A Concatenative Theory of Language

Breandan Considine

June 9, 2021

1 Introduction

Language is linear data structure composed of atomic symbols which can be concatenated to form new words and phrases. For example, this document. Natural language has very flexible set of rules for composition which are learned by example and can adapt over time to the needs of its users.

Artificial languages are also languages, but with a precise set of rules. By design, the rules for composing these languages are much more rigid to enable their mechanical interpretation. A vast number of domain-specific languages, called programming languages, have emerged in recent years.

The distinction between natural and artificial languages are not discontinuous, but rather fall along a spectrum of minimum description length. Linguists have developed various taxonomies for languages based on their algorithmic and information complexity. In practice, most examples are at worst, weakly linear context free due to physical constraints.

The natural question arose, is there a superset of all computable languages? Various equivalent definitions were proposed from Peano's arithmetic, to Turing's machines, to Schönfinkel's combinatory logic, to Church's λ-calculus. Although complete, these languages were shown to be inconsistent (Gödel). However, Presburger (1929) proposes a language which is both complete and consistent, $P ::= 0 \mid 1$, with the following semantics:

1. $\neg(0 = x + 1)$
2. $x + 1 = y + 1 \rightarrow x = y$
3. $x + 0 = x$
4. $x + (y + 1) = (x + y) + 1$
5. $(P(0) \wedge \forall x(P(x) \rightarrow P(x + 1))) \rightarrow \forall y P(y)$

More recently, a new kind of programming language, with ties to combinatory logic, emerged: concatenative programming. Syntactically, it resembles natural language. It usually takes the following form:

1. $S ::= \texttt{A} \mid ... \mid \texttt{Z}$
2. $W ::= S \mid SS$
3. $F ::= W \mid W\text{␣}W$
4. $D ::= W\text{␣}\texttt{:=}\text{␣}F$

If we consider a slight variation of concatenative programming:

1. $C ::= ! \mid ... \mid \sim$
2. $C ::= CC$

This is similar to a Kleene algebra, except there is no multiplication operator as we wish to remain consistent. Note that C is still expressive enough to encode finite-length ASCII text.

We note $C \to P$ using the successor representation, $0 ::= 1$, $x{+}1 ::= 1x$.[1] Furthermore, $P \to C$ using $xy ::= x + y$. Thus, P and C are equivalent. ■

Via staging, we can embed any ASCII-based language in C. For example, consider the programming language...

What about types? Since C is both complete and consistent, we have boolean judgements. Judgemental equality is simply string equality.

In concatenative programming, words typically denote function application in the point-free style. What if we took a more granular approach and allowed each symbol to be a higher-order function?

Mirroring probabilistic semantics, individual symbols are functions which accept and return a continuation, representing the contextual semantics.

This continuation can be approximated by a multidimensional array, i.e. a tensor. Type checking and inference are performed via tensor contraction. Eliminating repetition is equivalent to factorization.

TODO: discuss type system

[1] In practice unary encoding is inefficient, so we can make an exception and permit lowering to ordinary integer arithmetic during implementation.

2 Symbolic automata

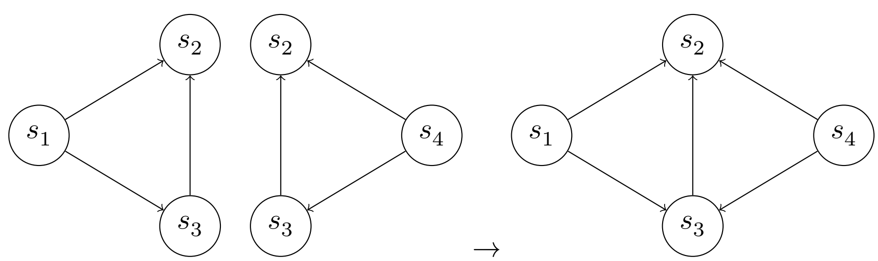
Syntactically our language can be compiled into a graph. Given a sequence $S \in C$, we first perform a byte pair encoding to compress the sequence. Then we have the following semantics:

1. $\texttt{car cdr}, G ::= S_0 S_{1 \ldots n}, \{\}$
2. $\texttt{car cdr}, G \vdash \texttt{cdr}, G \oplus (\texttt{car}, \texttt{cdr.car})$

Concatenation of two graphs $G_1, G_2 : V \times E \times V$ we define as:

1. $G_1 \oplus G_2 ::= (V_1 \cup V_2) \times (E_1 \cup E_2)$

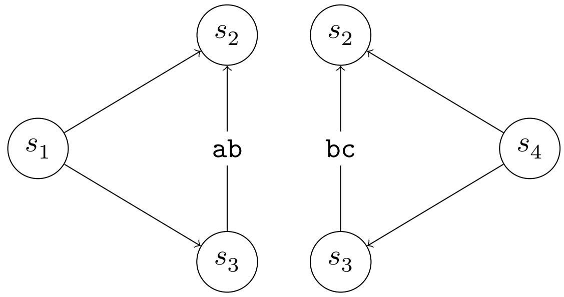
For example, suppose we have the following two graphs, G_1, G_2. $G_1 \oplus G_2$ can be visualized by gluing together the nodes and edges:



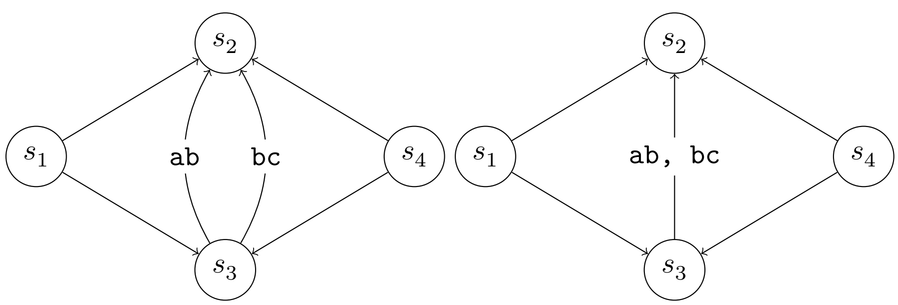
Now suppose we have an edge-labeled graph: we could either take the product or union of the edge labels.

Concatenation of two edge-labeled graphs $LG_1, LG_2 : (V \times E \times \Sigma^* \times V)$:

1. $G_1 \oplus G_2 ::= (V_1 \cup V_2) \times (\Sigma_1^* \cup \Sigma_2^*) \times (E_1 \cup E_2)$
2. $G_1 \oplus G_2 ::= (V_1 \cup V_2) \times \Big((E_1 \cup E_2) \rtimes (\Sigma_1^* \cup \Sigma_2^*)\Big)$



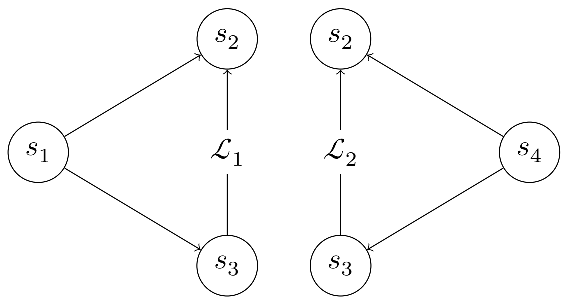
$G \oplus_1 G'$ repeats shared edges with unique labels. $G \oplus_2 G'$ combines shared edges, and unions the labels.



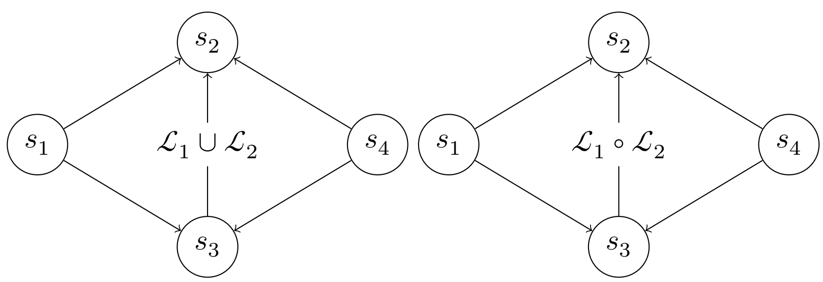
We can also add symbolic automata. Not only must we merge the nodes and edges, we must also consider the merger of edge languages.

Concatenation of two symbolic automata $SA_1, SA_2 : (V \times E \times \mathcal{L} \times V)$:

1. $SA_1 \oplus SA_2 ::= (V_1 \cup V_2) \times (\mathcal{L}_1 \cup \mathcal{L}_2) \times (E_1 \cup E_2)$
2. $SA_1 \oplus SA_2 ::= (V_1 \cup V_2) \times \Big((E_1 \cup E_2) \rtimes (\mathcal{L}_1 \cup \mathcal{L}_2)\Big)$



$G \oplus G'$ can be visualized by stitching together the nodes and unioning the edges:



This is essentially the idea behind symbolic automata. Instead of adding a bunch of edges corresponding to each individual ASCII symbol, we collapse all edges into the union. Each edge represents an algebraic data type in the Kleene algebra. The same idea works on any typed property graph, whose nodes or edges have a summable type.

3 Algebra on Languages

We recall that a language is just a set of strings, i.e. a finite-length sequences of symbols. The concatenation of two languages is essentially the Cartesian product over strings:

$X \circ Y := \{xy \mid x \in X, y \in Y\}$.

It is known that all recursive languages are closed under concatenation, union and intersection. As a recursive language, Presburger arithmetic too, falls under this category.