View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

Logic Characterization of Invisibly Structured Languages: the Case of Floyd Languages

Violetta Lonati[1], Dino Mandrioli[2], Matteo Pradella[2]

[1] DI - Università degli Studi di Milano, via Comelico 39/41, Milano, Italy
lonati@di.unimi.it
[2] DEI - Politecnico di Milano, via Ponzio 34/5, Milano, Italy
{dino.mandrioli, matteo.pradella}@polimi.it

Abstract. Operator precedence grammars define a classical Boolean and deterministic context-free language family (called Floyd languages or FLs). FLs have been shown to strictly include the well-known Visibly Pushdown Languages, and enjoy the same nice closure properties. In this paper we provide a complete characterization of FLs in terms of a suitable Monadic Second-Order Logic. Traditional approaches to logic characterization of formal languages refer explicitly to the structures over which they are interpreted - e.g, trees or graphs - or to strings that are isomorphic to the structure, as in parenthesis languages. In the case of FLs, instead, the syntactic structure of input strings is "invisible" and must be reconstructed through parsing. This requires that logic formulae encode some typical context-free parsing actions, such as shift-reduce ones.
Keywords: Operator precedence languages, Deterministic Context-Free languages, Monadic Second-Order Logic, Pushdown automata.

1 Introduction

Floyd languages (FL), as we renamed Operator Precedence Languages and grammars (FG) after their inventor, were originally introduced to support deterministic parsing of programming and other artificial languages [1]; then, interest in them decayed for several decades, probably due to the advent of more expressive grammars, such as LR ones [2] which also allow for efficient deterministic parsing.

In another context Visual Pushdown Languages (VPL) -and other connected families e.g. [3]- have been introduced and investigated [4] with the main motivation to extend to them the same or similar automatic analysis techniques -noticeably, model checking- that have been so successful for regular languages. Recently we discovered that VPL are a proper subclass of FL, which in turn enjoy the same properties that make regular and VP languages amenable to extend to them typical model checking techniques; in fact, to the best of our knowledge, FL are the largest family closed w.r.t. Boolean operation, concatenation, Kleene * and other classical operations [5]. Another relevant feature of FL is their "locality property", i.e., the fact that partial strings can be parsed independently of the context in which they occur within a whole string. This enables more effective parallel and incremental parsing techniques than for other deterministic languages [6].

We also introduced an appropriate automata family that matches FGs in terms of generative power: Floyd Automata (FA) are reported in [7] and, with more details and precision, in [8]. In this paper we provide the "last tile of the puzzle", i.e., a complete characterization of FL in terms of a suitable Monadic Second-Order (MSO) logic, so that, as well as with regular languages, one can, for instance, state a language property by means of an MSO formula; then automatically verify whether a given FA accepts a language that enjoys that property.

In the literature various other classes of languages (including VPL) and structures (trees and graphs) have been characterized by means of MSO logic [9] by extending the original approach of Büchi (presented e.g. in [10]). To the best of our knowledge, however, all these approaches refer to a tree or graph structure which is explicitly available. In the case of FLs, instead, the syntax tree is not immediately visible in the string, hence a parsing phase is needed. In fact, Floyd automata are the only non-real-time automata we are aware of, characterized in terms of MSO logic.

The paper is structured as follows: Section 2 provides the necessary background about FL and their automata. Section 3 defines an MSO over strings and provides two symmetric constructions to derive an equivalent FA from an MSO formula and conversely. Section 4 offers some conclusion and hints for future work.

2 Preliminaries

FL are normally defined through their generating grammars [1]; in this paper, however, we characterize them through their accepting automata [8,7] which are the natural way to state equivalence properties with logic characterization. Nevertheless we assume some familiarity with classical language theory concepts such as context-free grammar, parsing, shift-reduce algorithm, syntax tree [2].

Let $\Sigma = \{a_1, \dots, a_n\}$ be an alphabet. The empty string is denoted ϵ. We use a special symbol # not in Σ to mark the beginning and the end of any string. This is consistent with the typical operator parsing technique that requires the look-back and look-ahead of one character to determine the next parsing action [2].

[]

Definition 1. *An operator precedence matrix (OPM) M over an alphabet Σ is a partial function $(\Sigma \cup \{\#\})^2 \to \{\lessdot, \doteq, \gtrdot\}$, that with each ordered pair (a, b) associates the OP relation $M_{a,b}$ holding between a and b. We call the pair (Σ, M) an* operator precedence alphabet *(OP). Relations $\lessdot, \doteq, \gtrdot$, are named yields precedence, equal in precedence, takes precedence, respectively. By convention, the initial # can only yield precedence, and other symbols can only take precedence on the ending #.*

If $M_{a,b} = \circ$, where $\circ \in \{\lessdot, \doteq, \gtrdot\}$, we write $a \circ b$. For $u, v \in \Sigma^*$ we write $u \circ v$ if $u = xa$ and $v = by$ with $a \circ b$. M is *complete* if $M_{a,b}$ is defined for every a and b in Σ. Moreover in the following we assume that M is *acyclic*, which means that $c_1 \doteq c_2 \doteq \dots \doteq c_k \doteq c_1$ does not hold for any $c_1, c_2, \dots c_k \in \Sigma, k \geq 1$. See [11,5,8] for a discussion on this hypothesis.

Definition 2. *A nondeterministic* Floyd automaton *(FA) is a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ where: (Σ, M) is a precedence alphabet; Q is a set of states (disjoint from Σ); $I, F \subseteq Q$*

are sets of initial and final states, respectively; $\delta : Q \times (\Sigma \cup Q) \to 2^Q$ *is the transition function.*

The transition function is the union of two disjoint functions: $\delta_{\text{push}} : Q \times \Sigma \to 2^Q$, and $\delta_{\text{flush}} : Q \times Q \to 2^Q$. A nondeterministic FA can be represented by a graph with Q as the set of vertices and $\Sigma \cup Q$ as the set of edge labelings: there is an edge from state q to state p labelled by $a \in \Sigma$ if and only if $p \in \delta_{push}(q, a)$ and there is an edge from state q to state p labelled by $r \in Q$ if and only if $p \in \delta_{flush}(q, r)$. To distinguish flush transitions from push transitions we denote the former ones by a double arrow.

To define the semantics of the automaton, we introduce some notations. We use letters $p, q, p_i, q_i, \ldots$ for states in Q and we set $\Sigma' = \{a' \mid a \in \Sigma\}$; symbols in Σ' are called *marked* symbols. Let $\Gamma = (\Sigma \cup \Sigma' \cup \{\#\}) \times Q$; we denote symbols in Γ as $[a\ q]$, $[a' q]$, or $[\#\ q]$, respectively. We set $smb([a\ q]) = smb([a' q]) = a$, $smb([\#\ q]) = \#$, and $st([a\ q]) = st([a' q]) = st([\#\ q]) = q$.

A *configuration* of a FA is any pair $C = \langle B_1 B_2 \ldots B_n,\ a_1 a_2 \ldots a_m \rangle$, where $B_i \in \Gamma$ and $a_i \in \Sigma \cup \{\#\}$. The first component represents the contents of the stack, while the second component is the part of input still to be read.

A computation is a finite sequence of moves $C \vdash C_1$; there are three kinds of moves, depending on the precedence relation between $smb(B_n)$ and a_1:

(push) if $smb(B_n) \doteq a_1$ then $C_1 = \langle B_1 \ldots B_n [a_1\ q],\ a_2 \ldots a_m \rangle$, with $q \in \delta_{push}(st(B_n), a_1)$;
(mark) if $smb(B_n) \lessdot a_1$ then $C_1 = \langle B_1 \ldots B_n [a_1' q],\ a_2 \ldots a_m \rangle$, with $q \in \delta_{push}(st(B_n), a_1)$;
(flush) if $smb(B_n) \gtrdot a_1$ then let i the greatest index such that $smb(B_i) \in \Sigma'$.
$C_1 = \langle B_1 \ldots B_{i-2} [smb(B_{i-1})\ q],\ a_1 a_2 \ldots a_m \rangle$, with $q \in \delta_{flush}(st(B_n), st(B_{i-1}))$.

Finally, we say that a configuration $[\#\ q_I]$ is *starting* if $q_I \in I$ and a configuration $[\#\ q_F]$ is *accepting* if $q_F \in F$. The language accepted by the automaton is defined as:

$$L(\mathcal{A}) = \left\{ x \mid \langle [\#\ q_I],\ x\# \rangle \overset{*}{\vdash} \langle [\#\ q_F],\ \# \rangle, q_I \in I, q_F \in F \right\}.$$

Notice that transition function δ_{push} is used to perform both push and mark moves. To distinguish them, we need only to remember the last symbol read (i.e., the *look-back*), encoding such an information into the states. Hence, in the graphical representation of a FA we will use a bold arrow to denote mark moves in the state diagram.

The deterministic version of FA is defined along the usual lines.

Definition 3. *A FA is deterministic if I is a singleton, and the ranges of δ_{push} and δ_{flush} are both Q rather than 2^Q.*

In [8] we proved in a constructive way that nondeterministic FA have the same expressive power as the deterministic ones and both are equivalent to the original Floyd grammars.

Example 1. We define here the stack management of a simple programming language that is able to handle nested exceptions. For simplicity, there are only two procedures, called a and b. Calls and returns are denoted by $call_a$, $call_b$, ret_a, ret_b, respectively. During execution, it is possible to install an exception handler hnd. The last signal that we use is rst, that is issued when an exception occurs, or after a correct execution to

	$call_a$	ret_a	$call_b$	ret_b	hnd	rst	#
$call_a$	$\lessdot$	$\doteq$	$\lessdot$		$\lessdot$	$\gtrdot$	
ret_a	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
$call_b$	$\lessdot$		$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$	
ret_b	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
hnd	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$		$\doteq$	
rst	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
#	$\lessdot$		$\lessdot$		$\lessdot$		$\doteq$

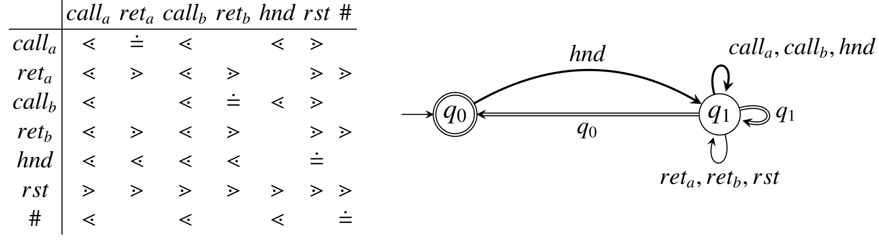


Fig. 1. Precedence matrix and automaton of Example 1.

uninstall the handler. With a *rst* the stack is "flushed", restoring the state right before the last *hnd*. Every *hnd* not installed during the execution of a procedure is managed by the OS. We require also that procedures are called in an environment controlled by the OS, hence calls must always be performed between a *hnd*/*rst* pair (in other words, we do not accept *top-level* calls). The automaton modeling the above behavior is presented in Figure 1. Note that every arrow labeled with *hnd* is bold as it represents a mark transition. An example run and the corresponding tree are presented in Figure 2.

Such a language is not a VPL but somewhat extends their rationale: in fact, whereas VPL allow for unmatched parentheses only at the beginning of a sentence (for returns) or at the end (for calls), in this language we can have unmatched $call_a$, $call_b$, ret_a, ret_b within a pair *hnd*, *rst*.

Definition 4. *A* simple chain *is a string* $c_0 c_1 c_2 \ldots c_\ell c_{\ell+1}$, *written as* ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$, *such that:* $c_0, c_{\ell+1} \in \Sigma \cup \{\#\}$, $c_i \in \Sigma$ *for every* $i = 1, 2, \ldots \ell$, *and* $c_0 \lessdot c_1 \doteq c_2 \ldots c_{\ell-1} \doteq c_\ell \gtrdot c_{\ell+1}$.

A composed chain *is a string* $c_0 s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell c_{\ell+1}$, *where* ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$ *is a simple chain, and* $s_i \in \Sigma^*$ *is the empty string or is such that* ${}^{c_i}[s_i]^{c_{i+1}}$ *is a chain (simple or composed), for every* $i = 0, 1, \ldots, \ell$. *Such a composed chain will be written as* ${}^{c_0}[s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell]^{c_{\ell+1}}$.

A string $s \in \Sigma^*$ *is* compatible *with the OPM M if* ${}^{\#}[s]^{\#}$ *is a chain.*

Definition 5. *Let* $\mathcal{A}$ *be a Floyd automaton. We call a* support *for the simple chain* ${}^{c_0}[c_1 c_2 \ldots c_\ell]^{c_{\ell+1}}$ *any path in* $\mathcal{A}$ *of the form*

$$q_0 \xrightarrow{c_1} q_1 \longrightarrow \ldots \longrightarrow q_{\ell-1} \xrightarrow{c_\ell} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1} \tag{1}$$

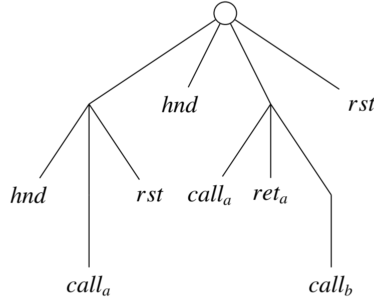
Notice that the label of the last (and only) flush is exactly q_0, *i.e. the first state of the path; this flush is executed because of relation* $c_\ell \gtrdot c_{\ell+1}$.

We call a support for the composed chain ${}^{c_0}[s_0 c_1 s_1 c_2 \ldots c_\ell s_\ell]^{c_{\ell+1}}$ *any path in* $\mathcal{A}$ *of the form*

$$q_0 \overset{s_0}{\rightsquigarrow} q'_0 \xrightarrow{c_1} q_1 \overset{s_1}{\rightsquigarrow} q'_1 \xrightarrow{c_2} \ldots \xrightarrow{c_\ell} q_\ell \overset{s_\ell}{\rightsquigarrow} q'_\ell \overset{q'_0}{\Longrightarrow} q_{\ell+1} \tag{2}$$

where, for every $i = 0, 1, \ldots, \ell$:

- *if* $s_i \neq \epsilon$, *then* $q_i \overset{s_i}{\rightsquigarrow} q'_i$ *is a support for the chain* ${}^{c_i}[s_i]^{c_{i+1}}$, *i.e., it can be decomposed as* $q_i \overset{s_i}{\rightsquigarrow} q''_i \overset{q_i}{\Longrightarrow} q'_i$.



	$\langle [\#\, q_0]$	, $hnd\ call_a\ rst\ hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
mark	$\langle [\#\, q_0][hnd'\ q_1]$	, $call_a\ rst\ hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
mark	$\langle [\#\, q_0][hnd'\ q_1][call'_a\ q_1]$	, $rst\ hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
flush	$\langle [\#\, q_0][hnd'\ q_1]$	, $rst\ hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
push	$\langle [\#\, q_0][hnd'\ q_1][rst\ q_1]$	, $hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
flush	$\langle [\#\, q_0]$	, $hnd\ call_a\ ret_a\ call_b\ rst\#\rangle$
mark	$\langle [\#\, q_0][hnd'\ q_1]$	, $call_a\ ret_a\ call_b\ rst\#\rangle$
mark	$\langle [\#\, q_0][hnd'\ q_1][call'_a\ q_1]$	, $ret_a\ call_b\ rst\#\rangle$
push	$\langle [\#\, q_0][hnd'\ q_1][call'_a\ q_1][ret_a\ q_1]$	, $call_b\ rst\#\rangle$
mark	$\langle [\#\, q_0][hnd'\ q_1][call'_a\ q_1][ret_a\ q_1][call'_b\ q_1]$	, $rst\#\rangle$
flush	$\langle [\#\, q_0][hnd'\ q_1][call'_a\ q_1][ret_a\ q_1]$	, $rst\#\rangle$
flush	$\langle [\#\, q_0][hnd'\ q_1]$	, $rst\#\rangle$
push	$\langle [\#\, q_0][hnd'\ q_1][rst\ q_1]$	, $\#\rangle$
flush	$\langle [\#\, q_0]$	, $\#\rangle$

Fig. 2. Example run and corresponding tree of the automaton of Example 1.

– *if* $s_i = \epsilon$, *then* $q'_i = q_i$.

Notice that the label of the last flush is exactly q'_0.

The chains fully determine the structure of the parsing of any automaton over (Σ, M). Indeed, if the automaton performs the computation $\langle [a\, q_0],\ sb\rangle \overset{*}{\vdash} \langle [a\, q],\ b\rangle$, then ${}^a[s]^b$ is necessarily a chain over (Σ, M) and there exists a support like (2) with $s = s_0 c_1 \ldots c_\ell s_\ell$ and $q_{\ell+1} = q$.

Furthermore, the above computation corresponds to the parsing by the automaton of the string $s_0 c_1 \ldots c_\ell s_\ell$ within the context a,b. Notice that such context contains all information needed to build the subtree whose frontier is that string. This is a distinguishing feature of FL, not shared by other deterministic languages: we call it the *locality principle* of Floyd languages.

Example 2. With reference to the tree in Figure 1, the parsing of substring $hnd\ call_a\ rst\ hnd$ is given by computation $\langle [\#\, q_0]\ ,\ hnd\ call_a\ rst\ hnd\rangle \overset{*}{\vdash} \langle [\#\, q_0]\ ,\ hnd\rangle$ which corresponds to support $q_0 \xrightarrow{hnd} q_1 \xrightarrow{call_a} q_1 \overset{q_1}{\Longrightarrow} q_1 \xrightarrow{rst} q_1 \overset{q_0}{\Longrightarrow} q_0$ of the composed chain ${}^{\#}[hnd\ call_a\ rst]^{hnd}$.

Definition 6. *Given the OP alphabet* (Σ, M)*, let us consider the FA* $\mathcal{A}(\Sigma, M) = \langle \Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max} \rangle$ *where* $\delta_{max}(q, q) = q$*, and* $\delta_{max}(q, c) = q$*,* $\forall c \in \Sigma$*. We call* $\mathcal{A}(\Sigma, M)$ *the* Floyd Max-Automaton *over* Σ, M*.*

For a max-automaton $\mathcal{A}(\Sigma, M)$ each chain has a support; since there is a chain $^{\#}[s]^{\#}$ for any string s compatible with M, a string is accepted by $\mathcal{A}(\Sigma, M)$ iff it is compatible with M. Also, whenever M is complete, each string is compatible with M, hence accepted by the max-automaton. It is not difficult to verify that a max-automaton is equivalent to a max-grammar as defined in [11]; thus, when M is complete both the max-automaton and the max-grammar define the universal language Σ^* by assigning to any string the (unique) structure compatible with the OPM.

In conclusion, given an OP alphabet, the OPM M assigns a structure to any string in Σ^*; unlike parentheses languages such a structure is not visible in the string, and must be built by means of a non-trivial parsing algorithm. A FA defined on the OP alphabet selects an appropriate subset within the "universe" of strings compatible with M. In some sense this property is yet another variation of the fundamental Chomsky-Shützenberger theorem.

3 Logic characterization of FL

We are now ready to provide a characterization of FL in terms of a suitable Monadic Second Order (MSO) logic in the same vein as originally proposed by Büchi for regular languages and subsequently extended by Alur and Madhusudan for VPL. The essence of the approach consists in defining language properties in terms of relations between the positions of characters in the strings: first order variables are used to denote positions whereas second order ones denote subsets of positions; then, suitable constructions build an automaton from a given formula and conversely, in such a way that formula and corresponding automaton define the same language. The extension designed by [4] introduced a new basic binary predicate $\leadsto$ in the syntax of the MSO logic, $x \leadsto y$ representing the fact that in positions x and y two matching parentheses –named call and return, respectively in their terminology– are located. In the case of FL, however, we have to face new problems.

- Both finite state automata and VPA are real-time machines, i.e., they read one input character at every move; this is not the case with more general machines such as FA, which do not advance the input head when performing flush transitions, and may also apply many flush transitions before the next push or mark which are the transitions that consume input. As a consequence, whereas in the logic characterization of regular and VP languages any first order variable can belong to only one second order variable representing an automaton state, in this case –when the automaton performs a flush– the same position may correspond to different states and therefore belong to different second-order variables.
- In VPL the $\leadsto$ relation is one-to-one, since any call matches with only one return, if any, and conversely. In FL, instead the same position y can be "paired" with different positions x in correspondence of many flush transitions with no push/mark

in between, as it happens for instance when parsing a derivation such as $A \overset{*}{\Rightarrow} \alpha^k A$, consisting of k immediate derivations $A \Rightarrow \alpha A$; symmetrically the same position x can be paired with many positions y.

In essence our goal is to formalize in terms of MSO formulas a complete parsing algorithm for FL, a much more complex algorithm than it is needed for regular and VP languages. The first step to achieve our goal is to define a new relation between (first order variables denoting) the positions in a string.

In some sense the new relation formalizes structural properties of FL strings in the same way as the VPL $\leadsto$ relation does for VPL; the new relation, however, is more complex than its VPL counterpart in a parallel way, as FL are richer than VPL.

Definition 7. *Consider a string $s \in \Sigma^*$ and a OPM M. For $0 \leq x < y \leq |s| + 1$, we write $x \curvearrowright y$ iff there exists a sub-string of $\#s\#$ which is a chain ${}^a[r]^b$, such that a is in position x and b is in position y.*

Example 3. With reference to the string of Figure 1, we have $1 \curvearrowright 3$, $0 \curvearrowright 4$, $6 \curvearrowright 8$, $4 \curvearrowright 8$, and $0 \curvearrowright 9$. In the parsing of the string, these pairs correspond to contexts where a reduce operation is executed (they are listed according to their execution order). For instance, the pair $6 \curvearrowright 8$ is the context for the reduction of the last $call_b$, whereas $4 \curvearrowright 8$ encloses $call_a$ ret_a $call_b$.

In general $x \curvearrowright y$ implies $y > x+1$, and a position x may be in such a relation with more than one position and vice versa. Moreover, if s is compatible with M, then $0 \curvearrowright |s|+1$.

3.1 A Monadic Second-Order Logic over Operator Precedence Alphabets

Let (Σ, M) be an OP alphabet. According to Definition 7 it induces the relation $\curvearrowright$ over positions of characters in any words in Σ^*. Let us define a countable infinite set of first-order variables $x, y, \ldots$ and a countable infinite set of monadic second-order (set) variables $X, Y, \ldots$.

Definition 8. *The $MSO_{\Sigma,M}$ (*monadic second-order logic *over (Σ, M)) is defined by the following syntax:*

$$\varphi := a(x) \mid x \in X \mid x \leq y \mid x \curvearrowright y \mid x = y + 1 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

where $a \in \Sigma$, x, y are first-order variables and X is a set variable.

$MSO_{\Sigma,M}$ formulae are interpreted over (Σ, M) strings and the positions of their characters in the following natural way:

- first-order variables are interpreted over positions of the string;
- second-order variables are interpreted over sets of positions;
- $a(x)$ is true iff the character in position x is a;
- $x \curvearrowright y$ is true iff x and y satisfy Definition 7;
- the other logical symbols have the usual meaning.

A sentence is a formula without free variables. The language defined by φ is $L(\varphi) = \{s \in \Sigma^* \mid \#s\# \models \varphi\}$ where $\models$ is the standard satisfaction relation.

Example 4. Consider the language of Example 1, with the structure implied by its OPM. The following sentence defines it:

$$\forall z \left(\begin{pmatrix} call_a(z) \vee ret_a(z) \\ \vee \\ call_b(z) \vee ret_b(z) \end{pmatrix} \Rightarrow \exists x, y \begin{pmatrix} x \curvearrowright y \wedge x < z < y \\ \wedge \\ hnd(x+1) \wedge rst(y-1) \end{pmatrix} \right).$$

Example 5. Consider again Example 1. If we want to add the additional constraint that procedure b cannot directly install handlers (e.g. for security reasons), we may state it through the following formula:

$$\forall z\,(hnd(z) \Rightarrow \neg \exists u\,(call_b(u) \;\wedge\; (u + 1 = z \vee u \curvearrowright z)))$$

We are now ready for the main result.

Theorem 1. *A language L over (Σ, M) is a FL if and only if there exists a $MSO_{\Sigma,M}$ sentence φ such that $L = L(\varphi)$.*

The proof is constructive and structured in the following two subsections.

3.2 From $MSO_{\Sigma,M}$ to Floyd automata

This part of the construction follows the lines of Thomas [10], with some extra technical difficulties due to the need of preserving precedence relations.

Proposition 1. *Let (Σ, M) be an operator precedence alphabet and φ be an $MSO_{\Sigma,M}$ sentence. Then $L(\varphi)$ can be recognized by a Floyd automaton over (Σ, M).*

Proof. The proof is composed of two steps: first the formula is rewritten so that no predicate symbols nor first order variables are used; then an equivalent FA is built inductively.

Let Σ be $\{a_1, a_2, \dots, a_n\}$. For each predicate symbol a_i we introduce a fresh set variable X_i, therefore formula $a_i(x)$ will be translated into $x \in X_i$. Following the standard construction of [10], we also translate every first order variable into a fresh second order variable with the additional constraint that the set it represents contains exactly one position.

Let φ' be the formula obtained from φ by such a translation and consider any subformula ψ of φ': let $X_1, X_2, \dots, X_n, X_{n+1}, \dots X_{n+m(\psi)}$ be the (second order) variables appearing in ψ. Recall that $X_1, \dots, X_n$ represent symbols in Σ, hence they are never quantified.

As usual we interpret formulae over strings; in this case we use the alphabet

$$\Lambda(\psi) = \left\{\alpha \in \{0,1\}^{n+m(\psi)} \mid \exists! i \text{ s.t. } 1 \le i \le n,\ \alpha_i = 1\right\}$$

A string $w \in \Lambda(\psi)^*$, with $|w| = \ell$, is used to interpret ψ in the following way: the projection over the j-th component of $\Lambda(\psi)$ gives an evaluation $\{1, 2, \dots, \ell\} \to \{0, 1\}$ of X_j, for every $1 \le j \le n + m(\psi)$.

For any $\alpha \in \Lambda(\psi)$, the projection of α over the first n components encodes a symbol in Σ, denoted as $symb(\alpha)$. The matrix M over Σ can be naturally extended to the OPM $M(\psi)$ over $\Lambda(\psi)$ by defining $M(\psi)_{\alpha,\beta} = M_{symb(\alpha),symb(\beta)}$ for any $\alpha, \beta \in \Lambda(\psi)$.

We now build a FA $\mathcal{A}$ equivalent to φ'. The construction is inductive on the structure of the formula: first we define the FA for all atomic formulae. We give here only the construction for $\curvearrowright$, since for the other ones the construction is standard and is the same as in [10].

Figure 3 represents the FA for atomic formula $\psi = X_i \curvearrowright X_j$ (notice that $i, j > n$). For the sake of brevity, we use notation $[X_i]$ to represent the set of all tuples $\Lambda(\psi)$ having the i-th component equal to 1; notation $[\bar{X}]$ represents the set of all tuples in $\Lambda(\psi)$ having both i-th and j-th components equal to 0.

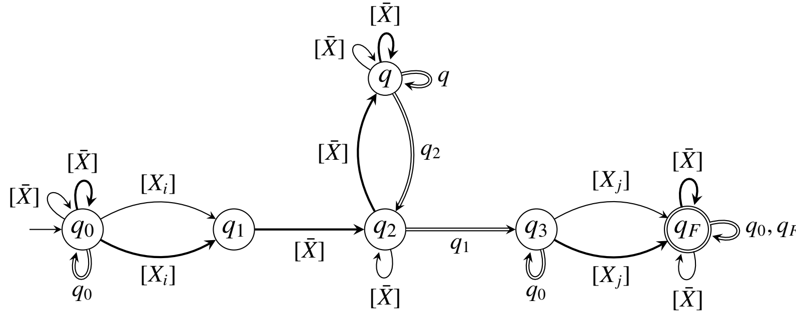


Fig. 3. Floyd automaton for atomic formula $\psi = X_i \curvearrowright X_j$

The automaton, after a generic sequence of moves corresponding to visiting an irrelevant portion of the syntax tree, when reading X_i performs either a mark or a push move, depending on whether X_i is a leftmost leaf of the tree or not; then it visits the subsequent subtree ending with a flush labeled q_1; at this point, if it reads X_j, it accepts anything else will follow the examined fragment.

Then, a natural inductive path leads to the construction of the automaton associated with a generic MSO formula: the disjunction of two subformulae can be obtained by building the union automaton of the two corresponding FA; similarly for negation. The existential quantification of X_i is obtained by projection erasing the i-th component. Notice that all matrices $M(\psi)$ are well defined for any ψ because the first n components of the alphabet are never erased by quantification. The alphabet of the automaton equivalent to φ' is $\Lambda(\varphi') = \{0, 1\}^n$, which is in bijection with Σ.

3.3 From Floyd automata to $\mathbf{MSO}_{\Sigma,M}$

Unlike the previous construction, this part of the theorem sharply departs from traditional techniques.

Let $\mathcal{A}$ be a deterministic FA over (Σ, M). We build an $\mathrm{MSO}_{\Sigma,M}$ sentence φ such that $L(\mathcal{A}) = L(\varphi)$. The main idea for encoding the behavior of the FA is based on assigning the states visited during its run to positions along the same lines stated by Büchi [10] and extended for VPL [4]. Unlike finite state automata and VPA, however, FA do not work on-line. Hence, it is not possible to assign a single state to every position. Let $Q = \{q_0, q_1, \ldots, q_N\}$ be the states of $\mathcal{A}$ with q_0 initial; as usual, we will use second order variables to encode them. We shall need three different sets of second order variables, namely $P_0, P_1, \ldots, P_N$, $M_0, M_1, \ldots, M_N$ and $F_0, F_1, \ldots, F_N$: set P_i contains those positions of s where state i may be assumed after a push transition. M_i and F_i represent the state reached after a flush: F_i contains the positions where the flush occurs, whereas M_i contains the positions preceding the corresponding mark. Notice that any position belongs to only one P_i, whereas it may belong to several F_i or M_i (see Figure 4).

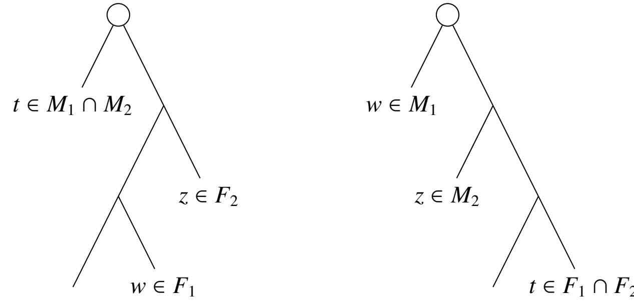


Fig. 4. Example trees with a position t belonging to more than one M_i (left) and F_i (right).

We show that $\mathcal{A}$ accepts a string s iff $\#s\# \models \varphi$, where

$$\begin{aligned}\varphi := \exists P_0, P_1, \ldots, P_N, M_0, M_1, \ldots, M_N, F_0, F_1, \ldots, F_N, e\, (0 \in P_0 \wedge \\ \wedge \bigvee_{i \in F} e \in F_i \wedge \neg\exists x(e+1 < x) \wedge \#(e+1) \wedge \varphi_\delta \wedge \varphi_{exist} \wedge \varphi_{unique}).\end{aligned} \tag{3}$$

The first clause encodes the initial state, whereas the second, third and fifth ones encode the final states. We use variable e to refer to the *end* of s, i.e., e equals the last position $|s|$. The other remaining clauses are defined in the following: the fourth one encodes the transition function; the last ones together encode the fact that there exists exactly one state that may be assumed by a push transition in any position, and exactly one state as a consequence of a flush transition.

For convenience we introduce the following notational conventions.

$$x \circ y := \bigvee_{M_{a,b} = \circ} a(x) \wedge b(y), \text{ for } \circ \in \{\lessdot, \doteq, \gtrdot\}$$

$$\mathrm{Tree}(x, z, w, y) := \begin{pmatrix} x \curvearrowright y \wedge \\ (x + 1 = z \ \vee \ x \curvearrowright z) \wedge \neg\exists t(x < t < z \wedge x \curvearrowright t) \wedge \\ (w + 1 = y \ \vee \ w \curvearrowright y) \wedge \neg\exists t(w < t < y \wedge t \curvearrowright y) \end{pmatrix}$$

$$\begin{aligned}
\mathrm{Succ}_k(x,y) &:= x+1 = y \wedge x \in P_k \\
\mathrm{Next}_k(x,y) &:= x \curvearrowright y \wedge x \in M_k \wedge y-1 \in F_k \\
\mathrm{Flush}_k(x,y) &:= x \curvearrowright y \ \wedge \ x \in M_k \ \wedge y-1 \in F_k \ \wedge \\
&\qquad \exists z, w \left(\mathrm{Tree}(x,z,w,y) \wedge \bigvee_{i=0}^{N} \bigvee_{j=0}^{N} \left(\begin{array}{c} \delta(q_i, q_j) = q_k \wedge \\ (\mathrm{Succ}_i(w,y) \vee \mathrm{Next}_i(w,y)) \wedge \\ (\mathrm{Succ}_j(x,z) \vee \mathrm{Next}_j(x,z)) \end{array} \right) \right) \\
\mathrm{Tree}_{i,j}(x,z,w,y) &:= \mathrm{Tree}(x,z,w,y) \wedge \left(\begin{array}{c} \mathrm{Succ}_i(w,y) \vee \mathrm{Flush}_i(w,y)) \wedge \\ (\mathrm{Succ}_j(x,z) \vee \mathrm{Flush}_j(x,z)) \end{array} \right)
\end{aligned}$$

Remark If ${}^{a}[c_1 c_2 \dots c_\ell]^{b}$ is a simple chain with support

$$q_i = q_{t_0} \xrightarrow{c_1} q_{t_1} \xrightarrow{c_2} \dots \xrightarrow{c_\ell} q_{t_\ell} \overset{q_{t_0}}{\Longrightarrow} q_k \tag{4}$$

then $\mathrm{Tree}_{t_0,t_\ell}(0,1,\ell,\ell+1)$ and $\mathrm{Flush}_k(0,\ell+1)$ hold; if ${}^{a}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{b}$ is a composed chain with support

$$q_i = q_{t_0} \overset{s_0}{\leadsto} q_{f_0} \xrightarrow{c_1} q_{t_1} \overset{s_1}{\leadsto} q_{f_1} \xrightarrow{c_2} \dots \xrightarrow{c_g} q_{t_g} \overset{s_g}{\leadsto} q_{f_g} \dots \xrightarrow{c_\ell} q_{t_\ell} \overset{s_\ell}{\leadsto} q_{f_\ell} \overset{q_{f_0}}{\Longrightarrow} q_k \tag{5}$$

then by induction we can see that $\mathrm{Tree}_{f_\ell,f_0}(0, x_1, x_\ell, |s|+1)$ and $\mathrm{Flush}_k(0, |s|+1)$ hold, where x_g is the position of c_g for $g = 1, 2, \dots, \ell$.

On the basis of the above definitions and properties, it is possible to bind $\mathcal{A}$ to suitable axioms φ_δ, φ_{exist}, and φ_{unique} that are satisfied by any chain with a support in $\mathcal{A}$, and in turn guarantee the existence of an $\mathcal{A}$'s support for every chain.

For the sake of brevity we report here only a (small) part of φ_δ which should provide enough evidence of the construction and of its correctness. The complete axiomatization and equivalence proof (based on a natural induction) are given in the Appendix [12].

The following *Forward formulae* formalize how $\mathcal{A}$ enters new states through push and flush transitions.

$$\varphi_{push_fw} := \forall x, y \bigwedge_{i=0}^{N} \left(\begin{array}{c} (x \lessdot y \vee x \doteq y) \wedge a(y) \\ \wedge \\ \mathrm{Succ}_i(x,y) \vee \mathrm{Flush}_i(x,y) \end{array} \Rightarrow y \in P_{\delta(q_i,a)} \right)$$

$$\varphi_{flush_fw} := \forall x, z, w, y \bigwedge_{i=0}^{N} \bigwedge_{j=0}^{N} \left(\mathrm{Tree}_{i,j}(x,z,w,y) \Rightarrow \begin{array}{c} x \in M_{\delta(q_i,q_j)} \\ \wedge \\ y-1 \in F_{\delta(q_i,q_j)} \end{array} \right)$$

Somewhat symmetric *Backward formulae* allow to reconstruct (in a unique way) $\mathcal{A}$'s states that lead to a given state.

Finally, for any chain $\#s\#$, by the complete axiom φ defined in (3) we obtain the following proposition which, together with Proposition 1, completes Theorem 1.

Proposition 2. *For any Floyd automaton $\mathcal{A}$ there exists an $MSO_{\Sigma,M}$ sentence φ such that $L(\mathcal{A}) = L(\varphi)$.*

4 Conclusions and future work

This paper at last completes a research path that began more than four decades ago and was resumed only recently with new -and old- goals. FLs enjoy most of the nice properties that made regular languages highly appreciated and applied to achieve decidability and, therefore, automatic analysis techniques. In this paper we added to the above collection the ability to formalize and analyze FL by means of suitable MSO logic formulae. New research topics, however, stimulate further investigation. Here we briefly mention only two mutually related ones. On the one hand, FA devoted to analyze strings should be extended in the usual way into suitable transducers. They could be applied, e.g. to translate typical mark-up languages such as XML, HTML, LaTeX, ... into their end-user view. Such languages, which motivated also the definition of VPL, could be classified as "explicit parenthesis languages" (EPL), i.e. languages whose syntactic structure is explicitly apparent in the input string. On the other hand, we plan to start from the remark that VPL are characterized by a well precise shape of the OPM [5] to characterize more general classes of such EPL: for instance the language of Example 1 is such a language that is not a VPL. Another notable feature of FL, in fact, is that they are suitable as well to parse languages with implicit syntax structure such as most programming languages, as well as to analyze and translate EPL.

References

1. Floyd, R.W.: Syntactic analysis and operator precedence. Journ. ACM **10** (1963) 316–333
2. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008)
3. Berstel, J., Boasson, L.: Balanced grammars and their languages. In et al., W.B., ed.: Formal and Natural Computing. Volume 2300 of LNCS., Springer (2002) 3–25
4. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journ. ACM **56** (2009)
5. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly pushdown property. Journal of Computer and System Science **78** (2012) 1837–1867
6. Barenghi, A., Viviani, E., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: PAPAGENO: a parallel parser generator for operator precedence grammars. In: SLE2012 - 5th International Conference on Software Language Engineering. (2012)
7. Lonati, V., Mandrioli, D., Pradella, M.: Precedence automata and languages. In Kulikov, A.S., Vereshchagin, N.K., eds.: CSR. Volume 6651 of Lecture Notes in Computer Science., Springer (2011) 291–304
8. Lonati, V., Mandrioli, D., Pradella, M.: Precedence automata and languages. CoRR-arXiv **1012.2321** (2010) http://arxiv.org/abs/1012.2321.
9. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Cambridge University Press (2012)
10. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics. (1990) 133–192
11. Crespi Reghizzi, S., Mandrioli, D., Martin, D.F.: Algebraic properties of operator precedence languages. Information and Control **37** (1978) 115–133
12. Lonati, V., Mandrioli, D., Pradella, M.: Logic characterization of Floyd languages. CoRR-arXiv **1204.4639** (2012) http://arxiv.org/abs/1204.4639.