$ makes us transition to the second argument of the 'align' operator. [align1] is replaced by $[\text{align2}, 3, 4]$ in which 3 and 4 the range expected of the second argument.
7. consuming segment $(3, h, 4)$, we simply change the top flowmax frame to $[\text{flowmax}, 3, 4, 4]$. At the same time a transition with that segment and the code c_2 is created on B.
8. reading the closing parenthesis), we verify and wrap-up the 'align' operator. The top two frames are removed and the flowrange frame is replaced with a flowmax frame $[\text{flowmax}, 3, 4, 4]$ where the first two numbers indicate the current matched range and the last number indicates that maximum left side of each range.

9. reading the second closing parenthesis, we verify and wrap-up the $<\sim|$ operator and updated the flow frame to the full range already mapped: $[\text{flow}, 0, 4]$.
10. we read the segment $(4, c, 5)$ which simply increases the range of the flow frame. At the same time, we add a transition with that segment and code index c_3 to B. Because $x = 11$ of T_1 is terminal, state 11 of B is also terminal.
11. At this point B is final. B has only one valid path:

$$((0,a,2),\epsilon)(\epsilon,\epsilon)((2,b,4),c_1)(\epsilon,\epsilon)(\epsilon,\epsilon)((3,f,4),\epsilon)(\epsilon,\epsilon)((3,h,4),c_2)(\epsilon,\epsilon)(\epsilon,\epsilon)((4,c,5),c_3)$$

12. This path is simplified by removing every transition that has a ϵ at the code position. This leads to this simplified path:

$$((2,b,4),c_1)((3,h,4),c_2)((4,c,5),c_3)$$

13. From this path the following program P_1 is generated.

segment of S	Code Index	Code Segment
$(2, b, 4)$	c_1	(START_CREATE_BLOCK, j, 1)
$(3, h, 4)$	c_2	(DELETE_BLOCK, ϵ, ϵ)
$(4, c, 5)$	c_3	(END_CREATE_BLOCK, 1, ϵ)

14. P_1 is then applied to S_1 to produces the following $A = \{(2, j, 5)\}$ and $D = \{(3, h, 4)\}$.
15. $S_2 = (S_1 - D) \cup A$

The full algorithm is described in Appendix A.

6 $\gamma-$ Space Transformations with Bimachines

When large sets of rules are compiled in a single FST, the non-determinicity of this FST grows which makes the runtime algorithm less efficient (i.e. it builds a large number of states of B which are not coaccessible, there is not path from these states to any terminal state).

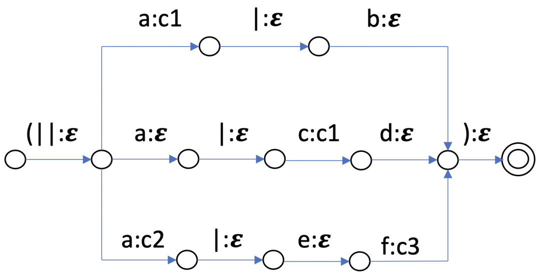
Let us consider the transformation $t_2 = t_3 \cup t_4 \cup t_5$ where:

$$t_3 = (||\,\text{a:-}\,|\,\text{b}\,) \quad (1)$$

$$t_4 = (||\,\text{a}\,|\,\text{c:-}\,\text{d}\,) \quad (2)$$

$$t_5 = (+\text{j}\,(||\,\text{a}\,|\,\text{e f}\,)) \quad (3)$$

t_2 can be represented by the following FST T_2 such that $\widetilde{T_2} = \widetilde{t_2}$:

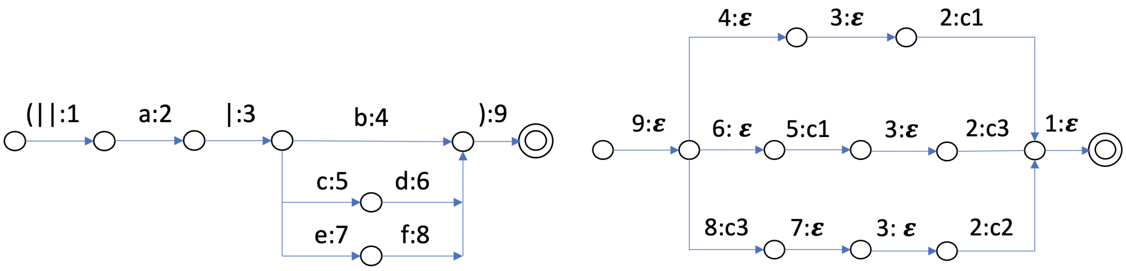


with the following code segments:

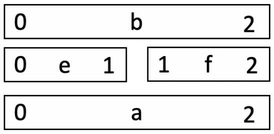
Code Index	Code Segment
c_1	(DELETE_BLOCK, ϵ, ϵ)
c_2	(START_CREATE_BLOCK, j, 1)
c_3	(END_CREATE_BLOCK, 1, ϵ)

Looking at the graph of T_2, there are indeed three transitions with label a as input out of the second state from the left. It is not possible to know which transition will be successful by looking only at the left context. This is general problem of FSTs as they favor a left-to-right direction. Using Bimachines re-introduces symmetry. Bimachines were first introduced by Schutzenberger (1961), described in more detail by Eilenberg (1974) and Berstel (1979). Roche and Schabes (1997) gives a description of Bimachines in the context of language processing.

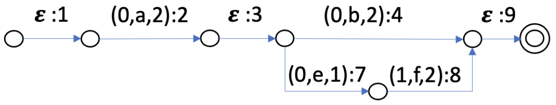
While the canonical representation Bimachines is completely symmetrical, it is more convenient to represent them with two FSTs. For instance, T_2 can be decomposed into two FSTs T_a (left) and T_b (right):



Taking the example of the $\gamma-$space S_3:

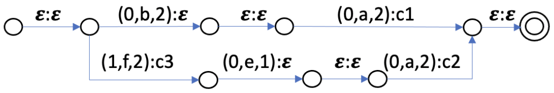


we first apply T_a as in the Trans$-\gamma$ algorithm to produce the following FST B_2:



Each transition of B_2 stores both the matched segment if there is one and the output label of the transition of T_a.

At that point, B_2 is read backward (starting from the terminal state on the right and moving toward the initial state) and applied to T_b through the traditional intersection operation. The output labels of B_2 or matched with the input labels of T_b. The result of that operation is the following B_2' where each transition stores the matching segment (if any) that was stored as input label of B_2 and the code index given by T_b:



Each path of B_2', read backward, produces a program (epsilon/epsilon transitions are ignored). In our example, this produces the two programs P_1' and P_2':

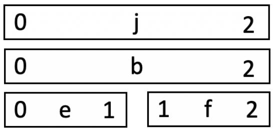
P_1':

segment of S	Code Index	Code Segment
$(0, a, 2)$	c_1	(DELETE_BLOCK, ϵ, ϵ)

P_2':

segment of S	Code Index	Code Segment
$(a, 0, 2)$	c_2	(START_CREATE_BLOCK, j, 1)
$(1, f, 2)$	c_3	(END_CREATE_BLOCK, 1, ϵ)

Finally, these two programs are applied to S_3 to produce the output space S_4:



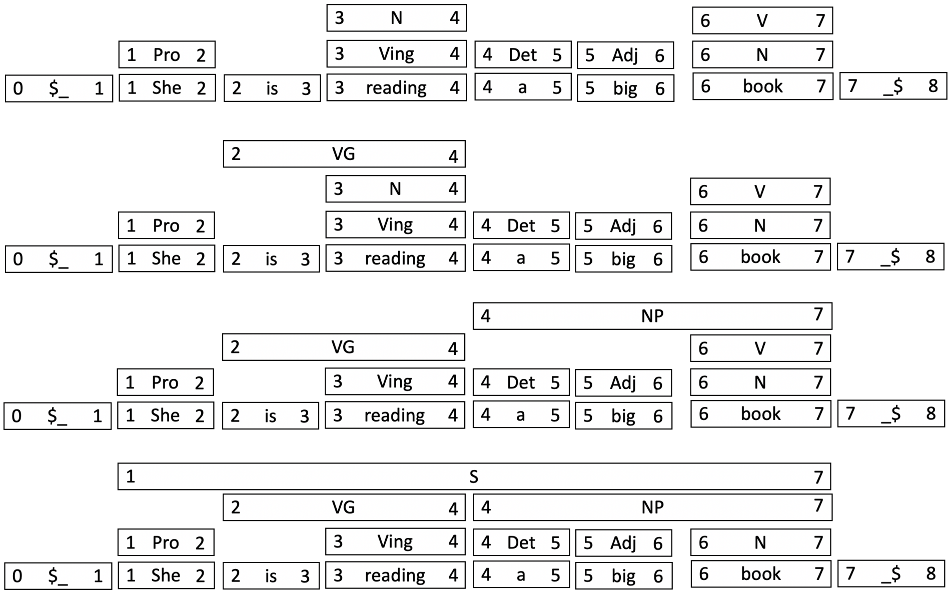
7 Application to Natural Language Parsing

The natural language example presented here is not intended to describe a full parsing algorithm but simply to allude to why $\gamma-$transformations are related to natural language syntax and to parsing.

Let us consider the following sentence:

She is reading a big book

We will see that the following sequence of $\gamma-$spaces from U_1 (at the top) to U_4 (at the bottom) can be seen as the trace of a parsing process:



Tokenization followed by a dictionary lookup, gives us U_1.

Let us further assume that the following five transformation expressions are known:

$$\begin{aligned}
t_{nat1} &= \text{Pro (+VG is Ving)Det} && (4)\\
t_{nat2} &= \text{(<}\sim \mid \text{ N:- } \mid \text{VG)} && (5)\\
t_{nat3} &= \text{VG (+NP Det Adj N) \_\$} && (6)\\
t_{nat4} &= \text{(|| NP | Det Adj V:-)} && (7)\\
t_{nat5} &= \text{\$\_ (+S Pro (<}\sim \text{ | VG | reading) (<}\sim \text{ | } NP \text{ |book)) \_\$} && (8)
\end{aligned}$$

If we call τ_{nat} the set of all transformation expressions:

$$\tau_{nat} = \bigcup_{i=1,5} \{t_{nat_i}\}$$

then, it can be verified that $U_2 = \widetilde{\tau_{nat}}(U_1)$, $U_3 = \widetilde{\tau_{nat}}(U_2)$, $U_4 = \widetilde{\tau_{nat}}(U_3)$ and finally U_4 is left unchanged: $U_4 = \widetilde{\tau_{nat}}(U_4)$.

We can think of $\widetilde{\tau_{nat}}$ as the part of the English grammar, or syntax (we consider both terms equivalent), necessary to analyze the sample sentence.

8 More on Linguistics: the Lexicon-Grammar

We live in a very peculiar time in the history of formal analysis of language. Consider that language has been recognized as an object of scientific inquiry for more than two thousand years, as Panini's Ashtadhyayi from 500 BCE or Dionysius Thrax's "Art of Grammar" from 100 BCE attest. This tradition continues through time with more recent examples such as Arnaud and Lancelot's "Grammaire de Port-Royal" (1660) or the more descriptive "A Modern English Grammar" from Otto Jesperson (1913).

This long arc seemed to take a turn in the mid-twentieth century with the quest for a formal, by that we mean mathematical, definition of language and its syntax in particularly Harris (Methods in Structural Linguistics, 1951) and Chomsky (Syntactic Structures, 1957) in particular, introduce the foundation of what was thought to be the promise of something radically new: language, and syntax in particular, could be described in a closed, and relatively simple, mathematical formalism. In the case of Chomsky, that meant formal rewriting rules now commonly used to define programming language. It was not made completely clear which type of rules it should be, as the Chomsky Hierarchy was more an indication of what a formalism couldn't be than what it was. Work followed to see if other types of formal grammar would be more suitable, such as Tree Adjoining Grammars by Aravind Joshi.

Something strange happened in the following decades. If we put aside the more philosophical components of Chomsky's discussion (such as universal grammar or innate language knowledge), the part which states that a set of formal rewriting rules can describe the syntax of any language is imminently falsifiable. Indeed, if that statement is true, it should be possible to write such formal syntax for at least one language; English being the most obvious candidate. However, if you try to go to any library today and ask for an "a la Chomsky" grammar of English; it doesn't exist. Strangely, Chomsky never alluded to the size of this book (one volume, ten volumes, the entire library,..). This linguistic revolution that started with a bang ended in a long whisper (see also John Searle's "the end of a Revolution").

Philip Lieberman puts it best when he states: "No comprehensive description of the syntax of any natural language has been achieved, despite intense efforts by hundreds of linguists over the course of more than four decades". . (Lieberman, 2006)

This had been identified long before by Maurice Gross in "On the Failure of Generative Grammar" (Gross, 1979). Surprisingly, this piece which should have been seen as important as the original introduction of Generative Grammar (GG), we largely ignored.

The problem was not that the hypothesis failed, this is, after all, how science advances, but that its failure was never fully acknowledged, at least not by its primary author, Chomsky.

In the same piece, Gross provides the following viewpoint on GG: "The real difference between GG and traditional grammar appears to lie in the nature of the metalanguage which is formalized in GG; in traditional studies, concepts are imperfectly defined and fluctuate with intuition."

which, if GGs have failed, raises the following question: "Is there a formal, perfectly well defined, formalism that can describe either a given language or a substantial part of its syntax?".

This is this question that we attempt to answer positively in this document and the following ones.

9 Conclusion

This document defines the simplest form of geometric transductions. The following documents will elaborate on this formalism to describe this can be applied to designing new state-of-the-art parsing algorithms.

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. Openfst: a general and efficient weighted finite-state transducer library. In *CIAA'07 Proceedings of the 12th International Conference on Implementation and Application of Automata*.

Jean Berstel. 1979. *Transductions and Context-Free Languages*. Teubner, Stuttgart.

Samuel Eilenberg. 1974. *Automata languages and machines*, volume A. Academic Press, New York.

Pablo Gamallo and Marcos Garcia. 2018. Dependency parsing with finite state transducers and compression rules.

Maurice Gross. 1968. *Grammaire transformationnelle du français. Syntaxe du verbe*. Larousse, Paris, France.

Maurice Gross. 1975. *Méthodes en syntaxe*. Hermann, Paris.

Maurice Gross. 1979. On the failure of generative grammar. *Language*, 55(4):859–885.

Maurice Gross. 1986. *Grammaire transformationnelle du français. Syntaxe de l'adverbe*. Cantilene, Paris, France.

Aravind K. Joshi and Yves Schabes. 1992. Tree-adjoining grammars and lexicalized grammars. In *Tree Automata and Languages*. Elsevier Science.

Daniel Jurafsky and James H. Martin. 2019. *Speech and Language Processing*. Draft of October 2, 2019.

Philip Lieberman. 2006. *Toward an Evolutionary Biology of Language*. Belknap Press.

Mehryar Mohri. 2000. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234:177–201.

Emmanuel Roche. 1993. *Analyse syntaxique transformationelle du francais par transducteurs et lexique-grammaire*. Ph.D. thesis, Univeriste Paris 7.

Emmanuel Roche and Yves Schabes. 1997. *Finite-State Language Processing*. MIT Press, Cambridge, MA.

Marcel Paul Schutzenberger. 1961. A remark on finite transducers. *Information and Control*, 4:185–187.

Stuart M. Shieber. 1984. The design of a computer language for linguistic information. In *Proceedings of the Tenth International Conference on Computational Linguistics*, pages 362–366, Stanford, CA.

Max Silberztein. 1993. *Dictionnaires electroniques et analyse automatique de textes : le systeme INTEX*. Masson Ed., Paris.

Terry Winograd. 1972. *Understanding Natural Language*. Academic Press, New York, New York.

Anssi Yli-Jyra. 2005. Linguistic grammars with very low complexity. In *Inquiries into Words, Constraints and Contexts: Festschrift in the Honor of Kimmo Koskenniemi*, CSLI Studies in Computational Linguistics, pages 172–183. CSLI Publications.

A Algorithm Trans-γ: Computing $\widetilde{T}(S)$ given T and S

We can now define the algorithm **Trans-**γ which computes, given an input FST T and a gamma space S, the transformed $\gamma-$space $S' = \widetilde{T}(S)$. This algorithm is given on Figure 8. It uses the notation defined above with T being equal to $((\Sigma \cup \overline{OP}), C^*, X, i, F, E)$ and B, the result of the first step, being equal to $(S, C^*, Z, i_B, F_B, E_B)$.

```
INPUT:
$T = ((\Sigma \cup \overline{OP}), C^*, X, i, F, E)$ // An FST
$S \in \mathbb{G}(\Sigma)$
OUTPUT:
$S' = \widetilde{T}(S)$ // The transformed $\gamma-$space
RUNTIME
STEP 1: $B = f1(T, S)$ where $f1$ is described on Figure 9
STEP 2: Apply trans2-$\gamma$ on $B$ and $S$ // trans2-$\gamma$ is described on Figure 14
```

Figure 8: **Trans-**γ: Transformation of$\gamma-$space

The first step is described on Figure 9. The f_1 algorithm of Figure 9 follows the shape of FSA intersection algorithms. It builds the states and transition of the output object, the FST B in our case, starting from the initial state. Each state contains information about the current state of T and the position in S. To make the algorithm more readable, separate figures show the code associated to the processing of each frame symbol $y \in Y$. For instance, the code CODE-OUT is referring to Figure 10. This should be understood as a GOTO to that part of the code followed by a return at the original point.

CODE-OUT of Figure 10 looks each transition $(x', a, c, x”)$ of T and then, for that label $a \in \Sigma$, for all segments of S with that label. The arrival state will create a frame with $y =$ flow which indicates that we are now at a given position S. In addition, [flow, s, e]) records where that match started in S, namely s.

CODE-FLOW of Figure 11 looks at the case where we are at a given position e_1 in S. We look at all the transitions $(x', a, c, x”)$ at the current state of T, and for each of these, at the segments (e_1, a, e_2) of S starting at e_1 with the same label a.

```
**INPUT:**
$T = (\Sigma \cup \overline{OP}, C^*, X, i, F, E)$ // An FST
$S \in \mathbb{G}(\Sigma)$
**OUTPUT:**
$B = (S, C^*, Z, i_B, F_B, E_B) = f1(T, S)$
**RUNTIME**
$Q =$ FIFO queue whose elements are in $\mathscr{Q}$.
$Q = \emptyset$ // $Q$ starts empty
$i_B = (i, \text{[out]})$
$Z = \{i_B\}; F_B = \emptyset; E_B = \emptyset$
push$(Q, i_B)$
while$(Q \neq \emptyset)$
   $z_1 = (x', \phi) =$ pop(Q) where $\phi == f1...fn$ and $f_i = [y_i, ...]$
   if $f_1 =$ [out]
      CODE-OUT //code below
   else if $f_1 = [\text{flow}, s_1, e_1]$
      CODE-FLOW //code below
   else if $f_1 = [\text{flowrange}, s_1, e_1]$
      CODE-FLOWRANGE//code below
   else if $f_1 = [\text{flowmax}, s_1, e_1, m]$
      CODE-FLOWMAX//code below
```

Figure 9: $f1$: Pattern Matching Phase

```
for $(x', a, c, x”) \in E$
   for $(s, a, e) \in S$
      $z = (x”, [\text{flow}, s, e])$
      if $z \notin Z$
         add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
      add $(z_1, (s, a, e), c, z)$ to $E_B$

for $(x', (<\sim |, x”) \in E$ // note that $(<\sim |$ is one symbol, in fact $(<\sim | \in \overline{OP}$
   $\phi_3 = [\text{flow}, e_1, e_1][\text{smallRightAlign1}] \cdot \phi]$ // adds these three frames at the head
   $z = (x”, \phi_3)$
   if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
   add $(z_1, \epsilon, \epsilon, z)$ to $E_B$

for $(x', (//, x”) \in E$ // note that $(// \in \overline{OP}$ indicates the start of the align operator
   $\phi_3 = [\text{out}][\text{align1}] \cdot \phi$
   $z = (x”, \phi_3)$
   if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
   add $(z_1, \epsilon, \epsilon, z)$ to $E_B$
```

Figure 10: Pattern Matching with FSA: CODE-OUT

```
// we have $f_1 = [\text{flow}, s_1, e_1]$
for $(x', a, c, x”) \in E$
  for $(e_1, a, e_2) \in S$
    $f_2 = [\text{flow}, s_1, e_2]$ // increases the range of flow to $e_2$
    $\phi_2 = \mathscr{F}^{-1} \cdot \phi$ // removes the first frame from $\phi$
    $\phi_3 = f_2 \cdot \phi_2$
    $z = (x”, \phi_3)$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, (s, a, e), c, z)$ to $E_B$

for $(x', (<\sim |, x”) \in E$ // note that $(<\sim | \in \overline{OP}$
  $\phi_3 = [\text{flow}, e_1, e_1][\text{smallRightAlign1}] \cdot \phi$ // adds these three frames at the head
  $z = (x”, \phi_3)$
  if $z \notin Z$
    add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
  add $(z_1, \epsilon, \epsilon, z)$ to $E_B$

for $(x', (//, x”) \in E$ // note that $(// \in \overline{OP}$ indicates the start of the align operator
  $\phi_3 = [\text{flow}, e_1, e_1][\text{align1}] \cdot \phi$ // adds these three frames at the head
  $z = (x”, \phi_3)$
  if $z \notin Z$
    add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
  add $(z_1, \epsilon, \epsilon, z)$ to $E_B$

for $(x', |, x”) \in E$ //handles the separator
  if $\phi = [\text{flow}, s_1, e_1][\text{smallRightAlign1}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    $\phi_3 = [\text{flowrange}, s_1 + 1, e_1][\text{smallRightAlign2}, s_1, e_1] \cdot \phi_2$
    $z = (x”, \phi_3)$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, \epsilon, \epsilon, z)$ to $E_B$
  if $\phi = [\text{flow}, s_1, e_1][\text{align1}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    $\phi_3 = [\text{flowmax}, s_1, s_1, e_1][\text{align2}, s_1, e_1] \cdot \phi_2$
    $z = (x”, \phi_3)$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, \epsilon, \epsilon, z)$ to $E_B$
```

Figure 11: CODE-FLOW

```
// we are in the case where $f_1 = [\text{flowrange}, s_1, e_1]$

for $(x', a, c, x”) \in E$
  for $(s_2, a, e_2) \in S$ with $s_2 \geq s_1$ and $e_2 \leq e_1$
    $f_2 = [\text{flowmax}, s_2, e_2, e_1]$ // $s_2$: where it starts, $e_2$: where it is, $e_1$: left-most allowed point
    $\phi_2 = \mathscr{F}^{-1} \cdot \phi$ // removes the first frame from $\phi$
    $\phi_3 = f_2 \cdot \phi_2$
    $z = (x”, \phi_3)$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    // adds the transition to record which interval matched
    add $(z_1, (s_2, a, e_2), c, z)$ to $E_B$

for $(x', (//, x”) \in E$
  $z = (x”, [\text{flowrange}, s_1, e_1][\text{align1}] \cdot \phi)$
  if $z \notin Z$
    add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
  add $(z_1, \epsilon, \epsilon, z)$ to $E_B$

for $(x', (<\sim |, x”) \in E$
  $z = (x”, [\text{flowrange}, s_1, e_1][\text{smallRightAlign1}] \cdot \phi)$
  if $z \notin Z$
    add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
  add $(z_1, \epsilon, \epsilon, z)$ to $E_B$
```

Figure 12: CODE-FLOWRANGE

```
// we are in the case where $f_1 = [\text{flowmax}, s_1, e_1, m]$
for $(x', a, c, x”) \in E$
  for $(e_1, a, e_2) \in S$ with $e_2 \leq e_1$
    $f_2 = [\text{flowmax}, s_1, e_2, e_1]$ // $s_2$: where it starts, $e_2$: where it is, $e_1$: left-most allowed point
    $\phi_2 = \mathscr{F}^{-1} \cdot \phi$ // removes the first frame from $\phi$
    $\phi_3 = f_2 \cdot \phi_2$
    $z = (x”, \phi_3)$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, (e_1, a, e_2), c, z)$ to $E_B$

for $(x', |, x”) \in E$ //handles the separator
  $z = \emptyset$
  if $\phi = [\text{flowmax}, s_1, e_1, m][\text{align1}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    $z = (x”, [\text{flowmax}, s_1, s_1, e_1][\text{align2}, s_1, e_1] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_1, m][\text{smallRightAlign1}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    $z = (x”, [\text{flowmax}, s_1, s_1, e_1][\text{smallRightAlign2}, s_1, e_1] \cdot \phi_2)$
  if $z \neq \emptyset$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, \epsilon, \epsilon, z)$ to $E_B$

for $(x', \,)\,, x”) \in E$ //handles the end of current operator
  $z = \emptyset$
  if $\phi = [\text{flowmax}, s_1, e_2, m][\text{align2}, s_3, e_3][\text{flowrange}, s_4, e_4] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $s_1 == s_3$ and $e_2 == e_3$
      $z = (x”, [\text{flowmax}, s_3, e_3, e_4] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{align2}, s_3, e_3][\text{flow}, s_4, e_4] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $s_1 == s_3$ and $e_2 == e_3$
      $z = (x”, [\text{flow}, s_4, e_3] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{align2}, s_3, e_3][\text{out}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $s_1 == s_3$ and $e_2 == e_3$
      $z = (x”, [\text{flow}, s_3, e_3] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{align2}, s_3, e_3][\text{flowmax}, s_4, e_4, e_5] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $s_1 == s_3$ and $e_2 == e_3$ and $e_3 \leq e_5$
      $z = (x”, [\text{flowmax}, s_4, e_3, e_5] \cdot \phi_2)$

  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{smallRightAlign2}, s_3, e_3][\text{flowrange}, s_4, e_4] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $e_2 == e_3$ and $e_3 \leq e_4$
      $z = (x”, [\text{flowmax}, s_3, e_3, e_4] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{smallRightAlign2}, s_3, e_3][\text{flow}, s_4, e_4] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $e_2 == e_3$
      $z = (x”, [\text{flow}, s_4, e_3] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{smallRightAlign2}, s_3, e_3][\text{out}] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $e_2 == e_3$
      $z = (x”, [\text{flow}, s_3, e_3] \cdot \phi_2)$
  else if $\phi = [\text{flowmax}, s_1, e_2, m][\text{smallRightAlign2}, s_3, e_3][\text{flowmax}, s_4, e_4, e_5] \cdot \phi_2$ // for some $\phi_2 \in \mathscr{F}^*$
    if $e_2 == e_3$ and $e_3 \leq e_5$
      $z = (x”, [\text{flowmax}, s_4, e_3, e_5] \cdot \phi_2)$
  if $z \neq \emptyset$
    if $z \notin Z$
      add $z$ to $Z$;push$(Q, z)$;if $x'' \in F$ then add $z$ to $F_B$
    add $(z_1, \epsilon, \epsilon, z)$ to $E_B$
```

Figure 13: CODE-FLOWMAX

```
**INPUT:**
$B = (S \times C, Z, i_B, F_B, E_B)$
$S = \{(s_1, l_1, e_1), (s_2, l_2, e_2), ..\}$ // a $\gamma-$space on $\Sigma$
**OUTPUT:**
$S_2$
**RUNTIME**
for each path $p$ of $B$
  assuming $p = ((s_1, l_1, e_1), c_1, ), ..., ((s_i, l_i, e_i), c_i), ...$
  for $((s_i, l_i, s_i), c_i)$ in $p$
    if $c_i = \epsilon$ do nothing;
```

Figure 14: **trans2-**γ Applying Code Segments