TERMS AND AUTOMATA THROUGH LOGIC

KAMAL LODAYA

Abstract. This article gives an introduction to one kind of automata and logic connection in the setting of terms, represented as trees. Automata process terms to accept *term languages*, and logical formulae define such languages. There is a close connection between the formalisms. The article then proceeds to structures which are more general than trees, but can be interpreted into trees. Automata continue to process terms, the logics can be used to define languages of the general structures. A connection is maintained through a theory of transductions. Finally the article generalizes terms to straight-line programs, which allow shared processing by concurrent automata, while the logic defines partially ordered sets. Here some connections have been obtained, but the picture is not complete.

Keywords: Term rewriting, tree language, monadic second-order logic, Presburger successor constraints, algebraic automata theory, Petri net, context-free grammar, transformational grammar, tree transduction, series-parallel poset, poset language, Lawvere theory

1. Introducing terms

Fix a finite signature $\Sigma = (\Sigma_0, \dots, \Sigma_m, \Sigma_{AC})$ of finite sets of function symbols of arity 0 to m, $m \geq 0$, as well as an additional set of function symbols Σ_{AC}. At least Σ_0 and Σ_m are nonempty. Function symbols of arity 0 are called *constants* (a set of *variables* X will sometimes be used), arity 1 symbols are *unary*, arity 2 symbols are *binary*, and so on. Σ_{AC} is a set of binary function symbols which are declared to be associative and commutative. Σ is often equated with the set $\Sigma_0 \cup \cdots \cup \Sigma_m \cup \Sigma_{AC}$. Abbreviations are $\Sigma_{>0} = \Sigma_1 \cup \dots \Sigma_m \cup \Sigma_{AC}$, $\Sigma_{\geq 2} = \Sigma_2 \cup \dots \Sigma_m \cup \Sigma_{AC}$ and so on.

The set of all terms over Σ is written T_Σ, or $T_\Sigma(X)$ when a set of variables X is used. Familiar terms over an arithmetic signature are $(a+b)^2$, and $a^2+(2\times a\times b)+b^2$. The infix notation assumes that $+, \times \in \Sigma_{AC}$ are declared associative and commutative, and so is exponentiation. With $Add, Times, Exp \in \Sigma_2$, they would not be assumed associative or commutative. With constant $2 \in \Sigma_0$, variables $a, b \in X$, the first term would be written $Exp(Add(a,b),2)$ and the second one would be written $Add(Exp(a,2), Add(Times(2, Times(a,b)), Exp(b,2)))$.

Thus a *term* is seen as a Σ-labelled tree t, with its domain a prefix-closed set of *nodes*, say $dom(t) \subseteq \mathbb{N}^*$, such that a node labelled by Σ_i, $i = 0, m$, has i successor (child) nodes, a node labelled by Σ_{AC} has at least 2 children. When Σ_{AC} is empty, $m = 1$ and there is one constant, the unary terms are called *words*. A set of terms is called a *term language*.

I thank Paritosh Pandya, Deepak D'Souza and the Calcutta Logic Circle for providing me opportunities to lecture on some of the material for this article. I thank the referee and the editors for allowing me space for clarity.

The tree structure $(dom(t), \{\_ \circ n \mid n \in \mathbb{N}\}, \{f(\_) \mid f \in \Sigma\})$ is given by the tree nodes, concatenating a node by a number to form a child and monadic predicates indicating the node label from Σ. We denote the child relations by $S_1, \ldots, S_m$ or just their union, S. At each node, the monadic predicate corresponding to the Σ-label at that position holds, and no other. The *descendant* relation S^* is abbreviated *desc*. Its converse is the *ancestor* relation *anc*. A node with a label from Σ_{AC} can have any finite number of children (at least two), which are *unordered*. Otherwise the arity of the symbol determines the number of children the node has, and they are *ordered*.

To say it differently, $Exp(Add(a, b), 2)$ is a tree with some node labels. $2, a, b, Add, Exp$ have no meaning, they are symbols. In the definition above the natural numbers are used just to index positions in the tree. Why? As shall be seen later in this article, one can model things which are very different from arithmetic, and where discussion of reasoning and computation makes sense. Addition will be used in a restricted way to count nodes in the tree. This abstract view of terms started from the work of the logician GOTTLOB FREGE in 1879, and was significantly developed by JACQUES HERBRAND AND KURT GÖDEL in the 1930s.

In the term rewriting literature, the treatment of symbols with associativity and commutativity properties is important. See for instance the work of DEEPAK KAPUR AND G. SIVAKUMAR on their Rewrite Rule Laboratory. It appears that the first use of a signature with mixed associativity and commutativity properties in an automata context was in papers by LODAYA AND PASCAL WEIL around 2000, which dealt with structures more general than trees: series-parallel posets, which will be seen later in this article.

1.1. **Outline.** The purpose of this article is to give an introduction to an automata-logic connection in the setting of terms, requiring trees as well as structures which are more general than trees. While the results on these structures are discussed, the article mostly concentrates its explanations on trees.

The term rewriting community within mathematical logic and computer science has known these connections for decades. For instance an online textbook is maintained by the TATA group. WOLFGANG THOMAS has some nice course notes on the subject. Other textbooks are by FRANZ BAADER AND TOBIAS NIPKOW, and by the TERESE group. There is excellent tool support on the logical side, of the Mona tool for monadic second-order (MSO) logic by NILS KLARLUND, ANDERS MØLLER AND MICHAEL SCHWARTZBACH, and the DCvalid tool for interval logic developed by PARITOSH PANDYA around 2001.

Section 5 discusses some applications from natural language parsing. In computer science it is more common to have examples from compilers or document languages like XML (for instance, in THOMAS's notes). With developments in machine learning, natural language processing has grown to be a big area in artificial intelligence. The approach here is a lot more restricted. As in the work of NOAM CHOMSKY and JOACHIM LAMBEK, there is no attempt to go beyond syntactic structure.

Section 6 describes the theory of *transductions*, or functions from trees to trees. More generally one can transduce a tree structure into a graph structure, more than one if the transduction is a relation. There are few direct ideas of automata which work on structures more general than trees, so here logic provides a lead on how to reason about finite state computation on these structures.

The last two sections consider poset structures. These require terms to be extended to straight-line programs. The nets of CARL-ADAM PETRI play the role that the automata play in earlier sections. The logic used is a significant enhancement of existential monadic second-order logic with counting comparisons.

2. PREDICATES AND LOGICAL FORMULAE

A more systematic notation that programmers and computer scientists have come to use is logical formulae. Here is an equality formula describing an arithmetic equivalence: $(a + b)^2 = a^2 + (2 \times a \times b) + b^2$. The formula $a < b$ describes an arithmetic predicate, this can also be denoted by the converse predicate, $b > a$.

This article works with formulae over trees. As we saw in the previous section, there is a significant amount of abbreviation involved. The formula above is an abbreviation for its parse tree, with nodes labelled from the ranked signature with $\Sigma_0 = \{2, a, b\}$, $\Sigma_2 = \{Eq, Exp, Times, Add\}$.

In our logic there are predicates of specified arity on tree positions, such as equality (of tree positions), order, S_i and *desc* from Section 1 above. Each element of the signature is a unary predicate, intuitively $2(\_)$ is *true* and $Add(\_)$ is $false$ for tree nodes labelled by 2, $Add(\_)$ is *true* and $2(\_)$ is $false$ for tree nodes labelled by Add, and so on. That these trees have arithmetic significance is lost, our logic only sees them as trees.

These unary predicates are subsumed under *tree formulae*, they have constants *true*, $false$, unary connective $\neg$ and binary connectives $\wedge, \vee, \oplus, \supset, \equiv$. They include also formulae made up using the quantifiers $\forall x \alpha$ and $\exists x \alpha$. Inside the formula α occurrences of the variable x get *bound* to the quantifier, perhaps to inner quantifiers if x is reused. Variables which are not bound remain *free*. We will also use monadic second-order variables standing for sets and quantify over them $\forall Z \alpha$ and $\exists Z \alpha$, examples of their use are in the next section. Unary label predicates $f(\_)$ also stand for sets, the positions on the tree which are labelled f, but they cannot be quantified out.

Formulae with no free variables—that is, all variables occurring in the formula are bound to quantifiers—are said to be *closed*. Usually boolean formulae will use terms of another signature. $\forall x \forall y (\neg(x < y) \wedge \neg(x > y) \supset x = y)$ is a closed formula with two universal quantifiers over the arithmetic signature which, given an order predicate $<$ between arithmetic terms, expresses linearity of the order. One can ask whether a closed formula is *valid*: that is, it holds over all structures, or if it is *satisfiable*: it holds over some structure.

Positions of variables in the tree are indicated by pointer functions like $s = [x \mapsto \varepsilon, y \mapsto 21]$, called "assignments" in logic textbooks—this one says that variable x maps to the root of the tree, and y to its second child's first child.

For the tree structure we inductively define the satisfaction relation $t, s \models \alpha$ specifying that a tree t with assignment s for free variables, satisfies the formula α. Here is an example

$$\begin{tikzcd}
 & & z \arrow[dll] \arrow[dl, "S_i"] \arrow[d] \arrow[dr, "S_j"'] \arrow[drr] & & \\
 {} & z_1 \arrow[d, dash] & {} & z_2 \arrow[d, dash] & {} \\
 & x & & y &
\end{tikzcd}$$

FIGURE 1. Going up to a least common ancestor

formula β with pointers to free variables z, x, x', y, y' indicated by underlining. There is also a bound variable v.

$$\begin{aligned}
\underline{f}(\underline{g}(\ldots(\underline{a}))), \underline{g'}(\ldots(\underline{a})))) \models\ & (\bigvee_{i=1,m-1} (S_i(z,y) \wedge g(y) \wedge desc(y,x) \wedge \\
& \bigvee_{j=i+1,m} S_j(z,y') \wedge g'(y') \wedge desc(y',x'))) \wedge \\
& f(z) \wedge leaf(x) \wedge a(x) \wedge leaf(x') \wedge a(x'), \\
\text{where } leaf(x) \iff\ & \neg\exists v(\bigvee_{i=1,m} S_i(x,v))
\end{aligned}$$

More precisely, since the underlining is ambiguous, we put the pointers into the tree, expanding the alphabet to $\Sigma \times \wp(Var_1)$, where the variables Var_1 are those which appear in the first-order formula, constrained to occur exactly once in the model.

$$\begin{pmatrix} f \\ \{z\} \end{pmatrix} (\begin{pmatrix} g \\ \{y\} \end{pmatrix} (\begin{pmatrix} \Sigma_{>0} \\ \emptyset \end{pmatrix} \ldots (\begin{pmatrix} a \\ \{x\} \end{pmatrix}))), \begin{pmatrix} g' \\ \{y'\} \end{pmatrix} (\begin{pmatrix} \Sigma_{>0} \\ \emptyset \end{pmatrix} \ldots (\begin{pmatrix} a \\ \{x'\} \end{pmatrix})))) \models \beta(z,x,x',y,y')$$

Position z in the tree having the letter $(f, \{x,y\})$ means that in addition to its having the letter f, the variables x and y are assigned the position z. Quantifying out variables, we get the formula $precedes(x,x') \iff \exists z \exists y \exists y' \beta$, expressing that the leaf x appears before the leaf x' in the frontier of the tree. In the model the variable alphabet shrinks.

$$\begin{pmatrix} \Sigma_{\geq 2} \\ \emptyset \end{pmatrix} (\begin{pmatrix} \Sigma_{>0} \\ \emptyset \end{pmatrix} (\begin{pmatrix} \Sigma \\ \emptyset \end{pmatrix} \ldots (\begin{pmatrix} \Sigma_0 \\ \{x\} \end{pmatrix}))), \begin{pmatrix} \Sigma_{>0} \\ \emptyset \end{pmatrix} (\begin{pmatrix} \Sigma \\ \emptyset \end{pmatrix} \ldots (\begin{pmatrix} \Sigma_0 \\ \{x'\} \end{pmatrix})))) \models precedes(x,x')$$

A Σ-term becomes a model for a closed formula with no free variables. To express that the frontier has no b before an a, for $a, b \in \Sigma_0$, we have the closed formula:

$$\forall x(a(x) \wedge leaf(x) \supset \neg\exists y(b(y) \wedge leaf(y) \wedge precedes(x,y))$$

The set of terms which are *finite models* for a closed formula, is the language *defined* by that formula. Such languages we call *definable.*

Exercise 1. *Write a closed formula for the tree language $TwoALeaves$ saying that there are at least two a-labelled leaves. Write a formula $root(x)$ with free variable x marking the root of the tree. Write formulae $first_f(x)$, $last_f(x)$ which identify the leftmost and rightmost occurrences of an f-labelled node in a tree, and $next_f(x,y)$ which, given an f-labelled node x, identifies the next f-labelled node y in a left-to-right traversal.*

2.1. **Counting formulae.** We have neglected to deal with the associative and commutative signature Σ_{AC}, clearly the previous formulae and exercise have no meaning for a node which has unordered children. To deal with these we use a special child predicate $S(x, \phi)$ in our tree formulae, where ϕ is a quantifier-free *Presburger* formula over the tree signature, using "count" variables $|Z|$, Z a monadic second-order variable or unary predicate, and allowing addition and comparison of these counts. For this formula the assignment of x should be a Σ_{AC}-labelled node and ϕ checks a property of the children of this node. For example, the formula $S(x, |a| + |b| = |c| + 1)$ says that the node assigned by x has c-children whose number is just one below the sum of its a-children and its b-children. Here $+$ is used inside the second argument of the S predicate to *count* children of a tree node, it is not just a label from the signature as we saw in an example in Section 1. Addition is allowed for counting, but not multiplication.

These formulae are named after Mojżesz Presburger, a student of Alfred Tarski, who gave a decision procedure for checking satisfiability and validity of such first-order formulae in 1930. In the context of trees, formulae like these were used by Silvano Dal-Zilio and Denis Lugiez in 2003. This logic is called $P_S MSO$, monadic second-order logic with Presburger successor constraints.

To summarize, a general signature of terms, with operations of various kinds, including some being declared associative and commutative, has been introduced. These terms can also be thought of as trees. Logical formulae specify properties of trees, including counting properties of successors of a node. Checking a logical property of a tree amounts to checking membership of a tree in a language. Checking satisfiability of a logical formula over a tree signature amounts to checking nonemptiness of the associated language. This association will be made effective in the next section, using an idea of computation by automata.

3. Homomorphisms as behaviour

Given a function $h : \Sigma \to Q$, its *homomorphic extension* $h^\# : T_\Sigma \to Q$ is given by $h^\#(f(t_1, \ldots, t_n)) = h(f)(h^\#(t_1), \ldots, h^\#(t_n))$ for $f \in \Sigma_n$. We will sometimes denote $h^\#$ also as h. Thus Q becomes an *algebra*, endowed with term structure. In the special case when the range is itself a term algebra T_Γ, a *term homomorphism* $h^\# : T_\Sigma \to T_\Gamma(X_m)$ is given by $h^\#(f(t_1, \ldots, t_n)) = h(f)(x_{i_1}, \ldots, x_{i_n})$ for $f \in \Sigma_n$, $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, n\}$, where the fresh variables from $X_m = \{x_1, \ldots, x_m\}$ are assigned $x_i = h^\#(t_i)$ for $i = 1, n$. This allows a term homomorphism to produce a term, perhaps of different arity, with subterms copied, or permuted in a different order, as required. We will restrict to *linear* term homomorphisms, where every variable in X_m occurs at most once. This allows permutation but not copying.

When Σ has an associative and commutative symbol $f \in \Sigma_{AC}$, the term homomorphism is weakened to only preserve counting, $h^\#(f(|Z_1|, \ldots, |Z_n|)) = h(f)(h^\#(|Z_{i_1}|), \ldots, h^\#(|Z_{i_n}|))$. For example, if $h(f)(|a|, |b|, |c|)) = g(|x_2|, |x_1|, |x_3|)$ and h maps n_1, n_2, n_3 subtrees at children labelled a, b, c of an f-labelled tree to subtrees t_a, t_b, t_c respectively, these are designated by the variables $|x_1|, |x_2|, |x_3|$ on the right hand side and the extension $h^\#$ makes a g-term having as children n_2 copies of t_a, n_1 copies of t_b and n_3 copies of t_c. Since our signature has associative and commutative symbols Σ_{AC} represented by Presburger counting conditions,

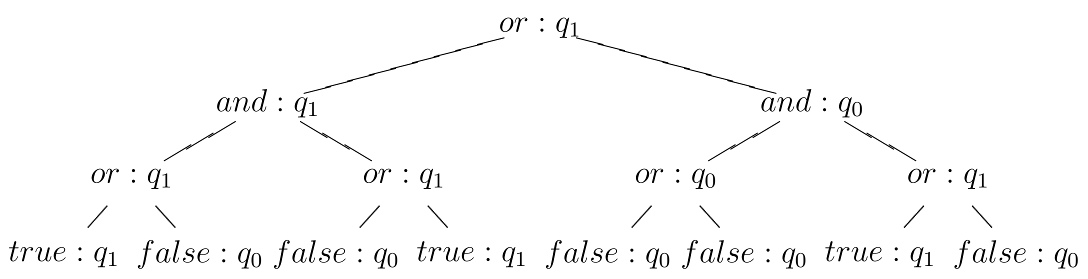

FIGURE 2. Evaluating a Boolean term

this is an appropriate definition of tree homomorphism for these symbols. This has not appeared in the literature earlier.

3.1. **Automata.** Cognitive scientists, like WARREN MCCULLOCH AND WALTER PITTS in the 1940s, invented a graphical notation called *automata*. For trees, this was developed by JAMES THATCHER AND JESSE WRIGHT.

Formally, a *transition system* is an algebra, or directed hypergraph (Q, δ) of *states*, with Σ-ranked *joining* hyperedges (called *transitions*) $\delta : \Sigma_i \to (Q^i \to Q),\ i = 1, m$, taking tuples of states to a state. An automaton is a transition system with a set of *final states* $F \subseteq Q$.

Here is how an automaton operates on a tree. It begins at the leaves, with the transition for constants determining the state at the node. On each interior node, the transition with the node's label takes the states at the children into a state at the node. At the root, if the automaton reaches a final state, it *accepts* the tree. Figure 2 describes how an evaluation proceeds for a Boolean term with signature $\Sigma_0 = \{true, false\}$, $\Sigma_1 = \{not\}$, $\Sigma_2 = \{and, or\}$, and an automaton with a finite set of states $Q = \{q_0, q_1\}$ and $F = \{q_1\}$. The *run* (homomorphic extension) of the automaton on the input tree assigns a state to every node. If we assume that the states are "output" by the processing of the automaton, we can imagine the input tree over Σ giving an output tree $q_1(q_1(q_1(q_1, q_0), q_1(q_0, q_1)), q_0(q_0(q_0, q_0), q_1(q_1, q_0)))$ over the labels Q (which do not form a signature). We will return to such input-output *transductions* in Section 5.2.

The *language accepted* by the automaton is all trees for which the run ends with $\delta^{\#}(t) \in F$. The tree in the example above is accepted because the run ended in $q_1 \in F$. Languages accepted by *finite* automata are called *regular* languages. The trick here is that we can talk of languages of terms or languages of trees, using terms as abbreviations for trees.

Exercise 2. *Design a finite automaton accepting the language Odd of trees which have an odd number of f-labelled nodes, $f \in \Sigma_2$.*

Does looking at languages using automata help in our attempt to describe them in logic? Yes, here is a closed formula mimicking the transition system of the automaton in Figure 2, accepting the regular language of trees (call it $EvalTrue$) over the boolean signature which

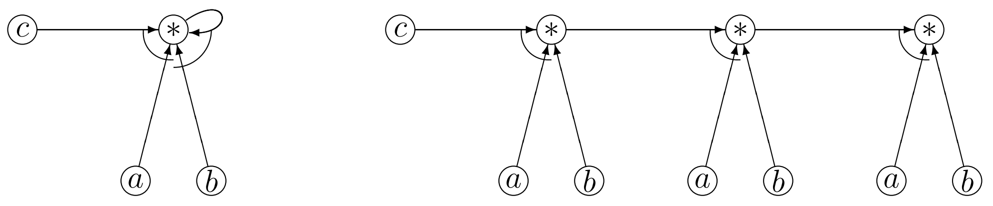

FIGURE 3. Evaluating long terms with an associative operation

represent terms evaluating to *true*. The notation $\oplus$ is used for exclusive or.

$$\begin{array}{l}
\exists Q_0 \exists Q_1 (\forall x (Q_0(x) \oplus Q_1(x)) \\
\wedge \forall z (leaf(z) \supset (true(z) \supset Q_1(z)) \wedge (false(z) \supset Q_0(z))) \\
\wedge \forall x \forall y \forall z (and(z) \wedge S_1(z,x) \wedge S_2(z,y) \supset (Q_1(z) \equiv (Q_1(x) \wedge Q_1(y)))) \\
\wedge \forall x \forall y \forall z (or(z) \wedge S_1(z,x) \wedge S_2(z,y) \supset (Q_1(z) \equiv (Q_1(x) \vee Q_1(y)))) \\
\wedge \forall y \forall z (not(z) \wedge S_1(z,y) \supset (Q_1(z) \equiv Q_0(y))) \\
)
\end{array}$$

This is a formula of *monadic second-order logic (MSO)*, with the variables $Q_0, Q_1 \in Var_2$ ranging over disjoint sets of positions. (Without a finite set of states there is no finite formula.) Each set variable stands for a state of the transition system, and is interpreted as being true for the positions of the tree at which the automaton, operating on the tree, is in that state. A little more is required to fully describe an automaton—the nodes of the tree must respect the arity of the signature, and the final states must be used.

Exercise 3. *Given an arbitrary finite automaton, make up a closed formula describing the language accepted by it.*

Using the earlier exercises, here is an *MSO* formula for the language of trees with an odd number of f-labelled nodes. It is known from model-theoretic arguments that this language cannot be defined in first-order logic.

$$\begin{array}{rl}
\exists Odd \exists Even \forall x (& (f(x) \supset Odd(x) \oplus Even(x)) \wedge \\
& (first_f(x) \supset Odd(x)) \wedge (last_f(y) \supset Odd(y)) \wedge \\
\forall y (& (next_f(x,y) \supset (Odd(x) \oplus Odd(y)))))
\end{array}$$

Formally, models are now also equipped with an interpretation for the second-order variables. Think of the automaton alphabet as being expanded with a set of first- and second-order variables, specifying at each position the variables which are true there.

Exercise 4. *Construct automata on the alphabet $\Sigma \times \wp(\{x,y\}) \times \wp(\{Z\})$ which check if $f(x)$, $S_i(x,y)$ and $Z(y)$ are true.*

3.2. **Nets.** The treatment of symbols which are associative and commutative is more involved. In the 1960s, PETRI came up with a graphical notation called a Petri net system, defined later in this article, which allows the hyperedges to be used more than once in repetitive passes of so-called "tokens". LODAYA AND WEIL modelled this by permitting cyclic joins. The example in Figure 3 illustrates how.

On the left is a net system with four states. Three of the states are reached by reading constants a, b, c. The fourth state is reached by two joining hyperedges on the symbol $*$, assumed associative and commutative. These labels from the signature also name the states. The hyperedge to the left has a self-loop from $a, b, *$ to $*$, the hyperedge to the right from a, b, c to $*$ does not. On the right we have a tree $c * a * a * a * b * b * b$ accepted by the automaton, by processing it as $c * (a * b) * (a * b) * (a * b)$, allowed because $*$ is associative and commutative. By repeating this idea, it can be seen that any product of a's and b's, with one c and as many a's as b's (which can be written as a Presburger constraint $|a| = |b| \wedge |c| = 1$), can be accepted by the net automaton.

For the tree signatures, nodes are labelled. The definitions of Lodaya and Weil were slightly different, they labelled transitions and worked in the setting of series-parallel posets, which are more general than trees. Trees require either joining or forking hyperedges, whereas both are required for series-parallel posets. The transitions were thus broken up into joining, forking and sequential ones. The definitions of net automata with these three kinds of transitions are not pursued here. Instead an intermediate route is taken, that of Helmut Seidl, Thomas Schwentick and Anca Muscholl who defined so-called mixed tree automata, where a joining hyperedge from unboundedly many states to a single target state is described using a Presburger formula.

Apart from the ranked transitions $\delta : \Sigma_i \to (Q^i \to Q)$, $i = 1, m$, for the associative-commutative symbols, we have transitions $\delta : \Sigma_{AC} \to (\mathit{Pres}(Q) \times Q)$. The latter transition is taken only if the sources of the hyperedge satisfy the constraint, a quantifier-free Presburger formula using variables $|q|$ which count how many sources are in state q.

Independently, John Doner in 1970, and Thatcher and Wright in 1968, proved the theorem below for *MSO*, which was extended to the logic extended with Presburger successor counting ($P_S MSO$) by Seidl, Schwentick and Muscholl in 2008.

Theorem 5. *The regular Σ-languages are those definable in $P_S MSO$ (as well as $\exists P_S MSO$) over finite trees.*

Proof. The forward direction begins with the exercises seen so far. To deal with cyclic joining states as in Figure 3, one can introduce a context-free grammar for the derivation of an acyclic term from a cyclic joining state. Because of associativity and commutativity of the symbol involved, it follows from the theorem of Rohit Parikh that the positions of children in the derived term can be described by a semilinear set. Following results of Seymour Ginsburg and Edwin Spanier, these can be described by Presburger successor formulae. Hence the entire construction can be done inside $P_S MSO$, in fact, no universal second-order quantifiers are required.

For the converse, an induction can use the properties that regular languages are closed under boolean operations. For the Presburger successors, this is a theorem of Samuel Eilenberg and Marcel-Paul Schützenberger. Closure under projection requires nondeterministic automata which are introduced in Section 4. □

Andreas Potthoff showed that the language *EvalTrue* is not definable in first-order logic. If it were, the simpler formula which just evaluates the negation operation would be

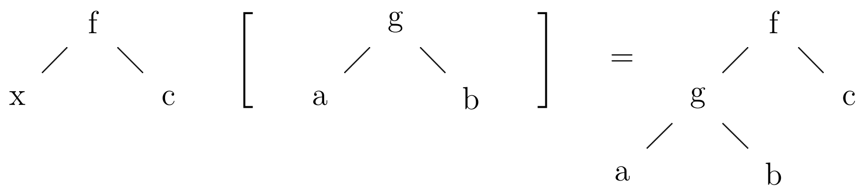

FIGURE 4. Substituting single occurrence of x in context $f(x, c)$ by term $g(a, b)$

definable, but evaluating this formula amounts to an odd/even parity check, which is not definable.

In this section it was shown that imposing a finite automaton structure by using homomorphisms into a finite set of states, allows the description of a set of terms by a logical formula. One can now import a lot of machinery from the theory of automata, which we rush through in the next section.

4. Regular languages

Michael Rabin and Dana Scott in a 1965 paper showed that one can also consider *nondeterministic* transition systems where there are multiple f-labelled joining hyperedges leaving a state tuple $\bar{q}$. To see this one considers, instead of Q, the powerset $\wp(Q)$ as the target algebra. Since the set of ranked trees T_Σ is the free Σ-algebra, the function $\delta : \Sigma_0 \to \wp(Q)$ and $\delta : \Sigma_n \to (Q^n \to \wp(Q))$ can be freely extended to the homomorphism $\delta^\# : T_\Sigma \to \wp(Q)$. For a nondeterministic automaton with final states $F \subseteq Q$, we check for acceptance of a run on t whether $\delta^\#(t) \cap F \neq \emptyset$. Eilenberg and Schützenberger showed such a construction which can be adapted for signatures with associative and commutative symbols.

Languages accepted by nondeterministic automata are closed under reversal of direction. Thus one can also define *top-down* nondeterministic automata, with f-labelled *forking* hyperedges, possibly many, leaving a state q and forking into an n-tuple of states for $f \in \Sigma_n$. Thus the transitions define a relation $\delta \subseteq (Q \times \Sigma_n \times Q^n)$, for constants one has *final combinations* $\delta \subseteq (Q \times \Sigma_0)$. For a top-down automaton with *initial* states $I \subseteq Q$, an accepting run on a tree must start from I and at every leaf, pass through a final combination (and not get stuck). Deterministic top-down automata accept less than the regular languages.

Exercise 6. *Show that any finite deterministic top-down automaton accepting the terms* $\{f(a, b), f(b, a)\}$ *where* $f \in \Sigma_2$ *and* $a, b \in \Sigma_0$*, must also accept the terms* $\{f(a, a), f(b, b)\}$*. Hence there is a finite regular language not accepted by such automata.*

As usual, to show that languages are not regular one uses a pumping theorem. We use it to show that the language $\{t \mid t\ is\ a\ full\ binary\ tree\ of\ height\ n,\ n \geq 1\}$ is not regular.

Theorem 7 (Pumping)**.** *For a finite ranked tree automaton with n states, if it accepts a tree of height $\geq n$, then there are two nodes on some path of the tree such that the tree can be decomposed $r[s[t]]$ at these nodes, where $r[\ ]$ and $s[\ ]$ are* singular contexts *and t is a tree, such that the trees $r[t]$, $r[s[s[t]]]$, $r[s[\ldots[s[t]]]] \ldots$, where the context s is iterated $i \geq 0$ times, are all accepted by the automaton.*

Corollary 8. *The language of all full binary trees is not regular.*

Proof. Suppose a tree automaton accepts a full binary tree of height n. Then it can be decomposed $r[s[t]]$ as in the pumping theorem. By the theorem, $r[s[s[t]]]$ is accepted, but this is not a full binary tree. □

For a nondeterministic term automaton one can check whether its language is nonempty (that is, it accepts at least one tree) by computing the set of states that are reached in a run on some tree. The algorithm goes up from the states reached by the leaf transitions and then by joining transitions, towards the root of an input tree. By intersecting with the final states one can check nonemptiness. There is a converse procedure for a top-down automaton.

Theorem 9. *There is an algorithm to check satisfiability of $P_S MSO$ formulae over trees.*

Proof. The proof from $P_S MSO$ formulae to automata is effective, and satisfiability reduces to nonemptiness of the accepted language. □

4.1. **Congruences of finite index.**

Two terms t and u are equivalent if they define the same homomorphism $h(t) = h(u)$ on Q. This is a finite-index congruence (called the *kernel* of h) studied by John Myhill and Anil Nerode in the 1950s. The relations which map to final states F in an automaton are a subset of its transition algebra, and the language accepted is $h^{-1}(F)$. The result is:

Theorem 10. *The regular languages are exactly those which are inverse images of morphisms into (designated subsets of) finite term algebras, as well as exactly those saturated by finite-index congruences, that is, unions of blocks of such congruences.*

Proof. The explanation above gives some idea of the forward directions. For the converse, the algebra itself (or the quotient by the congruence) can be used as the set of states and each function operation $f(q_1, \ldots, q_n) = q$, $f \in \Sigma$, gives a deterministic f-labelled transition. The designated subset gives the final states F. □

Myhill and Nerode showed that the congruences corresponding to finite automata form a lattice. For example, a product of two automata corresponds to an intersection of congruences. Therefore, given a regular language, there is a maximal saturating finite-index congruence for it, and hence a *minimal* deterministic finite automaton accepting it. Its transition system is called the *syntactic* algebra of the language.

Lodaya and Weil worked with the more general series-parallel posets, defining algebras and automata. Dietrich Kuske introduced *MSO* for series-parallel posets. There is an equivalence of the regular languages to $P_S MSO$, due to Nicolas Bedon.

At the level of term languages, including those over signatures with associative and commutative symbols, there is an effective equivalence between finite automata and definability in $P_S MSO$ logic which allows transfer of ideas and algorithms in both directions. This generalizes the equivalence over words of finite automata and definability in monadic second-order logic. The Presburger extension comes into play for Σ_{AC}, otherwise monadic second-order logic would suffice.

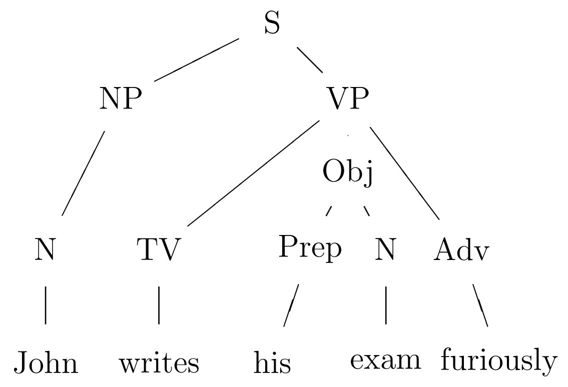

FIGURE 5. Parsing an English sentence

5. APPLICATIONS

5.1. **Yield and context-free languages.** CHOMSKY developed grammars for context-free languages in 1957. Deterministic pushdown automata for the parsing of natural language sentences were developed by IDA RHODES, MURRAY SHERRY AND ANTHONY OETTINGER in 1961. CHOMSKY AND SCHÜTZENBERGER connected the grammars to nondeterministic pushdown automata in 1963. Pushdowns are not examined in this article.

When tree automata were developed in 1965, it was immediately clear that they would accept parse trees which yield natural language sentences. Formally, the *yield* of a node of a tree, or a tree language, is the set of phrases or sentences formed from the subtree generated by that node, or a tree in the language, by traversing the frontier from left to right. This can be inductively generated by concatenating the leaves and ignoring the interior nodes. A *parse* of a sentence is a tree whose yield gives the sentence. A sentence can have multiple parses conforming to a single context-free grammar.

The finite set of *productions* (rules) below give a context-free grammar. Each symbol is called a *nonterminal.* They have a nonterminal on the left hand side and a sequence of nonterminals on the right. The abbreviation $N \to u_1 \mid \cdots \mid u_k$ for production number (Pi) stands for j productions $N \to u_1, \ldots, N \to u_j$ which are numbered $(Pi.1), \ldots, (Pi.j)$. There is also a *lexicon* relation which assigns constants to nonterminals which is left implicit here. For example, *John* is a noun, *writes* is a transitive verb, *furiously* is an adverb, and so on. The productions generate a parse tree top-down, with the lexicon being used at the bottom level. Figure 5 gives an example parse.

(1) $S \to NP\ VP$
(2) $NP \to Adj\ NP \mid NP$
(3) $VP \to TV\ Obj \mid TV\ Obj\ Adv \mid IV \mid IV\ Adv$
(4) $Obj \to Prep\ N \mid N$

The parse trees of this grammar are accepted by a bottom-up tree automaton with the constants Σ_0 (in this case all words in the English language are constants), and production numbers as function letters. For example, Σ_2 has $P1$ as a function symbol, Σ_3 has $P3.2$ and so on. The states of the tree automaton are the nonterminals. The final state required at the root of the parse is S.

$\delta(John) = N$, $\delta(writes) = TV$, $\delta(furiously) = Adv, \ldots,$
$\delta(P1)(NP, VP) = S$, $\delta(P2.1)(Adj, NP) = NP$, $\delta(P2.2)(NP) = NP$,
$\delta(P3.1)(TV, Obj) = VP$, $\delta(P3.2)(TV, Obj, Adv) = VP$, $\delta(P3.3)(IV) = VP$,
$\delta(P3.4)(IV, Adv) = VP$, $\delta(P4.1)(Prep, N) = Obj$, $\delta(P4.2)(N) = Obj$

Exercise 11. *Check that this idea works for any context-free grammar. Is the tree automaton constructed deterministic, or nondeterministic in general?*

Associative-commutative productions can be used in a context-free grammar, say:

$$S \to S\ Conj\ S, \text{ and } Conj \to then \mid and.$$

This could generate sentences like:

$$Teacher\ gives\ an\ exam\ paper,\ then\ John\ writes\ his\ exam.$$

Since the first production is associative and commutative, one can also generate:

$$John\ writes\ his\ exam,\ then\ Teacher\ gives\ an\ exam\ paper.$$

This is not useful in English, but may be useful in natural languages with free word order,

As an exercise to deal with yields, one can consider parse trees as being built on intervals of words. Intuitively, an *interval* $[b, e] = \{x \mid b \leq x \leq e\}$, where b, e are positions on a word, matches a phrase in the word which is the yield of a node in the parse. Thus the interval $[3, 4]$ marking the positions for the phrase "his exam" in the example corresponds to the node labelled Obj in the parse. Define this interval to be the *value* of the node Obj, and the *yield* of $[3, 4]$ to be "his exam". Another way of thinking about the interval structure is by inserting brackets: [[John][[writes][[his][exam]][furiously]]].

The *MSO* logic for words is well known, here is one for intervals on words. Intervals are modelled as second order variables Z which satisfy a connectedness property:

$$Intv(Z) = \forall x \forall y \forall z (x < y \wedge y < z \wedge Z(x) \wedge Z(z) \supset Z(y))$$

To reflect the parse tree structure, first of all a domain formula is required to say that the overlapping interval structure is *tree-like*:

$$\exists x (Z(x) \wedge Y(x) \wedge Intv(Z) \wedge Intv(Y)) \supset \forall x (Y(x) \supset Z(x)) \vee \forall x (Z(x) \supset Y(x))$$

The following atomic predicates, similar to those considered by Joseph Halpern and Yoav Shoham, are used. $S_B(Z, Y)$ and $S_E(Z, Y)$ say the interval Y is the first (last) child of interval Z in the parse. That is, Y is a beginning (ending) subinterval at the next level, there are no beginning (ending) subintervals between Z and Y (this last condition is stronger than in the Halpern-Shoham logic). $S_A(Z, Y_1, Y_2)$ says that, given that Y_1 is a child of Z in the parse, Y_2 is the next child of Z after it in the parse. This is called the "first child, next sibling, last child" interpretation of a tree.

By reflexive transitive closure $During(Z, Y)$ can be defined saying that Y is a descendant of Z in the parse, that is, it is a subinterval occurring during the course of the current interval. Following Halpern-Shoham logic one could define *Begins* and *Ends* as the reflexive transitive closures of S_B amd S_E. $Precedes(Z_1, Z_2)$ says that the interval Z_1 comes before Z_2 in a preorder traversal of the parse tree, thus $During(Z, Y)$ implies $Precedes(Z, Y)$. Its definition follows that of *precedes* on tree nodes which appeared earlier in this article.

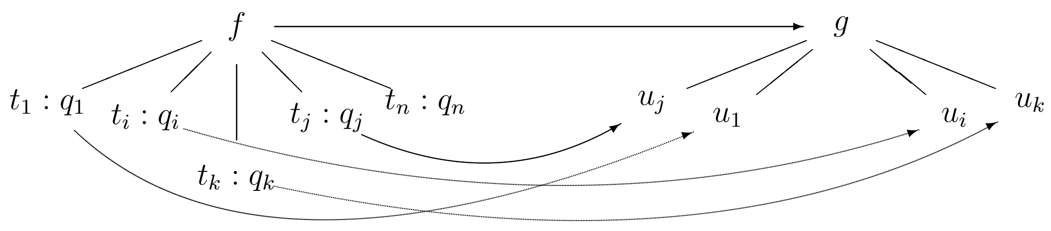

FIGURE 6. Building an output tree with $Val(f)(q_1, \dots, q_n) = g(x_j, x_1, x_i, x_k)$

Theorem 12. *The logic $MSO(S_B, S_E, S_A)$ on tree-like intervals of words is decidable.*

Proof. To formalize the interval logic as a tree-to-word *interpretation* of intervals of words in *MSO* logic on trees, translations are needed, as formalized in Section 6 below. The translation of $S_B(Z, Y)$ says that Y is a child of Z which precedes other children, that of $S_E(Z, Y)$ says that Y is a child of Z which is preceded by other children, and that of $S_A(Z, Y_1, Y_2)$ says that Y_1, Y_2 are children of Z which either precede or are preceded by other children. The monadic second-order variables for intervals translate to connected sets of leaf nodes with the linear order $<$ being represented by a *precedes* formula. The atomic predicates also have translations as tree formulae. Then Corollary 15 below gives decidability. □

5.2. **Transducers and the realization of tree homomorphisms.** A *deterministic bottom-up tree transducer* (Q, δ, Val, F) from Σ-terms to Γ-terms has, in addition to its states Q, final states $F \subseteq Q$ and transition function δ, an output function Val, for constants this is from $\Sigma_0 \to T_\Gamma$ and for function symbols we have linear $Val : \Sigma_i \to (Q^i \to T_\Gamma(X_i))$, where there is at most one occurrence of a variable on the right hand side.

A transducer works on a tree from the leaves upwards just like an automaton does. An additional thing happens, Figure 6 illustrates the description which follows. Suppose a bottom-up transition $\delta(f)(q_1, \dots, q_n)$ is taken from the n children of a node, the roots of the processed Σ-terms $t_1, \dots, t_n$, to the n-ary f-labelled parent, the root of the to-be-processed Σ-term $t = f(t_1, \dots, t_n)$. Then the linear Γ-term $Val(f)(q_1, \dots, q_n) = g(x_j, \dots, x_k)$, over variables $x_1, \dots, x_n$, is appended with the so far output Γ-terms $u_1, \dots, u_n$ for the processed terms $t_1, \dots, t_n$ (respectively), to form the to-be-output Γ-term $g(u_j, \dots, u_k)$, where the terms u_i are substituted for the variables $x_i, i = 1, n$. That is, the Γ-terms $u_i = Val^{\#}(t_i), i = 1, n$, are put together to form the Γ-term $u = Val^{\#}(t)$. There was an example immediately following Figure 2 describing the run of an automaton. (That does not work here because the output signature is not well-defined, this will be fixed soon and there is a proper example coming up.)

The function $\{(t \mapsto Val^{\#}(t)) \mid \delta^{\#}(t) \in F\}$ is called a *transduction* $T_\Sigma \to T_\Gamma$. It realizes the tree homomorphism given by $h(f(t_1, \dots, t_n)) = h(f)(x_1, \dots, x_n)$, where $x_i = h(t_i)$ and $h(f(|Z_1|, \dots, |Z_n|)) = h(f)(|Z_{i_1}|, \dots, |Z_{i_n}|)$. The book of BRUNO COURCELLE AND JOOST ENGELFRIET shows that such linear tree transductions can again be seen as *MSO* interpretations of the Γ-signature inside the Σ-signature. Their results extend to $P_S MSO$.

Here is an example transduction based on the one in Section 5.1. The grammar is a little expanded.

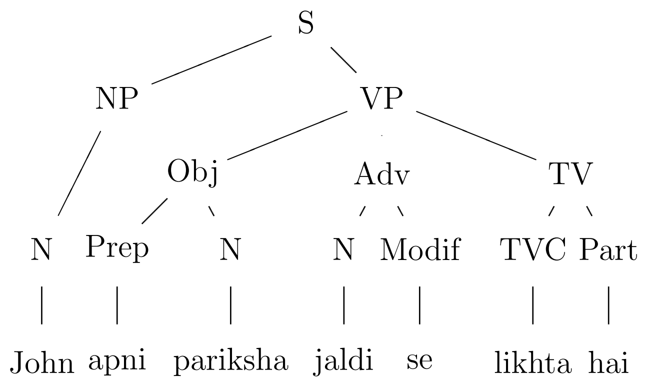

FIGURE 7. Putting out a Hindi sentence

(1) $S \to NP\ VP$
(2) $NP \to Adj\ NP \mid NP$
(3) $VP \to TV\ Obj \mid TV\ Obj\ Adv \mid IV \mid IV\ Adv$
(4) $Obj \to Prep\ N \mid N$
(5) $Adv \to N\ Modif$
(6) $TV \to TVC\ Part$

In this case we have in the Γ signature, $\Gamma_2 = \{S, NP, VP, Obj, TV, Adv\}$, $\Gamma_3 = \{VP\}$ and Γ_0 has all the words in the Hindi language, with obvious outputs, such as:

$$\begin{aligned}&Val(exam) = pariksha,\\ &Val(Adv) = Adv(x_1, x_2),\ Val(TV) = TV(x_1, x_2),\\ &Val(P3.4)(IV, Adv) = VP(x_2, x_1),\ Val(P3.2)(TV, Obj, Adv) = VP(x_2, x_3, x_1),\\ &Val(P4.1)(Prep, N) = Obj(x_1, x_2)\end{aligned}$$

These produce Γ-terms as expected. The output rules in the middle are more interesting, here phrases are rotated, as required for the translation from English to Hindi. Hindi makes slight use of counting, for example in a sentence like *Jab jab aap aayenge, tab tab aapko milenge.*

5.3. **Transformational grammars.**

CHOMSKY advocated transformational grammar in 1965 as a defining mechanism for linguistic structure. To illustrate this, the earlier example is converted from a declarative to an interrogative sentence.

(1) $S \to AdvQ\ NP\ VP$
(2) $NP \to Adj\ NP \mid NP$
(3) $VP \to TV\ Obj\ AdvTr \mid IV\ AdvTr$
(4) $Obj \to Prep\ N \mid N$
(5) $AdvQ \to AdvWh\ Aux$

Some of the transitions of the automaton for the parse tree are now:

$$\begin{aligned}&\delta(How) = AdvWh,\ \delta(does) = Aux,\\ &\delta(P3.2)(IV, AdvTr) = VP,\ \delta(P3.1)(TV, Obj, AdvTr) = VP\end{aligned}$$

CHOMSKY gave a short way of describing the move to an interrogative sentence by the transformation rule below, which transforms the sentence as shown in the figure. His 1965

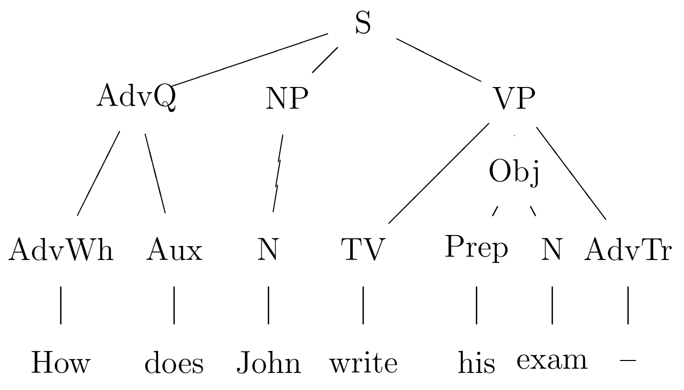

FIGURE 8. Making an interrogative sentence

book discusses many such transformations which apply in a natural language setting. Describing such tree transformations requires moving to richer notions of transduction.

$$S(NP, VP(TV, Obj, Adv)) \Longrightarrow S(AdvQ(AdvWh, Aux), NP, VP(TV, Obj, AdvTr))$$

This section discussed applications of the ideas seen earlier, from the domain of linguistics, inspired by the book of LAMBEK. There is a large computer science literature on transductions of various kinds. Some details are provided in the next section. Modelling transformations required in linguistics has not yet been fully addressed.

6. SEMANTIC INTERPRETATIONS INTO TREES

This section gives a formal basis for the study of tree transductions. Semantic interpretations are an old technique going back to the beginning of model theory, which was effectively used for showing decidability of theories by RABIN around 1970 after he proved that the *MSO* theory of the infinite tree is decidable. DETLEF SEESE AND COURCELLE sought to use this technique in the late 1980s for proving decidability of *MSO* theories of sets of graphs of so-called bounded treewidth, which is not defined here. The technique was to interpret them in tree languages. This has been developed in the book of COURCELLE AND ENGELFRIET.

The next definition is from the textbook of WILFRID HODGES. It is a bit of a mouthful because it is very general. The basic idea is that the "graph" is imagined inside a "tree". The "vertices" of the graph are the set defined by a "domain formula" with one free variable. The "edges" of the graph are defined by a "translation formula" $Tran(E)$ with two free variables. The "equality" of two graph vertices is defined by another "translation formula" $Tran(=)$ which should describe a congruence relation. The "coordinate map" shows how one talks about the graph inside the tree. The transduction Val maps the tree to the graph.

Definition 13. *An* ***interpretation*** *of a set $\mathcal{G}$ of graphs (Γ-structures) in a set $\mathcal{T}$ of trees (Σ-structures), alternatively of a Γ-theory Υ in a Σ-theory Θ, consists of:*

(1)
- *A* domain formula *(in Σ) Dom with one free variable;*
- Translation formulas *(in Θ): $Tran(E)$ and $Tran(=)$ each with two free variables, and $Tran(P_a), a \in A$, each with one free variable. The relation defined by $Tran(=)$ must be a congruence.*

For every Σ-structure T, these defining formulas *yield (upto isomorphism) a Γ-structure $G = (Dom, Tran(E), \{Tran(P_a), a \in A\})$ "inside" T.*

(2) *An onto* coordinate map *$c : Dom \to G$ such that for all $u, v \in Dom$,*
- $G \models E(c(u), c(v)) \iff T \models Tran(E)(u, v)$;
- $G \models c(u) = c(v) \iff T \models Tran(=)(u, v)$;
- $G \models P_a(c(u)) \iff T \models Tran(P_a)(u)$.

(3) *A* value map *Val, defined to map T to the quotient of G by $Tran(=)$, is required to be an onto function from $\mathcal{T}$ to $\mathcal{G}$. If the map is a tree homomorphism, we could specify Val and use its extension $Val^{\#}$ as the required map.*

Val is also known as a definable transduction or *reduction*. Note that it goes in the opposite direction to the translating formulae in the interpretation! If you think of Σ-structures as representing terms, Val gives the graph corresponding to a term. Going back to the tree transducers of Section 5.2, imagine now a tree-to-graph transducer running on an input tree and putting out not just a tree but a graph. The value map does the same thing, but no automata are necessary to build up $Val^{\#}$ from Val. Instead, $Tran$ induces a translation $Tran^{\#}$ from Γ-formulae to Σ-formulae:

$$Tran^{\#}(P_a(x)) = Dom(x) \wedge P_a(x)$$
$$Tran^{\#}(E(x_1, x_2)) = Dom(x_1) \wedge Dom(x_2) \wedge Tran(E)(x_1, x_2)$$
$$Tran^{\#}(t_1 = t_2) = Tran(=)(Tran^{\#}(t_1), Tran^{\#}(t_2))$$
$$Tran^{\#}(|X| + |Y| = |Z|) = |X| + |Y| = |Z|$$
$$Tran^{\#}(\neg\alpha) = \neg Tran^{\#}(\alpha)$$
$$Tran^{\#}(\alpha \vee \beta) = Tran^{\#}(\alpha) \vee Tran^{\#}(\beta)$$
$$Tran^{\#}(\exists x \alpha) = \exists x(Dom(x) \wedge Tran^{\#}(\alpha))$$
$$Tran^{\#}(\exists X \alpha) = \exists X(\forall y(X(y) \supset Dom(y)) \wedge Tran^{\#}(\alpha))$$

Theorem 14 (Reduction)**.** $T \models Tran^{\#}\alpha \iff Val(T) \models \alpha$.

Corollary 15 (Decidability)**.** *If there is an interpretation of $\mathcal{G}$ in $\mathcal{T}$ (with computable $Tran$), then $Th(\mathcal{T})$ is decidable implies $Th(\mathcal{G})$ is decidable.*

Proof. To decide satisfiability of Γ-formula α, compute $Tran^{\#}\alpha$ and return its satisfiability.

If the algorithm returns yes, $\exists T \in \mathcal{T} : T \models Tran^{\#}\alpha$. By reduction, $\mathcal{T}$ has $Val(T) \models \alpha$.

If the algorithm returns no and supposing $\exists G \in \mathcal{G} : G \models \alpha$, since Val is onto, there is $T \in \mathcal{T}$ such that $T \models Tran^{\#}\alpha$, a contradiction. ☐

6.1. **Evaluating definable problems.** Evaluating a formula on a given graph (this includes so-called model checking) can also be transferred. An interpretation is said to be *evaluation preserving polytime* if Val has a polynomial time converse function $Parse$ which identifies a representative term (parse tree) for each graph, and $f(G)(Val([a])) = \sum_{b \in [a]} f(Parse(G))(b)$.

Stefan Arnborg, Jens Lagergren and Seese used this to transfer efficient algorithms.

Theorem 16 (Polytime reduction)**.** *If $\mathcal{G}$ has an evaluation preserving polytime interpretation in $\mathcal{T}$, every Γ-definable evaluation problem π on $\mathcal{G}$ has a polynomial time algorithm.*

Proof. The algorithm computes $Parse(G)$ and $Tran^{\#}\pi$. Evaluating a $P_S MSO$-definable problem on a tree is linear in the size of the tree. $\square$

6.2. **An example: series-parallel posets.** A poset is a transitive directed acyclic graph. Let SP be the signature with $\Sigma_2 = \{\circ\}$ and $\Sigma_{AC} = \{||\}$, called "series" and "parallel" operations. Usually the series operation is required to be associative, but here we stick with the general tree signature. These objects are called the series-parallel posets (*sp-posets*), we define them by interpretation:

$Val(a)$ = singleton graph labelled by $a \in A$
$Val(G||H)$ = disjoint union of graphs G and H
$Val(G \circ H)$ = disjoint union and edges from all of G to all of H.

T_{SP} stands for SP-terms and G_{SP} for the range of Val on graphs. To show that Val from T_{SP} to G_{SP} is a $P_S MSO$-reduction, we must produce the (backwards) interpretation of sp-posets in trees. The defining Σ-formulas are:

$$Dom(x) = (leaf(x) \supset \bigvee_{a \in \Sigma_0} a(x)) \wedge (\neg leaf(x) \supset ||(x) \vee \circ(x)),$$

$Tran(a)(x) = a(x)$, $Tran(=)(x,y) = (x = y)$,
$Tran(E)(x,y)$ = the least common ancestor of x and y is a $\circ$-labelled node.

Every sp-poset has one or more term representations, so Val is onto; and one of them can be found in polynomial time. We can conclude that the $P_S MSO$ theory of sp-posets is decidable and $P_S MSO$-definable evaluation problems on sp-posets have polynomial time algorithms.

As mentioned earlier, the papers of Lodaya and Weil work directly with languages of sp-posets, which generalize trees. Thus they have algebras and automata directly running on these structures. Bedon provided a logical characterization, directly connecting this work to $P_S MSO$ logic. Following the ideas of $P_S MSO$ transductions presented in this section, it could also have been possible to work indirectly using automata on trees which are imagined to interpret sp-posets. This allows using the large number of tools which are available for tree automata.

This section presented the theory of transductions for the logic $P_S MSO$ that was used earlier.

7. Generalizing terms to straight-line programs

Terms were generalized to "jungles" by Annegret Habel, Hans-Jörg Kreowski and Detlef Plump to allow sharing of subterms. These ideas were used by Zoltán Ésik and Lodaya to model the basic transitions of Petri's nets, which are multi-output functions, taking a set of i elements as argument and returning a set of j elements as value. First signatures are generalized. Note that the associativity and commutativity requirements seen in Section 1 are dropped.

Definition 17. *A* ***signature*** Σ *consists of a finite set, associated with each element is its arity* $j \leftarrow i$, *more formally a* pre-arity i *and a* post-arity j.

If $f^{j\leftarrow i}$ is an i-to-j-ary function symbol from the signature, the "assignment command" $\{v_1,\ldots,v_j\} := f\{u_1,\ldots,u_i\}$ is called an $i\Sigma j$**-transition** (or Σ-transition). Here the $u_1,\ldots,u_i$ ("preconditions") and $v_1,\ldots,v_j$ ("postconditions") are *distinct variables*, which come from a set B.

The syntactic entities are called straight-line programs. The aim is to map them to runs of a Petri net just as terms get mapped to runs of an automaton.

Definition 18. *A **(straight-line) Σ-program** is a sequence of Σ-transitions, satisfying the following conditions:*

- *All the variables occurring on the right and left hand side of a transition must be distinct. (Sets of variables, not multisets.)*
- *The variables on the left hand side can only occur on the right hand side of later transitions. (A variable cannot be read and then written to.)*
- *A variable can be assigned to only once. (A variable cannot be overwritten.)*
- *A variable can appear only once on the right hand side. (Linearity.)*

The *input* variables of the program are those which appear on the right hand side but not on the left hand side of any transition in the program. The *output* variables are those which appear on the left hand side but not on the right. The rest of the variables are called *internal*, and programs upto renaming of internal variables are identified. The **arity** of a program is $o \leftarrow n$ if it has n input variables and o output variables. Such a program is called an $n\Sigma o$**-program**. We also use $n\Sigma$**-program** if all outputs are allowed. An $n\Sigma$**-**($n\Sigma o$**-)language** is a set of $n\Sigma$- ($n\Sigma o$-)programs.

Programs of arbitrary length are possible, independent of $|\Sigma|$, because of the unbounded number of variables available. In a sense, every "control point" of a program is named by a unique variable.

Here is an example program with a potential "run", which will be defined in the next section (along with the notation). Figure 9 has a hypergraph and a poset derived from the program, which are defined below.

$\{q_1, buf\} := f\{p_1\}$
$\{r_1\} := g\{q_1\}$
$\{q_2\} := g\{p_2\}$
$\{r_2\} := h\{buf, q_2\}$

$$\wr p_1, p_2 \wr \xrightarrow{g} \wr p_1, q_2 \wr \xrightarrow{f} \wr q_1, buf, q_2 \wr \xrightarrow{h} \wr q_1, r_2 \wr \xrightarrow{g} \wr q_2, r_2 \wr$$

7.1. **Hypergraph representation.** A Σ-program can be represented as a finite directed hypergraph $\mathcal{S} = (B, \{E_f | f \in \Sigma\})$, where B is the set of variables in the program, and each transition $\{v_1,\ldots,v_j\} := f\{u_1,\ldots,u_i\}$ is modelled by a directed hyperedge in E_f from the source nodes $u_1,\ldots,u_i$ to the target nodes $v_1,\ldots,v_j$ where f has arity $j \leftarrow i$. The conditions on a Σ-program ensure that the hypergraph is acyclic and *unbranched*, that is, two different hyperedges do not share source and target vertices. Renaming of variables yields an isomorphic hypergraph. Hypergraphs whose labelling respects the signature Σ are called Σ-*hypergraphs*.

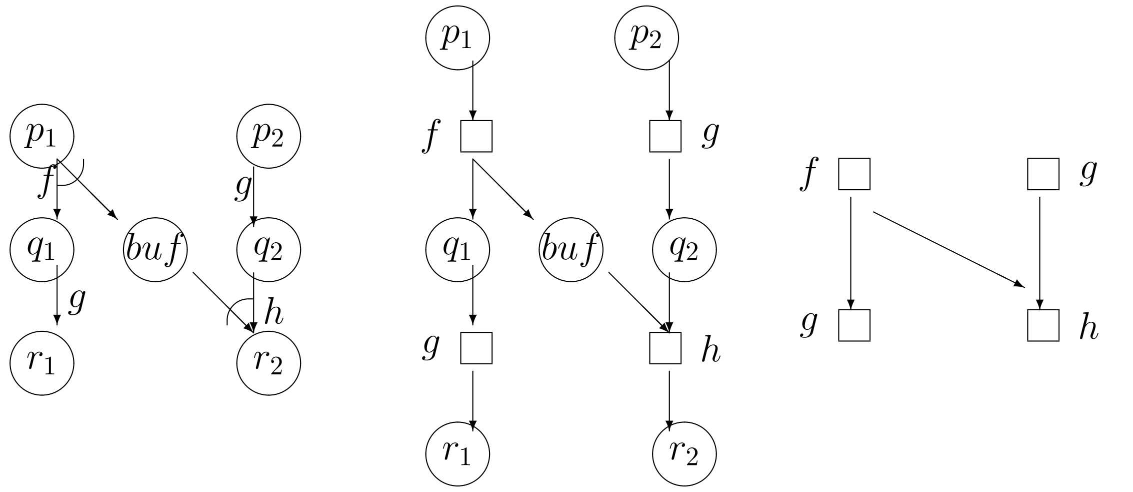

FIGURE 9. Hypergraph, bipartite graph and derived poset of a straight-line program

Conversely, from an acyclic unbranched Σ-hypergraph we can easily write a Σ-program corresponding to it by using its nodes as names of variables.

A directed hypergraph can in turn be represented as a bipartite directed graph $\mathcal{K} = (B, E, F, l)$ where B is as before a "sort" of nodes and E is a "sort" of transitions (hyperedges). The source and target nodes are connected by an incidence relation $F \subseteq (B \times E) \cup (E \times B)$ as follows: for the hyperedge e representing the transition above, there are edges in F from the source nodes $u_1, \ldots, u_i$ of the hyperedge to e, and from e to the target nodes $v_1, \ldots, v_j$ of the hyperedge. l labels nodes of sort E by letters from Σ: the vertex e of the graph above is labelled f.

Such graphs are known in the literature as *(labelled finite) causal nets*. B is the set of *basic conditions* of the net, E is the set of *events* and F is called the *flow* relation. A causal net is *unbranched*, that is, all its basic conditions b have at most one predecessor and one successor, and *acyclic*: the reflexive transitive closure of F is antisymmetric (that is, a partial order). Since $l : E \to \Sigma$ is an arity-respecting labelling, we call them *causal* Σ*-nets*. They give a purely relational representation of an acyclic unbranched Σ-hypergraph.

7.2. **Labelled posets.** A Σ-labelled poset $\mathcal{P} = (E, F^* \cap (E \times E), l)$ can be derived from an acyclic unbranched hypergraph. The picture above shows the poset corresponding to the program and hypergraph to its left.

For an element e of $\mathcal{P}$, let ${}^\bullet e$ and $e^\bullet$ stand for the immediate predecessors and successors of e in the Hasse diagram of $\mathcal{P}$. The arity of $l(e)$ in Σ is lower bounded by $|e^\bullet| \leftarrow |{}^\bullet e|$, but can be higher. An anonymous referee gave an example showing two programs with different signatures which map to the same poset. Suppose r is the transition $\{w_1, w_2\} := f\{v_1\}$ and s the transition $\{v_2\} := g\{w_1, w_2\}$. The arity of f and g is higher in $r; s$ than in $r'; s'$, where r' is $\{w\} := f\{v_1\}$ and s' is $\{v_2\} := g\{w\}$. But the derived poset is the same for both programs.

Conversely, from $\mathcal{P}$ we can define the hypergraph $(H, \{E_f | f \in \Sigma\})$, where H is the set of edges in the Hasse diagram of $\mathcal{P}$ together with the minimal and maximal elements, and E_f

takes ${}^\bullet e$ to $e^\bullet$ precisely when e is labelled f, and an obvious extension for the minimal and maximal elements.

The *width* of a poset is the size of its largest antichain. A Σ-language is said to be **bounded width** if there is a bound $k \in \mathbb{N}$ such that the width of the posets corresponding to each term in it is at most k.

7.3. **Composition.** Programs seem straightforward, but the difficulty lies in formalizing the *composition* $s;t$ of two programs (assuming the result is a program), which we have done in the example by sequencing t after s. Lodaya and Weil introduced the notion of a *series* Σ-*algebra* where one Σ-term t can be sequenced after another s. But the semantics there requires that the execution of s be complete before the execution of t begins.

Series semantics is inadequate to model the N-shaped execution in the example above. If s has postconditions $J \cup J_1$ and t takes preconditions J_1 to postconditions K, then their composition has postconditions $J \cup K$. Conversely, if we sequence a program t after s, we get a composition depending on the relationship between the postconditions of s and the preconditions of t. This was studied by Ursula Goltz and Wolfgang Reisig.

Simple projection operations π_k, $k \in \mathbb{N}$, do not work either. The names of the variables in J_1 are crucial to making the definition of composition unambiguous. But it is not *really* the names which are important, they are just "place-holders" !

Remark. Suppose s is the transition $\{w_1\} := f_1\{v_1\}$ and t the transition $\{w_2\} := f_2\{v_2\}$. We say the two programs are *independent* since they work with disjoint sets of variables. This example suggests representing programs by the traces of Antoni Mazurkiewicz. However, note that f_1 and f_2 may or may not be independent in different contexts, so independence is not coded into the action label (as would be the case for Mazurkiewicz traces). The two g actions in the picture above are independent.

Authors have grappled with this problem in different ways within the field denoted by Robin Milner as "process calculi". José Meseguer and Ugo Montanari and Ésik and Lodaya moved to categories, Gheorghe Ştefănescu used operators for permuting a tuple, Jos Baeten and Jan Bergstra introduced an explicit set of *causes* similar to the program variables above.

This section presented an approach to generalizing results on terms using straight-line programs, which allow sharing of subterms. The next section generalizes from algebras to the clones of equational theories of F. William Lawvere, providing the means to process programs.

8. Theories for concurrent behaviour

Ésik and Lodaya gave a categorical model following Meseguer and Montanari, extending earlier work on tree theories by Ésik and on finitary preclones by Ésik and Weil. This section presents this hitherto unpublished sketch of work. Lodaya then obtained a Myhill-Nerode result.

Definition 19. *A **concurrency theory** over signature Σ and variables B is a strictly symmetric monoidal category $\mathcal{C}$ with* objects *the finite nonempty multisets over B,* arrows *the Σ-programs and a commutative tensor functor $|| : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ whose operation over objects is multiset union and over arrows (programs) is their parallel product. The theory is said to **allow redistribution** if it also has* duplicating *arrows $\epsilon_{u,\{v,w\}}$ and* collapsing *arrows $\epsilon_{\{u,v\},w}$ for variables u, v, w subject to the conditions below. Using the identity, duplicating and collapsing arrows, we can define the arrows $\epsilon_{M,N}$, where M, N are any objects. These arrows are required to satisfy the equations $\epsilon_{M,N}\epsilon_{N,P} = \epsilon_{M,P}$ and $\epsilon_{M,N}||\epsilon_{P,Q} = \epsilon_{M||P,N||Q}$. A **theory morphism** from a concurrency theory to another is a functor which preserves the tensor, and the special arrows as well if the theory allows redistribution.*

The notation $\wr p, p, q, q, q \int$ is used to denote a multiset with 5 elements, against the usual $\{p, q\}$ for its base set with 2 elements. $||$ stands for multiset union.

Each $i\Sigma j$**-transition** $N := fM$, where we drop the distinctness condition on variables from the previous section, defines an arrow $M \xrightarrow{f} N$. Composition of Σ**-programs** is modelled by composition of arrows and parallel product. For example, if $M \xrightarrow{r} N_1||N$ and $N_1 \xrightarrow{s} P$, we get the arrow $M \xrightarrow{r(s||\epsilon_N)} P||N$ as their composition.

The main interest is morphisms which map into theories $\mathcal{F}$ which are *finitely generated*, where the set of variables is finite, as well as *locally finite*, where every hom-set $\mathcal{F}(M, N)$ is finite. We call a theory *finite* if it is finitely generated and locally finite.

Definition 20. *An $n\Sigma o$-language $L \subseteq \mathcal{C}(M, N)$ is a **recognizable language** if there is a morphism $\phi : \mathcal{C} \to \mathcal{F}$, $\mathcal{F}$ finite, such that $L = \phi^{-1}\phi(L)$. An $n\Sigma$-language is recognizable if each of its $n\Sigma o$ restrictions is recognizable.*

8.1. **Petri nets.** A finite concurrency theory allowing redistribution, with objects multisets (called *markings*) over a finite set of variables P (called *places*, each element of a marking is a "token" located at a place) and Σ-labelled atomic arrows T (called *transitions*) can be represented as a Σ-hypergraph $(P, \{T_f | f \in \Sigma\})$, which need be neither acyclic nor unbranched. As for automata, if there can be many transitions with the same label enabled at a marking, the labelled hypergraph is said to be *nondeterministic*, otherwise *deterministic*.

A relational representation of such a Σ-labelled hypergraph was called a *(Σ-labelled) net* by PETRI: a bipartite edge-weighted graph $\mathcal{N} = (P, T, W, \ell)$ where P is a set of *places*, T a (disjoint) set of *transitions*, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ defines a *weighted* flow relation which models hyperedges. The *labelling* ℓ is arity-respecting in the present article.

For a place or transition y, its pre-set $\{x \mid W(x, y) > 0\}$ (the sources of a transition) is conventionally denoted ${}^\bullet y$ and its post-set $\{z \mid W(y, z) > 0\}$ (the targets of a transition) is denoted $x^\bullet$. W satisfies the condition that for each transition a, ${}^\bullet a$ and $a^\bullet$ are nonempty, and for each place p, either ${}^\bullet p$ or $p^\bullet$ is nonempty. For all transitions a, the arity of $\ell(a)$ is from the indegree of a to its outdegree $(\sum_q W(a, q) \leftarrow \sum_p W(p, a))$. If a net satisfies this condition, we call it a Σ-*net*. The label of an unlabelled deterministic Petri net can be taken to be the signature of the transition.

Conversely, any (finite) Σ-net is a Σ-hypergraph and a deterministic net is a (finite) concurrency theory allowing redistribution, with objects its set of places and atomic arrows its set of transitions.

By specifying an initial multiset, a finite concurrency theory allowing redistribution can be represented as a Petri net system, a net with an initial marking. A set of final multisets can be provided as final markings. It is conventional for Petri nets to model system behaviours which go on as long as possible from a single initial marking.

Unlike general Σ-programs, Petri nets have a fixed interpretation. A marking is a multiset of places, fixing the domain as $\mathbb{N}^P$. The interpretation of every transition a is a fixed partial function $\mathbb{N}^{{}^\bullet a \times a^\bullet} \to \mathbb{N}^{{}^\bullet a \times a^\bullet}$ (which can be extended to the other places which remain unaffected), dependent only on its signature $i\Sigma j$. It is defined (a can be *fired*) when all pre-places p of a occur $\geq W(p,a)$ times in the marking, in which case the function removes $W(p,a)$ occurrences of each pre-place p from the marking and adds $W(a,q)$ occurrences of each post-place q, to obtain the target marking.

A tensor composition of arrows allows defining the *run* of the net system (or *firing sequence* of transition labels) from an initial marking, the enabling condition for each firing has to be met all along the run for the functions in the interpretation to be applied. Firing sequences are words which describe the behaviour of a net system and can be collected into a language. An example run of the net system in Figure 9 with initial marking $\wr p_1, p_2 \int$ was shown in the previous section.

8.2. ***k*-safe Petri nets.** The net system in Figure 9 is *1-safe*, it was sufficient to use sets rather than multisets for the markings. A net system is said to be ***k*-safe** if for all markings M reachable from the initial marking and for all places p, p occurs at most k times in M. As a consequence, the weight of the flow relation of a 1-safe net system can only be 0 or 1 and we can interpret it as a subset of $(P \times T) \cup (T \times P)$, markings M are then subsets of places P.

Figure 10 shows a 1-safe net system modelling a bank example, in its bipartite graph representation. Depending on the choice made, either a ticket is printed or cash is withdrawn. More sophisticated modelling would have a k-safe system where k is the cash storage capacity of the ATM and withdrawal of amounts upto k.

This article does not go into applications of Petri nets on which there is a large literature. A couple of favourite examples are the client-server choreographies modelled in logic by R. Ramanujam and S. Sheerazuddin, and the negotiations modelled as games by Jörg Desel and Javier Esparza. Recent theoretical work on 1-safe nets includes that on regular expressions by Ramachandra Phawade and Lodaya and on complexity by M. Praveen and Lodaya.

8.3. **Labelled posets as behaviour.** A *non-sequential process* of a net system $\mathcal{N}$, studied by Ursula Goltz and Wolfgang Reisig, is a more refined notion of behaviour than a firing sequence. A non-sequential process is a causal net $\mathcal{K} = (B, E, F, l)$ such that there are two functions $\pi : B \to P$ and $\pi : E \to T$ (the context will make clear which function is meant) satisfying the following properties:

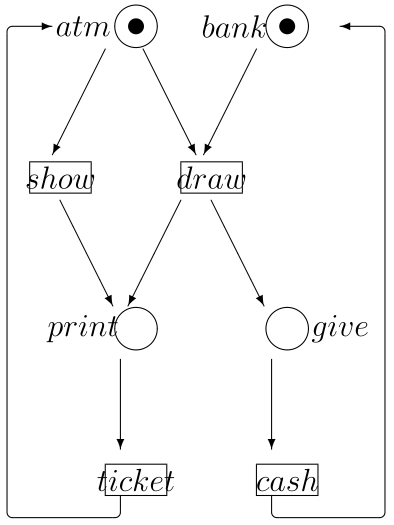

FIGURE 10. An ATM offering a choice of showing balance and withdrawing money

- for all e in E, $\pi({}^\bullet e)$ is in bijection with ${}^\bullet\pi(e)$, $\pi(e^\bullet)$ is in bijection with $\pi(e)^\bullet$ and $\ell(\pi(e)) = l(e)$ (transition neighbourhood and labelling respected);
- The initial marking is the multiset $\wr p \mid {}^\bullet b = \emptyset \ \ \& \ \ \pi(b) = p \int$ (minimal elements respect initial marking).

The bijection requirement can be relaxed for weighted nets. The definition of $\mathcal{K}$ as a non-sequential process of $\mathcal{N}$ is close to that of Σ-programs corresponding to $\mathcal{K}$ forming a morphism into the concurrency theory corresponding to $\mathcal{N}$. A graphical example was seen in Figure 3. The poset language of $\mathcal{N}$ is formed using the next definition, since a poset can be derived from a causal net.

Definition 21. *A net system $\mathcal{N}$* ***accepts*** *an $n\Sigma$-program t if a causal net for t is a non-sequential process of $\mathcal{N}$.*

Extending Theorem 5, MICHEL PARIGOT AND ELISABETH PELZ used *existential* second-order formulae (that is, no universal second-order quantifiers are used) of *MSO* logic with threshold counting comparisons for monadic predicates (not just for children of a node), call it $\exists$*ThrMSO*. This is more expressive than regular word languages, for example the formula $|a| = |b| \wedge |b| = |c|$ can describe the semilinear set of words with equal numbers of letters a, b, c. PARIGOT AND PELZ's proof extends to posets.

Theorem 22. *Word and poset languages of Petri nets can be defined using closed formulae of $\exists$ThrMSO over a linear order and over a partial order respectively.*

Proof. For each place p, use monadic predicates (similar to PARIGOT AND PELZ's E_p and S_p) for positions where a token is added to p and removed from p, respectively. Comparisons of counts of these predicates are used to ensure that a transition is firable. The firing sequence is coded as a linear order, the nonsequential process as a partial order. □

Parigot and Pelz proved a converse for sets of words using the fact that first-order word languages are regular, and regular languages are closed under complement. Bedon's $P_S MSO$ characterization for sets of sp-posets used the Eilenberg-Schützenberger result to prove closure under complement. For sets of posets in general this technique breaks down.

8.4. **Poset languages, bounded width and bounded layering.** A finite k-safe net system with set of places P can only accept a language with width bounded by $k|P|$, because its set of markings has maximum cardinality $k^{|P|}$. (If one only considers firing sequences, a k-safe net system accepts a regular language.) Conversely, if a net system accepts a bounded width poset language, then there is a k-safe net system accepting the same language, for some k.

Those Σ-programs which violate the k-safety condition in their execution are no longer valid. Thus at a marking M, one can list which Σ-transitions are possible in the net and which are valid but not present in the net. By using this property, a Rabin-Scott powerset construction and a translation of Eike Best and Harro Wimmel from k-safe nets to 1-safe nets, Lodaya obtained a Myhill-Nerode theorem for bounded width Σ-languages accepted by 1-safe Σ-nets. The corresponding poset languages can be seen as analogues of regular languages in a jungle setting, wider than that of words (and Mazurkiewicz traces), terms and series-parallel posets, seen earlier in this article. The development needs to be related to logic.

B. Meenakshi and Ramanujam defined languages of posets which have bounded or communication-closed layering, a notion defined by Tzilla Elrad and Nissim Francez which allows decomposition of posets into "episodic" structure. They also have a logic. Béatrice Bérard, Stefan Haar and Loïc Hélouët showed that such layering leads to the notion of bounded treewidth which appeared in graph theory and was briefly mentioned in Section 6. Tying up ideas in this direction in a general framework for hypergraphs and posets, as was done for graphs earlier, remains to be done.

9. Summary

This article began with the representation of terms as trees and algebras as automata. Closed formulae of $P_S MSO$ logic were used to define term languages, in other words, boolean functions of terms. From accepting or rejecting, automata were generalized to deterministic transducers which computed functions from terms to terms. An application to context-free grammars, seen as defining languages of parse trees, suggested possible uses in translation from one natural language to another. The model-theoretic technique of $P_S MSO$ transductions further allowed graph languages to be interpreted as term languages, with closed logical formulae directly defining graph languages. Towards the end, terms were generalized to programs which can be represented as hypergraphs or posets, allowing discussion of poset languages. The last section moved from algebras to Lawvere theories representable as Petri nets, which generalize automata. Closed formulae of $\exists ThrMSO$ define poset languages, but here the results need to be improved further.

In brief, the article presents a view articulated frequently in computing science, that the structures which are processed in computation and the processors that are used, be seen through a logical perspective.

References

Stefan Arnborg, Jens Lagergren and Detlef Seese. Easy problems for tree-decomposable graphs, *J. Algorithms* **12**2 (1991), pp 308–340.

Franz Baader and Tobias Nipkow. *Term rewriting and all that* , Cambridge University Press (1998).

Jos Baeten and Twan Basten. Partial order process algebra (and its relation to Petri nets), in *Handbook of process algebra* (Jan Bergstra, Alban Ponse and Scott Smolka, eds.) , Elsevier (2001), pp 769–872.

Nicolas Bedon. Logic and branching automata, *Log. Meth. Comp. Sci.* **11**,4:2 (2015), pp 1–38.

Béatrice Bérard, Stefan Haar and Loïc Hélouët. Hyper partial order logic, *Proc. 38th FSTTCS*, Ahmedabad (Sumit Ganguly and Paritosh Pandya, eds.) *Lipics* **122** (2018), pp 20:1–20:15.

Eike Best and Harro Wimmel. Reducing k-safe Petri nets to pomset-equivalent 1-safe Petri nets, *Proc. ATPN*, Aarhus (Mogens Nielsen and Dan Simpson, eds.), *LNCS* **1825** (2000), pp 146–165.

Richard Büchi. *Finite automata, their algebras and grammars: Towards a theory of formal expressions* (D. Siefkes, ed.) , Springer (1989).

Noam Chomsky. *Syntactic structures* , Mouton (1957).

Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages, in *Computer programming and formal systems* (P. Braffort and D. Hirschberg, eds.) , North-Holland (1963), pp 118–161.

Noam Chomsky. *Aspects of the theory of syntax* , MIT Press (1965).

Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison and Marc Tommassi. *Tree automata and their applications*, online at `www.grappa.univ-lille3.fr/tata`.

Bruno Courcelle. On context-free sets of graphs and their monadic second-order theory, in *Proc. 3rd Graph Grammars*, Warrenton (Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg and Azriel Rosenfeld, eds.), *LNCS* **291** (1986), pp 133–146.

Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic* , Cambridge University Press (2012).

Silvano Dal-Zilio and Denis Lugiez. XML schema, tree logic and sheaves automata, in *Proc. 14th RTA*, Valencia (Robert Nieuwenhuis, ed.), *LNCS* **2706** (2003), pp 246–263.

Jörg Desel and Javier Esparza. Negotiations and Petri nets, *Trans. Petri Nets Other Models Conc.* **XI**, in *LNCS* **9930** (2016), pp 203–225.

John Doner. Tree acceptors and some of their applications, *J. Comp. Syst. Sci.* **4** (1970), pp 406–451.

Samuel Eilenberg and Marcel-Paul Schützenberger. Rational sets in commutative monoids, *J. Alg.* **13**,2 (1969), pp 173–191.

Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication closed layers, *Sci. Comp. Program.* **2** (1982), pp 155–173.

Zoltán Ésik. A variety theorem for trees and theories, *Publ. Math. Debrecen* **54 Suppl.** (1999), pp 711–762.

Zoltán Ésik and Pascal Weil. Algebraic recognizability of regular tree languages, *Theoret. Comp. Sci.* **340** (2005), pp 291–321.

Zoltán Ésik and Kamal Lodaya. Notes on Petri nets and recognizability, manuscript (2005).

Gottlob Frege. *Begriffsschrift, a formula language, modelled upon that of arithmetic, for pure thought*, in van Heijenoort, pp 5–82.

Seymour Ginsburg and Edwin Spanier. Semigroups, Presburger formulas, and languages, *Pacific J. Math.* **16**,2 (1966), pp 285–296.

Kurt Gödel. On formally undecidable propositions of *Principa Mathematica* and related systems, *Monatshefte für Mathematik und Physik* **38** (1931), pp 173–198, in van Heijenoort, pp 596–616.

Ursula Goltz and Wolfgang Reisig. The non-sequential behaviour of Petri nets, *Inf. Comput.* **57** (1983), pp 125–147.

Annegret Habel, Hans-Jörg Kreowski and Detlef Plump. Jungle evaluation, *Fund. Inform.* **15**,1 (1991), pp 37–60.

Joseph Halpern and Yoav Shoham. A propositional modal logic of time intervals, *J. ACM* **38**,4 (1991), pp 935–962.

Jean van Heijenoort. *From Frege to Gödel: A source book in mathematical logic, 1879–1931*, Harvard University Press, 2nd ed. (1971).

Jacques Herbrand. On the consistency of arithmetic, in van Heijenoort, pp 618–628.

Wilfrid Hodges. *A shorter model theory*, Cambridge University Press (1997).

Deepak Kapur and G. Sivakumar. Architecture of and experiments with RRL, in *Proc. NSF workshop on RRL* (John Guttag, Deepak Kapur and David Musser, eds.), (1983), pp 33–56.

Nils Klarlund, Anders Møller and Michael Schwartzbach. Mona implementation secrets, in *Proc. CIAA 2000*, London(CA) (Sheng Yu and Andrei Paun, eds.), *LNCS* **2088** (2001), pp 182–194.

Dietrich Kuske. Towards a language theory for infinite N-free pomsets, *Theoret. Comp. Sci.* **299**,1-3 (2003), pp 347–386.

Joachim Lambek. The mathematics of sentence structure, *Amer. Math. Monthly* **65**,3 (1958), pp 154–170.

Joachim Lambek. *From word to sentence: a computational algebraic approach to grammar*, Polimetrika s.a.s. (2008).

F. William Lawvere. *Functorial semantics of algebraic theories*, PhD thesis, Columbia University (1963).

Kamal Lodaya and Pascal Weil. Series-parallel languages and the bounded width property, *Theoret. Comp. Sci.* **237**,1-2 (2000), pp 347–380.

Kamal Lodaya and Pascal Weil. Rationality in algebras with a series operation, *Inf. Comput.* **171**,2 (2001), pp 269–293.

Kamal Lodaya. Petri nets, event structures and algebra, in *Formal models, languages and applications* (K.G. Subramanian, K. Rangarajan and Madhavan Mukund, eds.), World Scientific (2006), pp 246–259.

Antoni Mazurkiewicz. Concurrent program schemes and their interpretation, *DAIMI* **PB-78**, Aarhus Univ (1977).

Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys.* **5** (1943), pp 115–133.

B. Meenakshi and R. Ramanujam. Reasoning about layered message passing systems, *Comp. Lang. Syst. Struct.* **30**,3-4 (2004), pp 171–206.

José Meseguer and Ugo Montanari. Petri nets are monoids, *Inform. Comput.* **88** (1990), pp 105–155.

Robin Milner. Calculi for interaction, *Acta Inform.* **33** (1996), pp 707–737.

John Myhill. *Finite automata and the representation of events*, WADD TR-57-624 (1957).

Anil Nerode. Linear automaton transformations, *Proc. AMS* **9**,4 (August 1958), pp 541–544.

Paritosh Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCvalid, in *Proc. RTtools*, Aalborg (2001).

Michel Parigot and Elisabeth Pelz. A logical approach of Petri net languages, *Theoret. Comp. Sci.* **39** (1985), pp 155–169.

Rohit Parikh. On context-free languages, *J. ACM* **13**,4 (Oct 1966), pp 570–581.

Carl-Adam Petri. Fundamentals of a theory of asynchronous information flow, *Proc. 2nd IFIP*, Munich (C.M. Popplewell, ed.), North-Holland (1962), pp 386–390.

Ramachandra Phawade and Kamal Lodaya. Kleene theorems for synchronous products with matching, *Trans. Petri Nets Other Models Conc.* **X**, in *LNCS* **9410** (2015), pp 84–108.

Andreas Potthoff. Modulo-counting quantifiers over finite trees, *Theoret. Comp. Sci.* **126**,1 (1994), pp 97–112.

M. Praveen and Kamal Lodaya. Parameterized complexity results for 1-safe Petri nets, in *Proc. Concur*, Aachen (Joost-Pieter Katoen and Barbara König, eds.), *LNCS* **6901** (2011), pp 358–372.

Mojżesz Presburger. On the completeness of certain systems of arithmetic of whole numbers in which addition occurs as the only operation, *Proc. Sprawozdaniez i kongresu matematykow krajow slowianskich*, Warsaw (1930), in *Hist. Philos. Logic* **12** (1991), pp 92–101.

Michael Rabin and Dana Scott. Finite automata and their decision problems, *IBM J. Res. Dev.* **3**,2 (April 1959), pp 114–125.

Michael Rabin. Decidability of second-order theories and automata on infinite trees, *Trans. AMS* **141** (July 1969), pp 1–35.

R. Ramanujam and S. Sheerazuddin. Realizable temporal logics for web service choreography, *J. Alg. Meth. Logic Program.* **85** (2016), pp 759–781.

Ida Rhodes. A new approach to the mechanical syntactic analysis of Russian, *Mech. Trans.* **6** (1961), pp 33–50.

Detlef Seese. Tree-partite graphs and the complexity of algorithms, in *Proc. 1st FCT*, Cottbus (Lothar Budach, ed.), *LNCS* **199** (1985), pp 412–421.

Helmut Seidl, Thomas Schwentick and Anca Muscholl. Counting in trees, in *Logic and automata* (Jörg Flum, Erich Grädel and Thomas Wilke, eds.) , Amsterdam University Press (2008), pp 575–612.

Murray Sherry and Anthony Oettinger. A new model of natural language for predictive syntactic analysis, in *Information theory* (Colin Cherry, ed.) , Butterworths (1961), pp 458–468.

Gheorghe Ştefănescu. *Network algebra* , Springer (2000).

Terese. *Term rewriting systems* (Marc Bezem, Jan Willem Klop and Roel de Vrijer, eds.), Cambridge University Press (2003).

James Thatcher and Jesse Wright. Generalized finite automata theory with an application to a decision problem of second-order logic, *Math. Syst. Th.* **2**,1 (March 1968), pp 57–81.

Wolfgang Thomas. *Applied automata theory*, course notes (2005).