Monadic logic programs and functional complexity

Armando B. Matos*
LIACC, Universidade do Porto, Rua do Campo Alegre 823, 4100 Porto, Portugal

Received October 1994; revised March 1996
Communicated by M. Nivat

Abstract

Problems related to the complexity and to the decidability of several languages weaker than Prolog are studied in this paper. In particular, monadic logic programs, that is, programs containing only monadic functions and monadic predicates, are considered in detail.

The functional complexity of a monadic logic program is the language of all words $f_1 \cdots f_k$ such that the literal $p(f_1(\cdots (f_k(a))\cdots))$ is a logical consequence of the program. The relationship between several subclasses of monadic programs, their functional complexities and the corresponding automata is studied.

It is proved that the class of monadic programs corresponds exactly to the class of regular languages. As a consequence, the "SUCCESS" problem is decidable for that class. It is also proved that the success set of a specific subclass of monadic programs ("simple" programs) corresponds exactly to regular languages with star-height not exceeding 1.

1. Introduction

The properties of several restricted forms of "classical" programming languages were studied in the 1960s and 1970s. In particular, the so-called "loop programs" have been considered in detail. Meyer and Ritchie [11] defined a hierarchy of functions $L_0, L_1, \ldots$ where $L_i$ is the class of functions computable by loop programs with a nesting of at most $i$ loops; this class of programs is called $L_i$. It has been proved that for all integers $n \geq 0$ we have $L_n \subseteq L_{n+1}$ and $L_n \neq L_{n+1}$; moreover, $\bigcup_{i \geq 0} L_i$ is the class of primitive recursive functions [12]. Cherniavsky [14] characterized another sub-recursive class $L_+$ of languages that essentially lies between $L_1$ and $L_2$ and that corresponds exactly to Presburger formulas [10]. In [1, 8] the time complexity of computations in sub-recursive languages is studied.

Several languages weaker than (i.e. properly included in) Prolog are used in practice, namely in the areas of deductive databases and type theory, so that it is important to...
characterize their expressiveness. In general, a more restricted language has simpler properties and it is theoretically interesting to explore the relationship between the expressive power of a language and its complexity or decidability properties.

This paper deals mainly with logic programs having only monadic function symbols and monadic predicate symbols. Such programs are called monadic. To each monadic program and goal we associate a language consisting of sequences of (monadic) function symbols, as illustrated in the following example.

**Example 1.** Consider the following logic program

\[ p(g(a)). \]
\[ p(f(X)) : - p(X). \]
\[ p(h(X)) : - r(X). \]
\[ r(a). \]
\[ r(h(X)) : - r(X). \]

The goal \( - p(t) \) where \( t \) is a ground term can be refuted (succeeds) for every \( t \) in

\[ \{ p(f^n(g(a))) : n \geq 0 \} \cup \{ p(f^n(h^m(a))) : n \geq 0, m \geq 1 \} \]

This set corresponds to a language with alphabet \( \Sigma = \{ f, g, h \} \) consisting of the **function symbols** present in the program; this particular language is regular and can be characterized by the following regular expression

\[ f^*g + f^*hh^* \]

A result proved in this paper (see Section 5.5) is that the language corresponding to a monadic program is always regular. It is interesting to look at some practical consequences of this fact.

**Example 2.** Consider the representation of the integer \( n \) by the term \( s^n(0) \) where \( s \) is the successor functor. It is possible to write a monadic program that recognizes, for instance, the multiples of any fixed integer (we leave this as a simple exercise). But there are other sets of integers, such as the set of prime numbers, which are not recognized by any monadic program.

**Example 3.** This example is taken from compiler theory. Consider the representation of expressions by suitable sequences of functors. For instance, the expression \("(12.2 + 1) \times 7"\) might be represented by the following term, where \("a"\) is some fixed, arbitrary, atomic constant.

\[ open(one(two(dot(two(plus(one(close(times(seven(a)))))))))))) \]

There are monadic programs (using a representation like this) that recognize, for instance, the set of floating point constants (like \("-0.7"\) or \("12.22E-5"\)) — we again leave
this construction as an exercise – but there is no monadic program that recognizes the set of well-formed arithmetic expressions (because the corresponding language is not regular).

This paper is organized as follows. After presenting several definitions in Section 2, we consider in Section 3 some restrictions on the form of monadic programs, that do not reduce their expressiveness. The subclasses of monadic programs that are studied in this paper are characterized in Section 4. Section 5 contains the main results of the paper: the functional complexities of "simple", "linear", "binary" and general monadic programs are studied, respectively, in Sections 5.2, 5.3, 5.4 and 5.5. In Section 6, the decidability for classes of programs slightly more general than monadic programs is studied. Some of the results of this paper are related to the "regular canonical systems" of Büchi (the equivalence between general monadic programs and regular languages can be proved with the help of Büchi results). This relationship is explained in Section 7. Conclusions and directions for future work are presented in Section 8.

2. Definitions

Monadic programs are logic programs containing only monadic predicate symbols and monadic function symbols. We are interested in the functional complexity problem, that is, in studying the sequences of function symbols $f_1 \cdots f_k$ (where $k \geq 0$) such that, for a given predicate $p$ called the starting predicate, the goal $p(f_1(\cdots f_k(a)\cdots))$ succeeds for some $a$. We associate to each monadic program and to each predicate symbol, the language of all those sequences.

**Definition 1.** Let $\mathcal{F}$ be the alphabet of functional (monadic) symbols of the monadic program $P$ which is assumed to be nonempty. The functional language associated with $P$ is the set $L_P \subseteq \mathcal{F}^*$ consisting of all words $f_1 f_2 \cdots f_k$ ($k \geq 0$) such that, for some atom "$a" and functional symbols $f_1 \in \mathcal{F}, \ldots, f_k \in \mathcal{F}$ ($k \geq 0$), the literal

$$p(f_1(f_2(\cdots f_k(a)\cdots)))$$

is a logical consequence of $P$. We say that program $P$ realizes the language $L_P$.

Note that the functional language of a goal–program pair is relative to the fixed atomic constant "$a" that occurs in every goal.

The reader may find it helpful to look again at Example 1 given in the introduction, and apply Definition 1.

Whenever there is no possible confusion, this mapping (between a goal–program pair and a language) will not be made explicit. A sequence of function symbols may either denote a word of a language or a function obtained by composition. If, for instance, we have the word of $\mathcal{F}^*$

$$F = f_1 f_2 \cdots f_k$$

($k \geq 0$)
we will also talk about the literal \( p(Fa) \) which is, of course
\[
p(f_1(f_2(\cdots f_k(a)\cdots)))
\]
Functional parenthesis will be omitted; the same literal can also be denoted by
\[
p(f_1f_2\cdots f_k a)
\]
The characterization of the \textit{linear} subclass of monadic programs and of the corresponding functional complexity, uses two concepts that we now define: \textit{predicate dependence graph} of a logic program and \textit{star-height} of a regular language.

\textbf{Definition 2.} Let \( P \) be a Prolog program, let \( V \) be the set of predicates defined in \( P \) and let \( E \) be the set of all pairs \((p, q)\) such that \( p \in V, q \in V \) and at least one clause defining \( p \) has a body containing a literal with predicate symbol \( q \) (we say that the definition of \( p \) uses \( q \) or, in short, that \( p \) "calls" \( q \)). The \textit{predicate dependence graph} of \( P \) is the directed graph \((V, E)\).

\textbf{Definition 3.} A Prolog program is called \textit{almost acyclic} if its predicate dependence graph contains no cycle of length greater than 1 (in other words, if all arcs with the form \((p, p)\) are removed from the predicate dependence graph, we get an acyclic graph).

Let us now define the star-height of a regular expression and of a regular language.

\textbf{Definition 4.} The star-height \( h(E) \) of a regular expression \( E \) is defined by the rules
\[
\begin{align*}
h(\varepsilon) &= 0 \\
h(\emptyset) &= 0 \\
h(a) &= 0 \quad \text{for every } a \in \Sigma \\
h(E + F) &= \max(h(E), h(F)) \\
h(EF) &= \max(h(E), h(F)) \\
h(E^*) &= h(E) + 1
\end{align*}
\]
The star-height of a regular language is the \textit{minimum} star-height of a regular expression denoting that language.

For example the expression \((fg^*)^*\) has a star-height 2 but denotes a language which has a star-height 1 because it is equivalent to the expression \(f(f + g)^* + \varepsilon\) and it is not equivalent to any regular expression not using "*".

Problems related to the star-height of a language are often more difficult than they seem at first glance; we invite the reader to solve the following question: find a regular language with a star height of 2 (and prove it!). The study of the star-height of regular
languages is an active area of research; for instance, before 1987 no algorithm to compute the star-height of a given regular language [6] was known.

3. Restrictions on monadic programs

Given any monadic program $P$ it may be possible (and useful) to find an equivalent program $P'$ with a more restricted form. By “equivalent” we mean that $L_P$ can be easily obtained from $L_{P'}$; in many cases we have in fact that $L_P = L_{P'}$.

We now define several such restrictions that do not cause any change in the corresponding language classes. These restrictions have to do with the number of atoms in the program and with the form of the clauses.

3.1. Reduction to one atom

Let $A$ be the set of atoms of $P$ and let "a" be the atom of the goal (which may or may not belong to $A$). As we are only interested in studying $L_P$, the functional complexity of $P$, we would like to reduce the number of atomic constants involved, without modifying $L_P$. A first idea is to replace in $P$ every atom by $a$. But this modification may, of course, change also $L_P$ as the following example shows:

$$p(fX) : - q(X), r(X).$$

$$q(a_1).$$

$$r(a_2).$$

We have $L_P = \emptyset$. But, if atoms $a_1$ and $a_2$ are replaced by $a$, we get a program $P'$ such that $L_{P'} = \{f\}$.

We will use another program transformation where the transformed functional language is similar, but not equal, to the original language. The method used is not complex but we will explain it in some detail. First we need the following language theoretic definition.

**Definition 5.** The projection $L \downarrow \Sigma'$ of a language $L$ with alphabet $\Sigma$ over the alphabet $\Sigma' \subseteq \Sigma$ is the set obtained by deleting in the words of $L$ all the occurrences of the symbols of $\Sigma$ not in $\Sigma'$.

The regular, context free and star-height not exceeding $k$ (with $k$ fixed) classes of languages are closed under language projections. We state this fact in the following theorem.

**Theorem 1.** Let $L$ be a language with alphabet $\Sigma$ and let $\Sigma'$ be any subset of $\Sigma$. If $L$ is context free, $L \downarrow \Sigma'$ is context free. If $L$ is regular, $L \downarrow \Sigma'$ is regular. If $L$ is a regular language with a star-height not exceeding a fixed value $k$, $L \downarrow \Sigma'$ is also a regular language with a star-height not exceeding $k$. 
Proof. The proofs are simple. We consider only the case of regular languages. Let \( A \) be a finite automaton recognizing \( L \). Replace in \( A \) every label belonging to \( \Sigma \setminus \Sigma' \) by \( \epsilon \). We get an \( \epsilon \)-automaton recognizing \( L \setminus \Sigma' \). \( \square \)

Let us now describe the program transformation used. Essentially, we replace in \( P \) every atom \( a_i \) distinct from \( a \) by \( g_i(a) \) where \( g_i \) is a new functor symbol.

Suppose that \( a_1, a_2, \ldots, a_k \) are the atoms in \( P \) distinct from \( a \) (it may happen that all atoms of \( P \) are distinct from \( a \) - this does not imply that \( L_P \) is empty). The transformed program \( P' \) is obtained as follows. For \( i = 1, \ldots, k \) replace every occurrence of \( a_i \) by \( g_i(a) \), where the \( g_i \)'s are new functor symbols (that is, they do not occur in \( P \)). It is not difficult to see that

\[
L_P = L_{P'} \downarrow \mathcal{F}
\]

where \( \mathcal{F} \) is the set of functor symbols of \( P \). So we see that the functional complexity of \( P \) can be studied by considering the transformed program \( P' \) (containing only one atom) and projecting the corresponding language over \( \mathcal{F} \), the set functional symbols of \( P \).

As an example of the one atom reduction consider the following program:

\[
p(fX) : - q(Y), r(Y).
q(a_1).
\]
\[
r(a).
\]
\[
r(a_1).
\]
\[
r(a_2).
\]

The functional complexity is \( L_P = f f^* \) (identifying a regular expression with the language denoted by it). The transformed program \( P' \) is

\[
p(fX) : - q(Y), r(Y).
q(g_1(a)).
\]
\[
r(a).
\]
\[
r(g_1(a)).
\]
\[
r(g_2(a)).
\]

and, noting that the Herbrand universe now includes terms with functors \( g_1 \) and \( g_2 \), we have \( L_{P'} = f(f + g_1 + g_2)^* \). We may check the relationship between the two functional languages.

\[
L_P = f f^* = (f(f + g_1 + g_2)^*) \downarrow \{f\} = L_{P'} \downarrow \{f\}
\]
3.2. Restriction on the form of the clauses

We can restrict the form of monadic programs clauses without changing the corresponding language. In the following, we use the results of Section 3.1 and assume that monadic programs contain only one atom called \( a \).

**Theorem 2.** For every monadic program \( P \) there is an equivalent monadic program \( P' \) satisfying the following conditions.

1. The argument of every unit clause is \( a \) (neither variables nor functional symbols can occur).
2. Every nonunit clause has one of the following forms:
   
   (a) \( p(fX) : - q(X) \)
   
   (b) \( p(X) : - q(fX) \)
   
   (c) \( p(X) : - q(X) \) where \( p \) and \( q \) are distinct predicate symbols
   
   (d) \( p(X) : - q(X), s(X) \) where \( p, q \) and \( s \) are distinct predicate symbols

Programs \( P \) and \( P' \) are equivalent in the sense that \( L_P = L_{P'} \).

**Proof.** Use iteratively the following transformations until none can be applied.

1. Replace each unit clause \( p(Fa) \) with \( F = f_1 \cdots f_k \neq e \) by the clauses

   \[ p(f_1X) : - q_1(X) \]
   
   \[ q_1(f_2X) : - q_2(X) \]
   
   \[ \vdots \]
   
   \[ q_{k-1}(f_kX) : - q_k(X) \]
   
   \[ q_k(a) \]

   where \( q_1, q_2, \ldots, q_k \) are new predicate symbols.

2. Make similar substitutions for each unit clause \( p(FX) \) with \( F \neq e \).

3. Replace each unit clause \( p(X) \) by the following clauses where \( f_1, f_2, \ldots, f_n \) are the functional symbols of \( P \).

   \[ p(f_1X) : - p(X) \]
   
   \[ p(f_2X) : - p(X) \]
   
   \[ \vdots \]
   
   \[ p(f_nX) : - p(X) \]
   
   \[ p(a) \]

   It is easy that, for every term \( t \) of the Herbrand universe, the goal \( p(t) \) will succeed.

4. Consider each ground literal \( p(a) \) occurring in the body of a clause. If the goal \( " : - p(a)" \) succeeds, remove the literal, otherwise remove the clause.
(5) For each clause body containing a literal \( p(X) \) such that \( X \) does not occur anywhere else in the clause do the following: if \( p(X) \) succeeds remove the literal; otherwise remove the clause.

(6) Use iteratively the following folding transformation in the bodies of clauses containing 3 or more literals: replace

\[ \ldots, p(F_1X), q(F_2X), \ldots \]

by \( \ldots, s(X), \ldots \) where \( s \) is a new predicate defined by the clause

\[ s(X) :- p(F_1X), q(F_2X) \]

(7) For each clause with the form \( p(GX) :- q(FX) \) with \( F = f_1 f_2 \cdots f_k (k \geq 1) \) introduce \( k \) new predicate symbols \( q_1, \ldots, q_k \) transforming it into

\[ p(GX) :- q_1(X) \]
\[ q_1(X) :- q_2(f_1X) \]
\[ q_2(X) :- q_3(f_2X) \]
\[ \vdots \]
\[ q_k(X) :- q(f_kX) \]

(8) Clauses \( p(F_1X) :- q(F_2X), r(F_3X) \) with \( |F_1| \geq 2 \) or \( |F_3| \geq 1 \) are similarly transformed.

(9) Transform each clause with the form \( p(FX) :- q(GX) \) with \( F = f_1 \cdots f_k, k \geq 2 \) into

\[ p(f_1X) :- p_1(X) \]
\[ p_1(f_2X) :- p_2(f_2X) \]
\[ \vdots \]
\[ p_{k-1}(f_kX) :- q(GX) \]

(10) Transform similarly the clauses with the form \( p(FX) :- q(X), r(X) \) with \( |F| \geq 1 \).

(11) Delete every clause with the form \( p(X) :- p(X) \).

(12) Delete every clause with the form \( p(X) :- p(X), p(X) \).

(13) Replace each clause with one of the forms (where \( p \neq q \))

\[ p(X) :- p(X), q(X) \]
\[ p(X) :- q(X), p(X) \]
\[ p(X) :- q(X), q(X) \]

by the clause \( p(X) :- q(X) \).
As a simple example, consider the clause
\[ p(fX) : - q(gX), r(hhX) \]
Applying rules 9 and 7 we get the equivalent set of clauses
\[ p(fX) : - p_1(X) \]
\[ p_1(X) : - q_1(X), r_1(X) \]
\[ q_1(X) : - q(aX) \]
\[ r_1(X) : - r_2(hX) \]
\[ r_2(X) : - r(hX) \]
Henceforth we may assume that monadic programs are in this form.

Note. In many situations, clauses of the form \( p(X) : - q(X) \) can also be eliminated. See the method of \( \varepsilon \)-transition elimination that is described before Theorem 8.

3.3. The proof of Theorem 2 is not constructive

There is a problem with the proof of Theorem 2: it does not give us a simplification algorithm, it only proves that it exists; in other words, it is not constructive.

The problem has to do with steps 4 and 5; for instance, in step 5 when we say "if \( p(X) \) succeeds..." (that is, "if, for some \( t \), there is a proof of \( p(t) \)"") we do not give a method to decide if it does. We only proved that a simplified (equivalent) program exists. We will now show that the goal is always decidable by giving a simplification algorithm.

Let us describe the constructive version of step 5 (step 4 is similar). Consider a clause having a literal \( p(X) \) in its body and suppose that the variable \( X \) does not occur anywhere else in the clause; we say that \( p(X) \) is an isolated literal (the proof of Theorem 2 shows how more complex cases can be reduced to this one). Let us call such a \( p(X) \) an isolated literal.

The method used for the elimination of an isolated literal (or the entire clause) is based on the following lemma (where by "proof of \( p(X) \)" we mean that, for some term \( t \), \( p(t) \) is a logical consequence of the program):

**Lemma 1.** If the body of a clause \( C \) contains the isolated literal \( p(X) \), then there is a proof of \( p(X) \) in the original program \( P \) iff there is a proof in program \( P' = P \setminus \{ C \} \).

In fact, it is easy to see that a shortest proof of \( p(X) \) does not involve the use of clause \( C \), because in order to "apply" the clause we must have already a proof of \( p(X) \). Note that \( P' \) is in general not equivalent to \( P \); it is only used to solve the goal \( p(X) \).
We have reduced the problem "solve\((p(X), P)\)" to the simpler problem
\[
solve\((p(X), P')\)
\]
where \(P'\) results from \(P\) by the deletion of a clause having \(p(X)\) as an isolated literal. This method can be applied recursively until we get a program containing no isolated literals. For these programs we assume (and prove later) that the solution can be found; let us call "\(succeeds(Goal,Prog)\)" to the corresponding algorithm.

In Fig. 1 we can see the algorithms to simplify a program eliminating all isolated literals.

It may be interesting to illustrate the method with a simple example; let us consider the goal \(\text{simplify}(P1)\) where \(P1\) is the program
\[
\begin{align*}
p(f(X)) & : - p(X), q(Y). \quad (P1) \\
q(X) & : - p(Y). \\
q(a).
\end{align*}
\]

First we will solve \(q(Y)\) (the isolated literal in first clause) in the following program (which results from deletion of the first clause of the original program); the goal is \(\text{succeeds}(q(Y), \text{simplify}(P2))\):
\[
\begin{align*}
q(X) & : - p(Y). \quad (P2) \\
q(a).
\end{align*}
\]

There is still one isolated literal, namely \(p(Y)\). So, we must consider the goal
\(\text{succeeds}(p(Y), \text{simplify}(P3))\)
\[
q(a). \quad (P3)
\]

\begin{verbatim}
function simplify(P)
    If P does not contain isolated literals
        return(P)
    Select an isolated literal \(p(X)\) in a clause \(C\)
    If succeeds\((p(X), simplify(P \{C\}))\)
        let \(P1 = P\) with \(p(X)\) removed from \(C\)
        return simplify(P1)
    else
        return simplify(P \{C\})
\end{verbatim}

Fig. 1. Algorithm for the elimination of isolated literals
As there is no solution, the first clause of program $P_2$ is deleted; so we want to know if $q(Y)$ has solution in

$q(a)$.  ($P_2$ simplified)

There is obviously the solution $Y = a$ so that $q(Y)$ is deleted from the first clause of the original program; we get:

$p(f(X)) : - p(X)$.  ($P_1$, first simplification)

$q(X) : - p(Y)$.

$q(a)$.

We must now solve $p(Y)$ (the remaining isolated literal) in the reduced program

$p(f(X)) : - p(X)$.  ($P_4$)

$q(a)$.

As there is no solution, the second clause of $P_4$ is deleted and we finally obtain the equivalent program (obviously, the first clause can also be deleted):

$p(f(X)) : - p(X)$.

(q(a).

4. Subclasses of monadic programs: binary, linear and simple programs

In this paper we study the following subclasses of monadic programs.

**Definition 6.** A monadic program such that each clause has at most one literal in its body is called **binary**.

The name "binary" comes from the fact that nonunit clauses have two literals, one positive and one negative.

**Definition 7.** A monadic program such that each nonunit clause has the form ("linear" clause)

$p(f_1 f_2 \ldots f_k X) : q(X)$  ($k \geq 0$)

(where $q$ may be identical to $p$) is called **linear**.

**Definition 8.** A program is called **simple** if it is linear and almost acyclic.

Notice that every simple program is linear, every linear program is binary and every binary program is monadic. Table 1 summarizes these definitions.
### Table 1
Monadic programs and its subclasses: summary

<table>
<thead>
<tr>
<th>Property</th>
<th>Monadic</th>
<th>Binary</th>
<th>Linear</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monadic</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Body literals</td>
<td>Note 1</td>
<td>≤ 1</td>
<td>≤ 1</td>
<td>≤ 1</td>
</tr>
<tr>
<td>Nonunit clauses</td>
<td>Note 2</td>
<td>$p(FX) : - q(GX)$</td>
<td>$p(FX) : - q(X)$</td>
<td>$p(FX) : - q(X)$</td>
</tr>
<tr>
<td>Almost acyclic</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Note 1:** Unrestricted. Without loss of generality, may be bounded by 2.

**Note 2:** Unrestricted. But see Theorem 2.

At this point we would like to remark the following. Any logic program $P$ can be "interpreted" by another program $P_1$ which uses only one predicate symbol (not occurring in $P$), say $s$. See for instance the example in Section 7. The resulting program $P'$ defines only one predicate ($s$) so that it is almost acyclic. Let us denote by $L(P, p)$ the functional language associated with the predicate $p$ defined in the logic program $P$. We have

$$L(P', s) = \bigcup_{j=1}^{n} p_j L(P, p_j)$$

where $p_1, \ldots, p_n$ are the predicates defined in $P$. We conclude that, for any logic program $P$ the union on the right-hand side of the equation can always be realized by an almost acyclic program. Notice, however, that the "interpretation" usually destroys linearity. For instance, the clause

$$p(f(g(X))) : - q(X)$$

is transformed into the nonlinear clause

$$s(p(f(g(X)))) : - s(q(X))$$

### 5. Some results on functional complexity

We now prove a number of results about the functional languages that correspond to monadic programs and to some of its subclasses. The classes themselves satisfy

$$\mathcal{M} \supseteq \mathcal{B} \supseteq \mathcal{L}_i \supseteq \mathcal{S}_i$$

where monadic, binary, linear and simple programs are denoted, respectively, by $\mathcal{M}$, $\mathcal{B}$, $\mathcal{L}_i$ and $\mathcal{S}_i$.

The corresponding languages satisfy

$$\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{L}_i) = \text{Reg} \supseteq \mathcal{L}(\mathcal{S}_i) = \mathcal{S}_1,$$

(2)
where $\mathcal{R}eg$ and $\mathcal{F}1$ denote, respectively, the class of (all) regular languages and the class of regular languages with a star-height 0 or 1. This relationship summarizes some results of this paper and will be proved in the rest of this section. To the classes $\mathcal{L}i$, $\mathcal{B}$ and $\mathcal{M}$ correspond, respectively (in a sense described below), finite automata, “inverse” automata and “intersection” finite automata.

After presenting in Section 5.1 a result about general monadic programs, we consider the successively wider classes of programs: simple (Section 5.2), linear (Section 5.3), binary (Section 5.4), and monadic (Section 5.5).

5.1. Functional complexity of monadic programs

Let us first present a result about some sets realizable by monadic programs, that will be used in Section 5.2 where simple programs are considered.

We use the standard notation for regular expressions [7] augmented with the operators $\cap$ (set intersection) and $\oplus$; the latter denotes (note that $L$ is a set and $w$ is a word)

$$L \oplus w = \{x | wx \in L\}.$$  

**Theorem 3.** Suppose that $L$, $L_1$ and $L_2$ are sets realizable by monadic programs and that $w, w_1, w_2, \ldots$ are words realizable by monadic programs; then the following are also realizable by monadic programs.

1. The word $\varepsilon$.
2. All words $f \in \Sigma$.
3. The set $\emptyset$.
4. The set $L_1 + L_2$.
5. The set $wL$.
6. The set $L \oplus w$.
7. The set $(w_1 + \cdots + w_k)^*L$.
8. The set $L_1 \cap L_2$.

**Proof.** A brief justification for each case follows.

1. The word $\varepsilon$ is realized by the one clause program $p(a)$.
2. The word $f \in \Sigma$ is realized by the one clause program $p(fa)$.
3. The set $\emptyset$ is realized by the empty program.
4. If $L_1$ and $L_2$ are realized by two disjoint programs whose starting predicates are, respectively, $q$ and $r$, then $L_1 + L_2$ is realized by predicate $p$ in a program consisting of the clauses of the two programs and

$$p(X) :- q(X)$$

$$p(X) :- r(X)$$

defining the new starting predicate $p$.
5. Suppose that $L$ is realized by the predicate $q$. The set $wL$ is realized by the new predicate $p$ defined by the clause $p(wX) :- q(X)$. 


(6) Use induction on $|w|$.

(7) The set $(w_1 \ldots w_k)^*L$ is realized by the following program where $q$ realizes $L$:

\[
\begin{align*}
p(X) & : - q(X) \\
p(w_1 X) & : - p(X) \\
& \vdots \\
p(w_k X) & : - p(X)
\end{align*}
\]

(8) If $L_1$ and $L_2$ are realized by two disjoint programs whose starting predicates are, respectively, $q$ and $r$, then $L_1 \cap L_2$ is realized by a program consisting of the clauses of the two programs and the new clause $p(X) : - q(X), r(X)$. □

5.2. Simple programs

We now study the class of regular languages that are realizable by monadic almost acyclic programs.

Lemma 2. A regular language with a star-height 0 or 1 is realizable by a simple program.

Proof. Obvious for star-height 0 (the language is finite). By definition, given a regular language with a star-height 1, there is some regular expression $E$ with star-height 1 that represents it. Let $F^*$ be a starred subexpression of $E$; as $F$ is finite we can rewrite it as

\[F^* = (w_1 + w_2 + \ldots + w_k)^*,\]

where all the $w_i$ are words. Using the properties of regular expressions, we can reduce the complete expression $E$ to

\[E = x_1 + x_2 + \ldots + x_n,\]

where each $x$ is a concatenation

\[x = y_1 y_2 \cdots y_m\]

Here $y_i$ is either a word or a reduced expression like $F^*$. The result follows if we use induction on $m$ and numbers 4, 5 and 7 of Theorem 3.

Note that cycles of the predicate dependence graph are only introduced by the use of number 7 of Theorem 3 which shows that the graph is almost acyclic; moreover, all clause bodies have only one literal. □

The class of languages realizable by simple programs is exactly the subclass of the regular languages that have star-height 0 or 1. Denote by $\mathcal{S}n$ ($n \geq 0$) the class of the regular languages that have star-height not exceeding $n$. The class $\mathcal{S}1$ has
a number of interesting properties; for instance, the following closure properties hold.

Theorem 4. If \( L, L_1 \) and \( L_2 \) are in \( \mathcal{S}1 \) and \( w, w_1, \ldots, w_k \) are fixed words then the following languages are also in \( \mathcal{S}1 \):

1. \( L_1 + L_2 \),
2. \( wL \),
3. \((w_1 + \cdots + w_k)^*L\),
4. \( L \ominus w \) (may be proved by induction on \(|w|\)).

Proof. Parts 1, 2 and 3 are obvious from the definition of star-height 0 or 1 languages. Part 4 can be proved by induction on the length of \( w \). \( \square \)

We will now prove the equivalence between simple programs and the \( \mathcal{S}1 \) class.

Lemma 3. A language realized by a simple program belongs to \( \mathcal{S}1 \).

Proof. Let the clauses which define the starting predicate \( p \) be

\[
p(u_1a).
\]

\[
\vdots
\]

\[
p(u_na).
\]

\[
p(v_1X) :- q_1(X)
\]

\[
\vdots
\]

\[
p(v_mX) :- q_m(X)
\]

\[
p(w_1X) :- p(X)
\]

\[
\vdots
\]

\[
p(w_nX) :- p(X)
\]

where the \( u \)'s, the \( v \)'s and the \( w \)'s are fixed words and the \( q \)'s are predicates distinct from \( p \) not necessarily all distinct. By induction we may assume that for \( 1 \leq i \leq m \) the language \( L_i \) which corresponds to \( q_i \) is in \( \mathcal{S}1 \) (recall that the predicate dependence graph is almost acyclic). The first \( k + m \) clauses realize the language

\[
L_a = (u_1 + \cdots + u_k) + (v_1L_1 + \cdots + v_mL_m)
\]

which is in \( \mathcal{S}1 \). It is not difficult to see that the language \( L \) realized by \( p \) is

\[
L = (w_1 + \cdots + w_n)^*L_a
\]
This language is also in $\mathcal{S} 1$. This result includes several particular cases; for instance, if $n = 0$ we get

$$L = \emptyset^*L_a = \varepsilon L_a = L_a$$

and if $k = m = 0$

$$L = \emptyset^*(\emptyset + \emptyset) = \emptyset$$

Lemmas 2 and 3 show that the class of languages realizable by simple programs is exactly the class of regular languages having a star-height of 0 or 1.

**Theorem 5.** The class of languages realizable by simple programs coincides with $\mathcal{S} 1$.

### 5.3. Functional complexity of linear programs

The following Theorem shows that the rather restricted class of linear programs corresponds already to the class of regular languages.

**Theorem 6.** Linear programs realize exactly the class of regular languages.

**Proof.** Given a linear program $P$, consider an equivalent linear program $P'$ such that:

- All unit clauses have the form $p(a)$.
- All nonunit clauses have the form $p(fX) : - q(X)$.

The transformation can easily be accomplished using Theorem 2. Consider for instance the clause $p(fgfX) : - q(X)$ in $P$. The corresponding clauses in $P'$ are

$$p(fX) : - r_1(X)$$

$$r_1(gX) : - r_2(X)$$

$$r_2(fX) : - q(X)$$

where $r_1$ and $r_2$ are new predicate symbols. From the transformed program we build a nondeterministic finite automaton $A$ as follows.

- The states of $A$ correspond to the predicate symbols defined in the program.
- The initial state of $A$ corresponds to the starting predicate $p(X)$.
- Final states of $A$ correspond to predicates whose definition includes a unit clause (recall that all unit clauses have the form $p(a)$).
- There is a transition in the automaton from state $p$ to state $q$ with symbol $f$ iff the program contains a clause

$$p(fX) : - q(X)$$

It is easy to see that the goal $p(X)$ can succeed instantiated as $p(FX)$ (where $F$ is a sequence of functional symbols) iff there is a path in $A$, corresponding to the symbols of $F$, from $p$ to a final state.
Conversely, given a regular language $L$, it is easy to define a linear program that realizes exactly the words of $L$. □

We give now an example of the construction explained in the previous proof. Consider the linear program

\[
\begin{align*}
{p}(fX) & : - {q}(X). \\
{q}(a) & . \\
{q}(gX) & : - {p}(X). \\
{q}(hX) & : - {r}(X). \\
{r}(a) & . \\
{r}(fX) & : - {q}(X).
\end{align*}
\]

The corresponding automaton is

As an example, we can see that the fact that the goal $p(X)$ can succeed as $p(fhfh)$, corresponds to the recognition by the automaton of the word $fhfh$.

5.4. Functional complexity of binary programs

As in the case of linear programs we can define an automaton that represents a binary program. We first consider an equivalent binary program where each nonunit clause has either the form $p(fX) : - q(X)$ or the form $p(X) : - q(fX)$. The corresponding automaton edge (from state $p$ to state $q$) is labeled, respectively, with $f$ or $f^{-1}$.

We give an example of this construction. Consider the linear program

\[
\begin{align*}
{p}(fX) & : - {q}(X). \\
{q}(a) & . \\
{q}(X) & : - {p}(fX). \\
{q}(hX) & : - {r}(X). \\
{r}(a) & . \\
{r}(fX) & : - {q}(X).
\end{align*}
\]

The corresponding automaton is
We call this kind of automaton an inverse automaton. Its semantics can be obtained from the corresponding logic program: if the automaton is in state $s$, there is a transition $s \xrightarrow{a^{-1}} s'$ and the word seen so far has the form $xa$, the automaton can go to state $s'$ changing the "word seen so far" to $x$.

We now define more rigorously the language accepted by an inverse automaton $A$.

**Definition 9.** Consider the following (non deterministic) $\varepsilon$-finite automaton $A'$ whose alphabet $\Sigma \cup \Sigma^{-1}$ includes inverse symbols (where $\Sigma^{-1}$ is the set of symbols $\{a^{-1} : a \in \Sigma\}$).

$$A' = (\Sigma \cup \Sigma^{-1}, S, s_0, F, \delta)$$

where $S$ is the set of states, $s_0 \in S$ the initial state, $F \subseteq S$ the set of final states and

$$\delta \subseteq S \times (\Sigma \cup \Sigma^{-1} \cup \{\varepsilon\}) \times S$$

We say that $u$ is recognized by $A$ or, equivalently that $u \in \mathcal{L}(A)$, if there is a word $v \in (\Sigma \cup \Sigma^{-1})^*$ recognized by $A'$ that can be transformed in $u$ by the application of the rule

$$aa^{-1} = \varepsilon \quad (a \in \Sigma)$$

We will now see that, somewhat surprisingly, inverse automata recognize only regular languages.

The algorithm in Fig. 2 eliminates from an inverse automaton (which can contain $\varepsilon$-transitions) all inverse transitions. The relationship between its variables can be summarized as follows where $\xrightarrow{\varepsilon,*}$ represents any sequence of $\varepsilon$ transitions.

$$q \xrightarrow{\varepsilon,*} s_1 \xrightarrow{a} s_2 \xrightarrow{\varepsilon,*} s \xrightarrow{a^{-1}} s'$$

The edges introduced are $q \xrightarrow{a} s'$. The correctness of the algorithm can easily be proved by noticing that inverse transitions are useless, except if there is at least one $s \xrightarrow{a^{-1}} s'$ such that state $s$ can be reached using a transition with label $a$ eventually followed or preceded by $\varepsilon$-transitions. The inverse transition can then be removed adding appropriate $\varepsilon$-transitions.

When there are no more "useful" inverse transitions, the remaining ones can be removed.

This transformation applied to the automaton considered above gives
function del_neg(graph (V, E))

while possible {

    Select an inverse edge \( s \xrightarrow{a^{-1}} s' \) such that

    there is (at least) an \( a \)-transition to a state in \( C^{-1}_e(s) \);

    for all edges \( s_1 \xrightarrow{a} s_2 \) such that \( s_2 \in C^{-1}_e(s) \) {

        for all states \( q \in C^{-1}_e(Q) \) {

            \( E = E \cup \{q \xrightarrow{c} s'\} \)

        }

    }

    \( E = E \setminus \{s \xrightarrow{a^{-1}} s'\} \)

}

Remove from \( E \) all remaining inverse edges

return \((V, E)\)

Fig. 2. Algorithm for the elimination of inverse edges in a \( e \)-automaton. Note: \( C^{-1}_e(S) \) represents the set of states from which \( s \) can be reached using only \( e \)-transitions.

The \( e \)-transition on the initial state is useless; we obtain the equivalent logic program

\[
\begin{align*}
p(fX) & : - q(X). \\
q(a). \\
q(hX) & : - h(X). \\
r(a). \\
r(fX) & : - q(X). \\
r(X) & : - p(X)
\end{align*}
\]

Noting that the class of binary programs includes the class of linear programs we have (using Theorem 6) the following result.

**Theorem 7.** Binary programs realize exactly the class of regular languages.

5.5. General monadic programs

We prove that the class of languages that corresponds to monadic programs is the class of regular languages. Before explaining the ideas behind the proof let us consider two simple examples.
5.5.1. Introductory examples

Consider the following program.

\begin{align*}
p(X) & : - q(X), r(X). \\
q(a). \\
q(fX) & : - p(X). \\
r(a). \\
r(fX) & : - p(X).
\end{align*}

Note that all clauses except the first, have at most one literal in their bodies. We will define an equivalent binary logic program (that is, a program where clause bodies have at most one literal). In general, is does not seem easy to achieve this transformation using only fold/unfold operations. On the other hand, the fact that the functional languages of every monadic programs is regular does not follow directly from the regularity of languages associated with binary program and from the closure of regular languages for intersection. We use a method based on a system of equations on language variables.

Let \( P, Q \) and \( R \) be the functional languages that correspond, respectively, to the predicates \( p, q \) and \( r \). These languages are the least fixed point of the following system of equations.

\begin{align*}
P & = Q \cap R \\
Q & = \varepsilon \cup fP \\
R & = \varepsilon \cup fP
\end{align*}

The first equation is deleted. To obtain an equivalent system of equations we must express the fact that \( P \) is the intersection of \( Q \) and \( R \). Using the last two equations we get

\[ P = Q \cap R = (\varepsilon \cup fP) \cap (\varepsilon \cup fP) = \varepsilon \cup fP \]

The following facts have been used.

- The sets represented by \( \varepsilon \) and by \( fP \) are disjoint.
- The intersection of \( fP \) with \( fP \) is of course \( fP \).

We get

\begin{align*}
P & = \varepsilon \cup fP \\
Q & = \varepsilon \cup fP \\
R & = \varepsilon \cup fP
\end{align*}

This system can now be solved and we get \( P = Q = R = f^* \). The equivalent program is

\begin{align*}
q(a). \\
q(fX) & : - p(X).
\end{align*}
As a more complex example consider the slightly different program

\[
p(X) : - q(X), r(X).
\]

\[
q(a).
\]

\[
q(fX) : - p(X).
\]

\[
r(a).
\]

\[
r(fX) : - r(X).
\]

The corresponding system of equations is

\[
P = Q \cap R
\]

\[
Q = \varepsilon \cup fP
\]

\[
R = \varepsilon \cup fR
\]

The computation of \( Q \cap R \) introduces a new intersection that will be denoted by \( S \).

\[
P = Q \cap R = (\varepsilon \cup fP) \cap (\varepsilon \cup fR) = \varepsilon \cup f\left(\overline{P \cap R}\right)
\]

The new equation is

\[
P = \varepsilon \cup fS
\]

We must now express the fact that \( S \) is the intersection of \( P \) with \( R \). The equations used are the previous one and the system equation for \( R \).

\[
S = P \cap R = (\varepsilon \cup fS) \cap (\varepsilon \cup fR) = \varepsilon \cup f\left(\overline{S \cap R}\right)
\]

Let us write the “current” system of equations (the equation for \( T \) will be defined later).

\[
Q = \varepsilon \cup fP
\]

\[
R = \varepsilon \cup fR
\]

\[
P = \varepsilon \cup fS
\]

\[
S = \varepsilon \cup fT
\]

We must now define \( T \)

\[
T = S \cap R = P \cap R \cap R = P \cap R = S
\]
No new equation is introduced; replace $T$ by $S$ in the system above and get

$$Q = \varepsilon \cup fP$$
$$R = \varepsilon \cup fR$$
$$P = \varepsilon \cup fS$$
$$S = \varepsilon \cup fS$$

The solution is (solve the last equation and replace $S$ by the solution) $P = Q = R = S = f^*.$

5.5.2. Elimination of nonbinary clauses

We now generalize the examples in Section 5.5.1 showing that clauses containing two literals in their body can be eliminated (possibly introducing new clauses).

First we define the normal form of language equations.

**Definition 10.** An equation is said to be in normal form if it is written as

$$L = [\varepsilon] \cup k_1L_1 \cup \cdots \cup k_nL_n \quad (n \geq 1)$$

where
- $L, L_1, \ldots, L_n$ are (not necessarily distinct) language variables.
- $[\varepsilon]$ denotes that the term $\varepsilon$ is optional.
- $k_1, \ldots, k_n$ are (not necessarily distinct) alphabet symbols.

As an example of a normal equation we have

$$L = \varepsilon \cup fL \cup fL_1 \cup gL_1 \cup gL_2$$

The following procedure describes a method for the transformation of a linear program plus a clause $p(X) : - q(X), r(X)$ into an equivalent linear program. It can be successively applied to transform a monadic program into an equivalent linear program. Recall that in Section 5.4 we have described an algorithm for the transformation of binary programs into equivalent linear programs. This result is used without explicit reference.

1. Consider a linear program together with an additional clause $p(X) : - q(X), r(X)$; without loss of generality, we may assume that this is the only clause defining $p.$ There exists an $\varepsilon$-automaton $A$ that corresponds to the logic program without the additional clause.

2. Edges in $A$ labeled with $\varepsilon$ can be eliminated. The algorithm presented here is different from the one presented in [7]. We want, in particular, that the transformed automaton has exactly the same set of states (corresponding to the set of predicate symbols defined in the program).

We assume that there are no $\varepsilon$-edges starting in the initial state of $A.$ If there are such edges, the transformation may not be possible. In this case consider another
Automaton $A_t$ which is identical to $A$ except for the following:

- The initial state of $A_t$, say $s_I$, is new.
- $A_t$ contains an edge labeled with a new symbol $b$ connecting $s_I$ to the initial state of $A$.

Automata $A$ and $A_t$ are similar in the sense that

$$L_A = \{ x : bx \in L_A, \}.$$  

We now explain briefly the algorithm for the elimination of $\epsilon$-edges in $A$ (or in $A_t$); the method used is somewhat similar to the one used for the elimination of inverse edges (described in Section 5.4).

If the $\epsilon$-edge connects a state to itself it is simply deleted.

Otherwise suppose that $(s,s')$ be the $\epsilon$-edge to be eliminated and denote by $C^\epsilon_C(s)$ the set of states from which $s$ is accessible using only edges labeled with $\epsilon$.

Consider all edges from $s_I$ to $s_2$ labeled with $a \in E$ such that $s_2 \in C^\epsilon_C(s)$ and all states $q \in C^\epsilon_C(s_1)$ (in particular, we have, of course, $s \in C^\epsilon_C(s)$ and $s_1 \in C^\epsilon_C(s_1)$). The relationship between these states is summarized in the following diagram.

Using this method, all $\epsilon$-edges can be deleted. Let $A'$ be the corresponding automaton.

(3) To $A'$ corresponds a normal system of language equations (once $A'$ has no $\epsilon$-transitions). Introduce the following equation corresponding to the new clause (where language $P$ corresponds to predicate $p$, etc.)

$$P = Q \cap R$$

Let $S$ be the corresponding system of equations. The least fixed point of $S$ corresponds to the success set of the program.

(4) Eliminate the equation $P = Q \cap R$ (possibly introducing new equations) as follows. Consider the equations defining $Q$ and $R$

$$Q = Q_{right}$$

$$R = R_{right}$$

and add the new equation

$$P = Q_{right} \cap R_{right}$$

stating that $P$ is the intersection of the languages defined by the two equations.
The computation of $Q_{\text{right}} \cap R_{\text{right}}$ may result in new intersections in terms like $f(S \cap T)$. If this is the case, there are two possibilities:

- If the intersection $S \cap T$ has been considered before and corresponds to a variable $V$, replace it by $V$.
- If the intersection $S \cap T$ has not yet been considered, define a new variable $X = S \cap T$ and eliminate the intersection as before (introducing a new equation).

As the number of (original) language intersections is finite, this process must terminate. We get an equivalent (having the same least fixed point) normal system of equations containing, possibly among others, the original language variables.

The following Theorem is based on the iterative application of the previous construction so as to eliminate all language intersections.

**Theorem 8.** For every monadic logic program $P$ there is an equivalent linear program $P'$ that contains, possibly among others, the definitions of the predicates characterized by $P$.

Under the point of view of functional complexity we see that linear, binary and monadic classes of programs are equivalent. The following important result is easy to prove.

**Corollary 1.** The class of languages realizable by monadic programs is the class of regular languages.

The discussion above suggests the definition of a new kind of automaton where the intersection of the languages represented by two states can be "added" to another state.

**Definition 11.** Intersection finite automata are finite automata that can also contain edges of the following kind:

These edges are interpreted as follows: The word $x$ can make the automaton go from the initial state to state $P$ if it can make it go both to $Q$ and to $R$.

Notice that in general intersection automata are not deterministic.

**Corollary 2.** Intersection finite automata recognize the class of regular languages.
6. Beyond monadic programs

It is interesting to study classes of logic programs slightly more general than the class of monadic programs and try to answer questions like:
- When does undecidability begin?
- Are there classes corresponding to nonregular languages?

Decidability is about a specific problem. The problem we have in mind is the SUCCESS problem which defined as follows.

Definition 12. Given a class $C$ of Prolog programs all of them including the definition of a predicate $p$ — the initial or starting predicate — and an instance program $P \in C$ consider the problem: does the query " :- $p(\cdot \cdot \cdot)$" have an SLD-refutation? This is the SUCCESS problem for class $C$. The exact form of the query must of course be specified.

Let us now summarize the decidability of the SUCCESS problem; some of these results are presented in the rest of this section.
- **General logic programs**
  Undecidable. Well-known result.
- **Monadic predicates, functions with arbitrary arity**
  Undecidable. Well-known result; a logic program can be simulated using only monadic predicates (in fact with one monadic predicate).
- **Monadic functions, predicates with arbitrary arity**
  Undecidable. The Post correspondence problem can be "programmed" using only monadic functions.
- **Monadic predicates, monadic functions**
  Decidable. An easy consequence of the results presented in Section 5.5.

6.1. Monadic programs are decidable

A simple consequence of the results presented in Section 5.5 is that the SUCCESS problem is decidable for every monadic program.

**Theorem 9.** The SUCCESS problem is decidable for every monadic program.

**Proof.** Consider the equivalent monadic program $P$ containing only one atom $a$ and the goal " :- $p(t)$" where $t$ is a term of the Herbrand Universe. Let $L$ be the corresponding functional language.

If $t$ is ground, $t = f_1 \ldots f_k a$, the goal succeeds iff the word $f_1 \ldots f_k$ belongs to $L$. This is decidable because the regular language $L$ can be constructively obtained from $P$. If $t = f_1 \ldots f_k X$, the goal succeeds iff the intersection of the language denoted by $f_1 \ldots f_k$ with the language $L$ is not empty. Again, because both languages are regular, this is decidable. \(\square\)
6.2. Programs with monadic predicates are general and undecidable

The following result is well known.

**Theorem 10.** For each logical program there exists an equivalent one that contains only one predicate symbol. This predicate is unary.

**Proof.** Let \( f_i \) with \( i \geq 0 \) be a collection of function symbols, one for each predicate \( p_i \) defined in the program (with the same arity) and let \( r \) be a new monadic predicate symbol. Each positive or negative literal occurring in the original program

\[
p_i(t_1, \ldots, t_k)
\]

is replaced by

\[
r(f_i(t_1, \ldots, t_k))
\]

Goals are transformed by the same process. \( \Box \)

**Corollary 3.** The SUCCESS problem is undecidable for the class of logic programs that define only one predicate \( r \) of arity 1.

Moreover it is a consequence of the results in [5] that, even for programs containing only a unary predicate defined by two clauses of the form

\[
r(f(r_1, \ldots, r_n)) : - r(f(s_1, \ldots, s_n))
\]

\[
r(f(t_1, \ldots, t_n))
\]

(where the \( r_i, s_i \) and \( t_i \) are terms) the SUCCESS problem is undecidable.

6.3. Programs with monadic functions are undecidable

**Theorem 11.** There are sub-classes of \( F1 \) (classes of programs having only monadic functors) for which the SUCCESS problem is recursively unsolvable.

**Proof.** Every instance of PCP (Post Correspondence Problem) can be "programmed" using only monadic functors.

This class of programs used is best illustrated by the example in Fig. 3 which corresponds to the PCP instance.

\[
\{(a,agg), (gfagg, gf), (gf, f)\}
\]

Different members of the class differ only in the definition of \( r \); each clause for \( r \) corresponds to a pair of words of the PCP instance.

The SUCCESS Problem for the class exemplified in Fig. 3 is recursively unsolvable. In fact, the query \( p \) succeeds if and only if the corresponding PCP instance has a solution. \( \Box \)
7. Relationship with the work of Büchi

When this paper was first submitted we were unaware of the work of Büchi [3, Chapter 5] which is related to our work. It is interesting to see how similar results can arise from independent work in two different areas (Büchi's work on certain kinds of rewrite systems and our work on restricted forms of logic programs).

Broadly speaking, his "regular systems" can be viewed as limited forms of rewrite systems (acting only on the left part of the words) or equivalently as (limited forms) of formal systems. Each regular system corresponds in a natural way to a regular language (this correspondence is the basic result in Chapter 5 of the mentioned book).

We begin by defining the form of regular system that is most important for us, the "many premise pure regular system with terminal set \( \{ \varepsilon \} \)". We follow Büchi but use a different notation.

**Definition 13.**

- An \( n \)-premise regular production is an expression

\[
x_1 \xi, x_2 \xi, \ldots, x_k \xi \rightarrow y \xi,
\]

where \( x_1, \ldots, x_k \) and \( y \) are words (members of \( \Sigma^* \)) and \( \xi \) is a variable over \( \Sigma^* \). The word \( w \) is a direct consequence of the words \( z_1, \ldots, z_k \) by this rule if there is a word \( u \) such that we can write

\[
w = yu, \quad z_1 = x_1 u, \quad \ldots, \quad z_k = x_k u
\]

- A regular system is a finite set \( S \) of regular productions. Each production may have an arbitrary (finite) number of premises. \( S \) directly produces the word \( y \) from the set of words \( U \) if there are \( u_1, \ldots, u_k \in U \) such that some rule of \( S \) directly produces \( y \) from \( \langle u_1, \ldots, u_k \rangle \).
- Let \( A \subseteq \Sigma^* \) be a finite set of axioms. A sequence \( \langle u_1, \ldots, u_n \rangle \) is an \((A,S)\)-deduction if, for each \( 0 \leq i < n \) either \( u_{i+1} \in A \) or \( S \) directly produces \( u_{i+1} \) from \( \{u_1, \ldots, u_i\} \).
- The set \( Pr(A,S) \subseteq \Sigma^* \) consists of all words \( u \in \Sigma^* \) such that there is an \((A,S)\)-deduction that ends in \( u \).
In order to understand how regular systems are related to the functional complexity of logic languages, consider again the program

\[
\begin{align*}
p(g(a)). \\
p(f(X)) & : \neg p(X). \\
p(h(X)) & : \neg r(X). \\
r(a). \\
r(h(X)) & : \neg r(X).
\end{align*}
\]

A regular system which is in some sense equivalent to the program can be obtained if we include the predicate symbols and the atoms in the productions. The set of axioms corresponds to the facts of the program

\[
\{ pga, ra \}
\]

The set of productions corresponds to the other clauses (in this particular program all productions have only one premise)

\[
\{ p\zeta \rightarrow pf\zeta, r\zeta \rightarrow ph\zeta, r\zeta \rightarrow rh\zeta \}
\]

A possible deduction is

\[
ra, pha, pfha, pfpha
\]

It corresponds to the functional word \(ffa\) (we have applied successively the first axiom, the second production, and twice the first production).

Büchi results could have been used to prove the equivalence between monadic programs and regular languages (Theorem 1). However, they are not directly applicable to sub-regular languages like \(\mathcal{F}1\) for which special (more restricted) forms of regular systems would be needed. One problem is that some interesting properties are not invariant with regard to the transformation suggested above. For instance, with the inclusion of the predicate symbols, the functional language corresponds to an interpreter, which in this case is

\[
\begin{align*}
s(p(g(a))). \\
s(p(f(X))) & : \neg s(p(X)). \\
s(p(h(X))) & : \neg s(r(X)). \\
s(r(a)). \\
s(r(h(X))) & : \neg s(r(X)).
\end{align*}
\]

so that we always get the functional language of an almost acyclic program.
8. Conclusions and open problems

In this paper we have studied the class of monadic logic programs. We have proved that the sequences of functor symbols (the functional complexity) in the success set of any such program, forms a regular language. As a consequence, the SUCCESS problem for such programs is decidable.

Although the class of monadic logic programs is decidable, most slight generalizations seem to make it undecidable. That happens, for instance, for logic programs containing only
- monadic functors (but not monadic predicate symbols),
- monadic predicate symbols (but not monadic functors).

It would be interesting to find simple classes of logic programs that correspond to more complex, but decidable, classes of languages (for the class of context free languages, it may be worthwhile to look at the structure of DCG's).

Several subclasses of the monadic class were considered. With one exception, the functional complexity of every subclass coincides with the class of regular languages. The exception is the "simple" program class whose functional language corresponds to the regular languages with star-height not exceeding 1 (as explained in the beginning, the star-height of a language is not a trivial property; the star-height of a regular expression is a trivial property).

Another open problem is the correspondence between the properties of the dependence graph and the classes \( \mathcal{F}_n \) with \( n \geq 2 \). For instance, are there natural limitations on the program recursivity (that is, on the cycles of the dependence graph) so that the corresponding class of programs is \( \mathcal{F}_2 \)?

As a side result we have defined two new kinds of automata – inverse and intersection automata – that correspond to two of the subclasses considered and that may be useful for other purposes. Also, a new algorithm for \( \epsilon \)-edges elimination in finite automata was described (the set of states is usually not altered by the algorithm).

The following diagram summarizes three potential areas of future work (\( \mathcal{F}1 \) denotes the class of regular languages with a star-height not exceeding 1). It would be nice (but possibly difficult or even impossible) to have relatively simple classes of logic programs corresponding to languages in A (regular but not \( \mathcal{F}1 \)), in B (decidable but not regular) and, as much as possible, to the decidability border C.

```
  Regular     | B          | Undecidable |
             | C          |
  \( \mathcal{F}1 \) | A          |
```

Acknowledgements

We would like to express our gratitude to a referee of the first version of this paper for calling our attention on the work of Büchi [3].

References