SMALLEST HORN CLAUSE PROGRAMS

P. DEVIENNE, P. LEBÈGUE, A. PARRAIN, J. C. ROUTIER, AND J. WÜRTZ

▷   The simplest nontrivial program pattern in logic programming is the
following:

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

where *fact*, *goal*, *left*, and *right* are arbitrary terms. Because the well-known
append program matches this pattern, we will denote such programs
"*append*-like."

In spite of their simple appearance, we prove in this paper that
termination and satisfiability (i.e., the existence of answer-substitutions,
called the *emptiness problem*) for *append*-like programs are undecidable.
We also study some subcases depending on the number of occurrences of
variables in *fact*, *goal*, *left*, or *right*.

Moreover, we prove that the computational power of *append*-like pro-
grams is equivalent to the one of Turing machines; we show that there
exists an *append*-like universal program. Thus, we propose an equivalent of
the Böhm–Jacopini theorem for logic programming. This result confirms
the expressiveness of logic programming.

The proofs are based on program transformations and encoding of
problems, unpredictable iterations within number theory defined by J. H.
Conway, or the Post correspondence problem.   ◁

1. INTRODUCTION

The study of minimal patterns of programming languages allows

- to extract useful properties for improving larger programs (for example, new technics of compilation),

- to strengthen the power of the language.

In Horn clause languages, the simplest nontrivial pattern is built with one fact, one two-literal recursive Horn clause (in the following we will say *binary*), and one goal:

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

where *fact*, *left*, *right*, and *goal* are arbitrary terms.

We will use this pattern many times in this paper, sometimes with a simple reference to *fact*, *left*, *right*, or *goal*. We will refer to such programs as *append*-like programs according to the most famous program matching this pattern:

Example 1.1.

$$\begin{cases} \mathit{append}([\ ], L, L) \leftarrow . \\ \mathit{append}([H \mid L], LL, [H \mid LLL]) \leftarrow \mathit{append}(L, LL, LLL). \quad \square \\ \leftarrow \mathit{append}(?,?,?). \end{cases}$$

While for simple examples good intuition on the behavior (halting and existence of solutions) of an *append*-like program is possible, the nonlinearity of the terms may cause high-complexity phenomena. Indeed, as we will see here, in spite of their structural simplicity, the computational power of *append*-like programs is the same as that of Turing machines.

The two first important problems are the *halting problem* and the *emptiness problem*, that is, the problem of the existence of at least one solution (answer-substitution). Different behaviors of *append*-like programs are possible depending on the goal: finite or infinite computation; empty, finite, or infinite set of solutions.

Schmidt-Schauβ [47] has shown that the two problems are decidable when *goal* and *fact* are ground.[1] This result is a corollary of his work on the implication of clauses, or equivalently, on the decision problem of clause sets consisting of one clause and some ground units (one-literal clause) (see also [36]). Dauchet et al. [11] and Devienne [17] studied the linear[2] case and proved it decidable as well. They used a new technic based on weighted directed graph (an extension of the directed graphs). Bibel et al. [2] considered the emptiness problem and proved it decidable for some particular cases.[3] They denoted this problem as the *cycle unification*

---

[1] A term t is said to be *ground* when it does not contain any variable occurrence [34].
[2] A term t is said to be *linear* when each variable occurs at most once.
[3] They consider cases where there exists a substitution σ such that $\sigma \mathit{left} = \mathit{right}$ or $\mathit{left} = \sigma \mathit{right}$, called left (resp. right) matching cycle.

problem, that is, a unification of the goal—which begins the cycle, and the fact—which terminates the cycle, through the binary Horn clause—which defines the cycle. For particular cycle unification classes, see [45, 51].

In this paper, we will show that the two problems are undecidable for *append*-like programs. The proof technic of [19,20] is based on an original encoding of the unpredictable iterations of Conway within number theory [8] which are close to Minsky machines [39]. An alternative proof of undecidability of the emptiness problem can be found in [28]. It has been made independently, and it is based on an encoding of the Post correspondence problem. We will present both proofs. We also study some particular subcases defined by the number of occurrences of variables in terms (linearity).

Another crucial question is the computational power of *append*-like programs. Dauchet [10] proved that it is possible to simulate any Turing machine with only one regular[4] left-linear rewrite rule. In [49], it is shown that every computable function is characterizable by a program consisting only of facts and binary Horn clauses. Another encoding of Turing machines using binary programs can be found in [4]. And in [41], a meta-interpreter has been written using only one fact, one goal, and two binary recursive clauses. In pure Datalog, Shmueli has proved that a single recursive predicate (not clause) is sufficient [46].

In this paper, we establish that all computations on Minsky machines can be expressed by an *append*-like program [21]. The class of *append*-like programs is Turing-complete. The proof uses the encoding of Conway functions and logical reductions on meta-programs.

This theorem is equivalent to the Böhm–Jacopini theorem for declarative languages. The Böhm–Jacopini theorem establishes that for imperative languages, every flowchart is equivalent to a while-program with one occurrence of `while-do`, provided additional variables are allowed (for more details, see [27]). This proof is constructive and usually cited as the mathematical justification for structured imperative programming.

We will show that in Horn clause languages, any program can be automatically transformed into another one composed of one binary Horn clause and two unit clauses. This transformation preserves both termination and solutions (answer-substitutions on the original variables). This shows the expressive power of a single Horn clause, and can be used as a theoretical tool for decision problems in theorem proving.

The paper is organized as follows. Section 2 states the main results. In Section 3, we introduce binary Horn clauses and resolution. In Section 4, the Minsky machine formalism and the unpredictable iterations of Conway are presented. In the next section, it is shown how they can be simulated by binary clauses. We present the halting and emptiness problems and some subcases in Sections 6 and 7, respectively. Section 8 is about meta-interpreters and the computational power of *append*-like programs. Some results about ternary (three-literal) programs, close to clause implication problem, are presented in Section 9. A conclusion summarizes the results.

---

[4]A rule is *regular* iff it is left-linear and nonoverlapping (i.e., there are no nontrivial critical pairs).

2. MAIN RESULTS

The main result of this paper is about the computational power of *append*-like programs.

Main Result (Theorem 8.1, p. 33). *There exists a meta-interpreter for Horn clauses in the form of a program with only one binary Horn clause, a fact and goal which, given as input a Horn clause program P, has the same solutions as P and terminates if and only if P terminates.*

The two other main results are, in fact, consequences of this theorem:

Result 1 (Theorem 6.2, p. 17). *There exists an explicitly constructable right-linear binary Horn clause for which the halting problem, according to SLD resolution, is undecidable. The resolution can be applied with or without occur-check.*

Result 2 (Theorem 7.1, p. 20). *For a program of the form*

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

where fact and right are linear, the emptiness problem is undecidable.

Due to the used proof technics, we will prove Result 1 and Result 2 first. The proof of the Main Result will use the principles introduced before.

3. PRELIMINARIES

We assume the reader to be familiar with the notions of unification and resolution introduced in [34].

The goal of this section is to present indexation of variables and how to express, in term of equations, the resolution of *append*-like program.

3.1. Binary Horn Clause

Let $\mathscr{F}$ be a set of function symbols (which contains at least two constants and one symbol whose arity is greater than 1), and *Var* an infinite countable set of variables. We denote by $\mathscr{T}(\mathscr{F}, Var)$ the set of terms built from $\mathscr{F}$ and *Var*.

Definition 3.1. Binary recursive Horn clauses have the following form:

$$p(l^1, \ldots, l^n) \leftarrow p(r^1, \ldots, r^n)$$

where l^i and r^i are any terms of $\mathscr{T}(\mathscr{F}, Var)$.

In the following, we will often abbreviate "binary recursive Horn clause" by "binary clause"; we also refer to it as the rule.

A binary clause $l \leftarrow r$ is said to be *right-linear* (resp. *left-linear*) if each variable occurs at most once in the body part r (resp. the head part l).
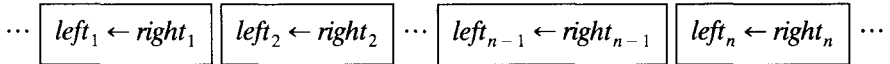
For example, "append([X |L], LL, [X |LLL]) ← append(L,LL,LLL))." is a right-lin-ear binary clause.

3.2. Variable Indexation

It is well known that, during resolution, formal variables of a clause are renamed to fresh variables. We introduce fresh variables by adding a subscript to the formal variables. This subscript will denote the number of the inference.

ith inference: $append([\ X_i\ |\ L_i\ ], LL_i, [\ X_i\ |\ LLL_i]) \leftarrow append(\ L_i, LL_i, LLL_i)$.

The sequence of inferences using the clause "$left \leftarrow right$" can be drawn in the form of a series of dominoes:

$$\cdots \boxed{left_1 \leftarrow right_1} \boxed{left_2 \leftarrow right_2} \cdots \boxed{left_{n-1} \leftarrow right_{n-1}} \boxed{left_n \leftarrow right_n} \cdots$$

The ith domino can be followed by an $(i + 1)$th one, if the terms $left_{i+1}$ and $right_i$ can be unified (and this is compatible with those of the other iterations). Hence, applying n times this binary clause is equivalent to solving the following system of equations:

$$\{left_{i+1} = right_i \mid i \in [1, n - 1]\}.$$

Example 3.1. Applying n times the *append* clause is equivalent to solving the system

$$\{append([\ X_{i+1}\ |\ L_{i+1}\ ], LL_{i+1}, [\ X_{i+1}\ |\ LLL_{i+1}])$$

$$= append(\ L_i, LL_i, LLL_i) \mid i \in [1, n - 1]\},$$

that is, in solved form

$$\forall i \in [1, n - 1] \begin{cases} L_i = [\ X_{i+1}\ |\ L_{i+1}\ ] \\ LL_i = LL_{i+1} \\ LLL_i = [\ X_{i+1}\ |\ LLL_{i+1}\ ]. \end{cases} \qquad \square$$

To express the whole resolution, the goal and the fact must be taken into account. Thus, applying n times the rule "$left \leftarrow right$." starting with the goal "$\leftarrow goal$." and checking whether there exists a solution at the nth iteration with the fact "$fact \leftarrow$." is equivalent to solving

$$\begin{cases} goal = left_1 \\ left_{i+1} = right_i, \quad i \in [1, n - 1] \\ right_n = fact. \end{cases}$$

This indexation of variables and the modeling of resolution through equations will be one of the basic notions in the following sections.

4. MINSKY MACHINES AND CONWAY ITERATIONS

In the following, the expression, "*It is undecidable whether or not ...*" stands for "*There exists no algorithm that always decides, whether or not ... .*"
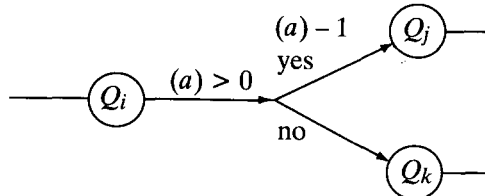
4.1. Minsky Machines

4.1.1. Presentation. The Minsky machines [39, 8] are deterministic machines with registers and instructions. Registers (finitely many of them) can hold arbitrary large nonnegative integers. A machine executes a program composed of instructions sequentially. Instructions are labeled by $Q_1, Q_2, \ldots, Q_n$ (for a program of n instructions). Three kinds of instructions are possible:

- "Halt": Stop the machine.

- "Successor": At step Q_i, add 1 to some register a and proceed to the next step Q_j (where (a) denotes the value of the register a):
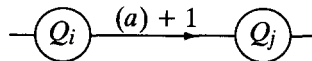
- "Decrement or jump": At step Q_i, if $(a) > 0$, then subtract 1 from register a and proceed to the next step WQ_j; else simply go to step Q_k:

These machines have the same computational power as Turing machines (two registers are sufficient [39]). For any partial recursive function f, there exists a Minsky machine which, started with register contents $n, 0, 0, \ldots$ (n will be called the *input* of the machine M), reaches the "Halt" instruction with register contents $f(n), 0, 0, \ldots$, if $f(n)$ exists and does not halt otherwise.

If the computation is finite, $M(n)$ denotes the content of the first register (i.e., $f(n)$), which we call the result of the Minsky machine's computation for initial register values (the input of the machine) $n, 0, 0, \cdots$. Otherwise (the computation does not terminate), $M(n)$ will be infinite.

Let us state some usual definitions and properties:

- The domain $Dom(M)$ of a Minsky machine M is: $\{n \in \mathbb{N} \mid M(n) \text{ is finite}\}$.

- A Minsky machine M is said to be total iff its domain is $\mathbb{N}$.

- Given a Minsky machine, it is undecidable whether or not this machine is total.

- Given a Minsky machine M, it is undecidable whether or not a given n belongs to $Dom(M)$.

4.1.2. A Particular Class of Minsky Machines. In the proofs of further sections, we use a particular class of Minsky machines given by the following definition.

Definition 4.1. Given a Minsky machine $M_?$, a new machine M, with α a fixed natural number, and $n \in \mathbb{N}^*$ as input, can be constructed as follows:

1. Compute $\alpha \times n$ and put it into a new register r.
2. If $(r) = 0$, then goto 5; else subtract one from r.
3. Execute one instruction of $M_?(0)$.
4. If $M_?(0)$ has reached the "Halt" instruction, then go into "an infinite loop"; else goto 2.
5. Put zero in all the registers and halt.

Moreover, we force $M(0)$ to be finite and equal 0. M is called a *linear* and *null* Minsky machine.

Property 1. Let M be a linear and null Minsky machine; the two following assertions stand:

- *null*: $0 \in Dom(M)$ and all the registers contains 0 at the final computation step (after the "Halt" instruction), in particular, the associated partial function, f, satisfies: $\forall n \in Dom(f)$, $f(n) = 0$.

- *linear*: For all input $n \in Dom(M)$, if $n > 0$, then $M(n)$ is computed in less than $\alpha \times n$ steps.

It is straightforward that such Minsky machines do exist. Simply consider the machine with one register, which decrements it until it reaches 0. It is clearly linear (with $\alpha = 2$) and null.

Theorem 4.1. It is undecidable whether or not a linear and null Minsky machine M is total.

PROOF.[5] Let $M_?$ be the Minsky machine from which M is constructed.

A natural integer n belongs to the domain of M iff the "infinite loop" is not reached, that is, $M_?(0)$ needs more than $\alpha \times n$ steps to be computed. By construction, this null Minsky machine is total iff $M_?(0)$ does not terminate. This is undecidable. It remains to prove that M is linear.

Let us compute the complexity of M for any $n \in Dom(M)$. Step 1 can be done in $((\alpha + 1) \times n)$ instructions. M reaches step 5 after $(2\alpha \times n)$ instructions. Once in this step, the sum of all the contents of the k registers of $M_?$ is, by construction, at most $(\alpha \times n)$. Consequently, it takes at worst $(\alpha \times n + k)$ instructions to put 0 in all the registers of $M_?$, then of M. Hence, the complexity of M is $((4\alpha + 1) \times n + k)$.

Thus, by construction, the linear null Minsky machine M is total iff $M_?(0)$ does not stop, but this is undecidable. $\square$

Definition 4.2. A set Σ is said to be *recursively enumerable* iff it is the domain of a Turing machine (or a Minsky machine).

---

[5]We would like to thank Prof. Jean-Paul Delahaye for the basic idea of this proof and previous definition.

From definitions, we can deduce the following property:

Property 2. Every recursively enumerable set containing {0} is the domain of a null Minsky machine.

Definition 4.3. A recursively enumerable set Σ is said to be *linear* if there exists a linear and null Minsky machine the domain of which is Σ.

Corollary 4.1. It is undecidable whether a linear recursively enumerable set is equal to $\mathbb{N}$.

PROOF. By application of Theorem 4.1.   □

4.2. Conway Unpredictable Iterations

In the previous section, we have considered the Minsky machines that can be seen as an arithmetization of Turing machines since the tape is replaced by registers with integer values. This section deals with the work of the mathematician J. H. Conway. He proposed an encoding in terms of numeric functions of the Minsky machines. It results from the study of a generalization of the Collatz conjecture. Including the characterization of recursively enumerable sets, the results obtained for the Minsky machines can be extended to Conway functions.

4.2.1. The Collatz Conjecture. This conjecture asserts that, given a positive integer n, the program below always terminates:

```
While n > 1 Do
    If n is even
        Then n ← n/2
        Else n ← 3n + 1
    Endif
EndWhile
```

The exact origin of this conjecture—also called "Syracuse conjecture" or "$3x + 1$ problem" [30, 31]—is not clearly known. This problem is credited to Lothar Collatz at the University of Hamburg in the 1930s.

Nabuo Yoneda at the University of Tokyo has checked the conjecture for all $n < 2^{40}$. The behavior of the Collatz series, that is, the sequence of all the numbers successively obtained during the execution of the above program, seems to be random. While it takes only 10 steps to meet 1 from 26, it takes 111 steps from 27:

$$26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$
$$\underbrace{27 \rightarrow 82 \rightarrow 41 \rightarrow \cdots \rightarrow 4 \rightarrow 2 \rightarrow 1}_{111 \text{ steps}}.$$

The conjecture may be formulated as follows:

Conjecture 4.1. (Collatz). Let g be the function defined as follows:

$$g(n) = \begin{cases} \frac{1}{2}n & (n \equiv 0 \bmod 2) \\ 3n + 1 & (n \equiv 1 \bmod 2) \end{cases}$$

for every natural n, there exists $k \in \mathbb{N}$ such that $g^{(k)}(n) = \underbrace{g(g \cdots (g(n)) \cdots)}_{k} = 1$.

Conway has considered the more general functions, which we will call in the following "*Conway functions*"[6]:

$$g(n) = \begin{cases} a_0 n + b_0 & (n \equiv 0 \bmod p) \\ \cdots & \cdots \\ a_k n + b_k & (n \equiv k \bmod p) \\ \cdots & \cdots \\ a_{p-1} n + b_{p-1} & (n \equiv p-1 \bmod p) \end{cases}$$

where p is a positive integer and a_k and b_k are rational numbers greater than 0 such that $g(n)$ is always a natural number. He studies the iterates $g^{(m)}(n)$. Conway proved that even if the b_k are all equal to zero, the behavior of such functions is unpredictable. This was achieved by a translation of the Minsky machines into Conway functions. We will also define by analogy with the null Minsky machines so-called *null* functions.

4.2.2. Presentation. Conway considered the class of periodically piecewise linear functions $g : \mathbb{N} \to \mathbb{N}$ having the structure

$$\forall k, 0 \le k \le p-1, \qquad g(n) = a_k n \ (n \equiv k \bmod p)$$

where $a_0, \ldots, a_{p-1}$ are rational numbers such that $g(n) \in \mathbb{N}$. These are exactly the functions $g : \mathbb{N}^* \to \mathbb{N}$ such that $g(n)/n$ is periodic ($\mathbb{N}^*$ denotes $\mathbb{N} \setminus \{0\}$).

Theorem 4.2. (Conway; see [8]). *If f is any partial recursive function, there exists a function g such that*

1. $g(n)/n$ *is periodic* mod p *for some p and takes rational values*
2. $\forall_n \in \mathbb{N}, n \in Dom(f)$ *iff* $\exists(m, j) \in \mathbb{N}^* \times \mathbb{N}$, *such that* $g^{(m)}(2^n) = 2^j$
3. $g^{(m)}(2^n) = 2^{f(n)}$ *for the minimal $m \ge 1$ such that $g^{(m)}(2^n)$ is a power of* 2.

The first point expresses that g is a Conway function. The second point shows how to characterize a member of the domain of a partial recursive function f from this function. The last explains how it is possible to compute the value of $f(n)$ through iterations of g. This means that the Conway functions are as expressive as the Turing or Minsky machines. This is not a surprise since, as we will see now, the Conway functions are a direct translation of the Minsky machines.

The following proof explicitly shows the connection between Minsky machines and Conway functions, and this connection is important in the following.

PRINCIPLE OF PROOF. Conway showed that with every Minsky machine, it is possible to associate such a function g which simulates step by step the behavior of this machine. In fact, he explains how to construct this g from the Minsky machine:

- With register r_i, we associate a prime number p_i and characterize the value (r_i) of this register by $p_i^{(r_i)}$.

- With each step Q_j, we associate a prime number P_j.

---

[6] In the following, g will denote a Conway function.

- The current situation of the machine, characterized by the contents k_i of the registers r_i and by the current step Q_j, is expressed by an integer of the following form:

$$p_1^{(r_1)} * p_2^{(r_2)} * \cdots * p_n^{(r_n)} * P_j.$$

Now, let us consider how to express the instructions. This can be done in a very natural way if the above encoding of the current situation by a number is well understood:

- For the "Successor" instructions:
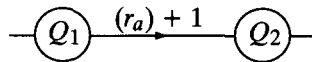
If step Q_1 is characterized by prime number P_1, step Q_2 by P_2, and register r_a by p_a, this instruction may be translated as the multiplication of the current situation of the machine by the factor:

$$\frac{P_2}{P_1} \times p_a$$

which means: "From step Q_1 ($\times 1/P_1$), proceed to Q_2 ($\times P_2$) and add one to register r_a ($\times p_a$)."

- For the "Decrement or jump" instructions:
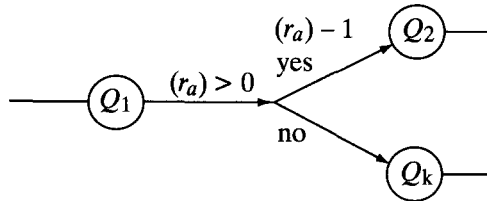
With the above conventions and if the prime number P_k is associated with Q_k, these instructions can be expressed by the factors

$$\frac{P_2}{P_1} \times \frac{1}{p_a} \quad \text{or} \quad \frac{P_k}{P_1}.$$

The choice between these two factors corresponds, respectively, to the cases $(r_a) > 0$ and $(r_a) = 0$. It will be achieved by the "mod p" in the definition of the g functions. It will be similar for the detection of "Am I at step P_j?."

For every instruction of the Minsky machine to be coded, we have to create the associated factors, and then from these factors, determine the period p (which is just $p_1 \times \cdots \times p_n \times P_1 \times \cdots \times P_q$) such that $g(n)$ always remains an integer, and finally compute all the a_k.

We can see that each instruction of the associated Minsky machine corresponds to one iteration of the function.   □

Since there exists a direct translation between Minsky machines and Conway functions, we will speak in the following about a Conway function *associated* with a Minsky machine (and conversely).

As an example of encoding of a recursive function into a Conway machine, in [26], Conway and Guy have detailed how to produce a prime number generator Conway functions from a machine.

Remark 4.1. By construction, the number of iterations required from $g(2^n)$ to reach $2^{f(n)}$ is equal to the number of elementary instructions used by the associated Minsky machine to produce $f(n)$ from n.

4.2.3. Conway Relations. We have seen that Minsky machines and Conway functions are strongly connected. Then it is natural to extend some definitions for Minsky machines to Conway functions.

Definition 4.4. Let g be a Conway function; the *domain* of g is

$$Dom(g) = \{n \in \mathbb{N} \mid \exists (k, p) \in \mathbb{N}^* \times \mathbb{N}, g^{(k)}(2^n) = 2^p\}.$$

A Conway function g is said to be *total* if its domain is $\mathbb{N}$.

Considering the Conway function associated to the null Minsky machines, we define the *null* Conway functions as follows:

Definition 4.5. A Conway function g is said to be *null* if

- 0 belongs to its domain,
- for every n in the domain of g, the first power of 2 reached by iterations from $g(2^n)$ is 2^0.

Since we have seen that to each instruction of a machine there corresponds one iteration (see Remark 1), it is reasonable to measure the "complexity" of the functions as the number of iterations:

Definition 4.6. A null Conway function is said to be *linear* if there exists a natural integer α such that for every $n \in Dom(g)$, 2^0 is reached from 2^n in less than $\alpha \times n$ iterations of g.

We will study these null functions more precisely and extract some properties to define so-called *Conway relations*. We will first analyze the behavior of the *negative iterations* of Conway functions.

Definition 4.7. Let g be a Conway function; the kth negative iterate of g on n is defined as

$$\forall k \in \mathbb{N}, \qquad g^{(-k)}(n) = \{m \in \mathbb{N} \mid g^{(k)}(m) = n\}.$$

Comment 4.1. $g^{(-1)}(n)$ (and therefore $g^{(-k)}(n)$ for any $k \in \mathbb{N}$) may be a set since n can be the range of many m for g.

Proposition 4.1. Let g be a null Conway function and n an arbitrary integer; the only power of 2 reachable (if any) from 2^n by iterative applications of g is 2^0. By negative iterations of g from 2^0, exactly the 2^i for all $i \in Dom(g)$ are reached.

PROOF. Since g is null, 0 belongs to its domain. Then there exists $k > 0$ such that $g^{(k)}(2^0) = 2^{f(0)} = 2^0$, and k is the smallest positive integer such that $g^{(k)}(2^0)$ is a power of 2. So it follows that no other power of 2 can be reached by iterations of g from 2^0.

Now, let us consider the definition of a null Conway function. For a given n, if it does not belong to the domain of g, no power of 2 will be reached. Conversely, if n belongs to it, since the first power of 2 reached by iterations from 2^n *is* $2^{f(n)} = 2^0$, then it is the only possible one.

The second part of the proposition follows immediately from Definitions 4.4 and 4.7. $\square$

Thus, if an integer n belongs to the domain of a null function g, then there exists only one path between 2^n and 2^0 using positive or negative iterates (we will neglect the loops on 2^0 since they do not contain other powers of 2). The existence of such a path is fully determined by n being in the domain of g.

Thus, we can define the *Conway relation*:

Definition 4.8. Let g be a null Conway function; we define the *Conway relation* associated to g, and we denote it by $\equiv_g$, obtaining the relation such that: $2^m \equiv_g 2^n$ if and only if there exists $k \in \mathbb{Z}$ such that $g^{(k)}(2^m) = 2^n$ $(m, n \in \mathbb{N}^*)$.

It is easy to check that $\equiv_g$ is really an equivalence relation: transitivity, reflexivity, and symmetry of $\equiv_g$ are straightforward.

Now, it is possible to characterize the recursively enumerable sets containing 0, that are the domains of null functions, with these relations.

Proposition 4.2. For every recursively enumerable set Σ containing 0, there exists a Conway relation $\equiv_g$ such that: $\Sigma = \{n \in \mathbb{N} \mid 2^n \equiv_g 2^0\}$

PROOF. The recursively enumerable sets containing 0 are the domains of the null Minsky machines, and consequently of the null Conway functions. If g is the function the domain of which is Σ, then $\equiv_g$ satisfies the proposition. $\square$

This proposition is crucial for the following. In many proofs, we will create some recursively enumerable sets from 2^0, and use the negative iterations of null Conway functions in order to enumerate all the elements of these sets. Then we will use known undecidable properties concerning these sets.

5. RECURSIVELY ENUMERABLE SETS AND BINARY HORN CLAUSES

In this section, we will establish the relationship between binary Horn clauses and the Conway functions. A Conway function g is expressed by relations like $g(n) = a_k n$ with $n = \alpha p + k$ and $0 \le k \le p - 1$, and such that $a_k n$ is always an integer. So a function g associates, with a given number of the form $\alpha p + k$, another number of the form $(a_k \alpha)p + (a_k k)$ with α, k, $a_k \alpha$ and $a_k k$ in $\mathbb{N}$.

We will first prove that it is possible to express with a binary clause and a goal every relation that associates a number $ai + b$ with another number $ci + d$ where a, b, c, d are integers. We will then prove that an encoding of the Conway functions with a binary clause is possible. This encoding will be described explicitly. We will deduce a characterization of recursively enumerable sets through a binary clause.

5.1. The Encoding

Example 5.1. Let us consider the following program:

$$\begin{cases} p(s(X), s(s(Y))) \leftarrow p(X, Y). \\ \leftarrow p(U, U). \end{cases}$$

It creates the following equalities between indexed variables:

$$X_i = s^i(Y_i) \text{ and } Y_i = X_{2i}.$$

The size of Y increases by 2, while the size of X increases only by 1. □

In general, we will establish that any relation of the form

$$X_{ai+b} = Y_{a'i+b'}$$

can be obtained with a binary clause and a nonlinear goal. The encoding will be very similar to the one of the example: it requires the use of one function symbol. However, in order to improve the reading, we will use the list constructor instead of the function $s(\_)$ of the example.

In fact, we will only consider relations such as

$$X_{ai+b} = X_{a'i+b'}$$

since they are sufficient in the following. The production of relations between two distinct variables X and Y will be obvious from the following.

Proposition 5.1. For every natural integer a, a', b, b', there exist a variable X, a right-linear binary clause "$p(left) \leftarrow p(right)$," and a goal "$\leftarrow p(goal)$" such that

$$(\{goal = left_1\} \cup \{right_i = left_{i+1} \mid \forall i > 0\}) \!\uparrow_X \equiv \{X_{ai+b} = X_{a'i+b'} \mid i > 0\}$$

where $\mathscr{S} \!\uparrow_X$ is the projection onto the X_i of the equations expressed in $\mathscr{S}$.

PROOF. Let us consider first the case where $a' = 1$ and $b = b' = 0$.

$$\begin{cases} p\!\left(\left[\overbrace{Z, \_, \ldots, \_}^{a} \mid L\right], [X \mid LL]\right) \leftarrow p(L, LL). \\ \leftarrow p(L, L). \end{cases}$$

As in the previous example, the size of the first argument of the Horn clause decreases by a, while the one of the second decreases by 1.

The equal arguments in the goal generate the equality of the two arguments. So we deduce the relation $Z_i = X_{ai}$.

Let us assume now that we want to establish a relation such as $Z_i = X_{ai+b}$. We have to shift the equality between the previous terms. This can be achieved by the goal

$$\leftarrow p\!\left(\left[\underbrace{\_, \ldots, \_}_{b} \mid L\right], L\right).$$

The relations in this case will affect the Z_i only from $i \geq b$.

Now, if we combine two relations $Z_i = X_{ai+b}$ and $Z_i = X_{a'i+b'}$, the transitive closure of these relations and the projection onto the variable X provides the intended relations

$$X_{ai+b} = X_{a'i+b'}$$

obtained by the program

$$\begin{cases} p\left(\left[\overbrace{Z, \_, \dots, \_}^{a} \mid L1 \right], [X \mid L2], \left[\overbrace{Z, \_, \dots, \_}^{a'} \mid L3 \right], [X \mid L4] \right) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \leftarrow p(L1, L2, L3, L4). \quad \Box \\ \leftarrow p\left(\left[\underbrace{\_, \dots, \_}_{b} \mid L \right], L, \left[\underbrace{\_, \dots, \_}_{b'} \mid LL \right], LL \right). \end{cases}$$

Remark 5.1. Another encoding of the relation $X_{ai+b} = X_{a'i+b'}$, when $b < a$ and $b' < a'$, is possible:

$$\begin{cases} p\left(\left[\overbrace{\underbrace{\_, \dots, Z, \_, \dots}_{b}}^{a} \mid L1 \right], [X \mid L2], \left[\overbrace{\underbrace{\_, \dots, Z, \_, \dots}_{b'}}^{a'} \mid L3 \right], [X \mid L4] \right) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \leftarrow p(L1, L2, L3, L4). \\ \leftarrow p(L, L, LL, LL). \end{cases}$$

The encoding of the Conway functions by a binary recursive clause is now straightforward.

Proposition 5.2. For every Conway function g, there exist a variable X, a right-linear binary clause "$p(left) \leftarrow p(right)$," and a goal "$\leftarrow p(goal)$" such that

$$(\{goal = left_1\} \cup \{right_i = left_{i+1} \mid \forall i > 0\}) \uparrow_X \equiv \{X_n = X_{g(n)} \mid \forall n > 0\}.$$

PROOF. Let g be a Conway function. It is defined by some $a_0, a_1, \dots, a_{p-1}$. As was previously discussed, g can be decomposed into a finite number of relations of the form $(X_{ai+b} = X_{a'i+b'})_{i>0}$, where a, b, a', and b' are integers. From the previous proposition, it is possible to associate with each of these relations a binary Horn clause and goal. All of these can be merged in only one clause and one goal which satisfy the proposition. $\Box$

Example 5.2. The Collatz program can be translated into equivalence relations on $Var \times \mathbb{N}$:

$\forall k \in \mathbb{N}$ If k is even Then $X_k = X_{k/2}$ Else $X_k = X_{3k+1}$.

Let f be the function such that $\forall i > 0$, $f(2i) = i$ and $f(2i + 1) = 6i + 4$. Since there does not exist some $k \in \mathbb{N}$ such that $f^{(k)}(1) = n$ ($\forall n > 4$), we can extend the previous relation to the following system of equations:

$$\begin{cases} X_i = X_{2i} \\ X_{2i+1} = X_{3(2i+1)+1}. \end{cases}$$

But we have seen that such relations can be expressed through a binary recursive clause. The following clause is constructed in a way slightly different from that described before since we have grouped two arguments into one.

$$\begin{cases} p\left(\overbrace{[X \mid U]}^{L_1}, \overbrace{[Y, X \mid V]}^{L_2}, \overbrace{[\_,\_,\_,Y,\_,\_ \mid W]}^{L_3} \right) \leftarrow p(U, V, W). \\ \leftarrow p(Z, Z, Z). \end{cases}$$

From the general goal "$\leftarrow p(L_1, L_2, L_3)$.," through the inferences the solved systems of equations increase as

1. $L_1 = [X_1 \mid U_1]$ $L_2 = [Y_1, X_1 \mid V_1]$
2. $L_1 = [X_1, X_2 \mid U_2]$ $L_2 = [Y_1, X_1, Y_2, X_2 \mid V_2]$
$\vdots$ $\vdots$ $\vdots$
n. $L_1 = [X_1, X_2, \ldots, X_n \mid U_n]$ $L_2 = [Y_1, X_1, Y_2, X_2, \ldots, Y_n, X_n \mid V_n]$

        1. $L_3 = [\_,\_,\_,Y_1,\_,\_ \mid W_1]$
        2. $L_3 = [\_,\_,\_,Y_1,\_,\_,\_,\_,\_,Y_2,\_,\_ \mid W_2]$
        $\vdots$ $\vdots$
        n. $L_3 = [\_,\_,\_,Y_1,\_,\_,\ldots,\_,\_,\_,Y_n,\_,\_ \mid W_n]$.

Therefore, after n iterations, we have

$$L_2 = [Y_1, X_1, Y_2, X_2, Y_3, X_3, \ldots, Y_n, X_n \mid V_n]$$
$$L_1 = [X_1, X_2, X_3, X_4, X_5, X_6, \ldots, X_{n-1}, X_n \mid U_n]$$
$$L_3 = [\_,\_,\_,Y_1,\_,\_,\ldots,\_,Y_n,\_,\_ \mid W_n].$$

Then, with the goal $\leftarrow p(Z, Z, Z)$, we force the equalities

1. $L_1 = L_2 \Rightarrow X_{2i+1} = Y_i$ and $X_{2i} = X_i$
2. $L_1 = L_3 \Rightarrow X_{6i+4} = Y_i$,

that is,

$$X_i = X_{2i} \text{ and } X_{2i+1} = X_{6i+4}.$$

With a goal of the form

$$\leftarrow p\left(\underbrace{[\#,\_,\ldots,\_,{}^\flat \mid L]}_{n}, \underbrace{[\#,\_,\ldots,\_,{}^\flat \mid L]}_{n}, \underbrace{[\#,\_,\ldots,\_,{}^\flat \mid L]}_{n} \right).$$

we obtain that $X_1 = \sharp$ and $X_n = \flat$. Therefore, the resolution is finite if and only if a unification fails because of $X_n \neq X_1$, that is, if the Collatz program with the input n terminates. In other words, the Collatz conjecture is equivalent to the assertion that, given any n, if the goal is of the above form, then the resolution is finite. $\square$

5.2. A Binary Clause and Recursively Enumerable Sets

In Section 4.2.3, we defined the notion of Conway relations. We have shown that they allow to characterize the recursively enumerable sets containing 0. Now, according to the previous paragraph, we are going to associate with each such set a program consisting of one binary right-linear recursive clause and one goal.

Theorem 5.1. Let $\sharp$ be a special symbol. For every recursively enumerable set Σ containing 0, there exist a right-linear binary clause and a goal such that a natural integer n belongs to Σ iff after a certain number of SLD resolution steps, the first argument of the initial goal becomes instantiated to a list where the (2^n)th element is $\sharp$.

PROOF. According to the two previous propositions, let X be the variable which codes the Conway relation associated with Σ (as in Proposition 4.2). The list L is built linearly as $[X_1, X_2, \ldots, X_n, \ldots]$ with all the X_i connected by the relations $X_i = X_{g(i)}$. Consequently, according to Proposition 4.2, we deduce that

$$\Sigma = \left\{ n \in \mathbb{N} \mid X_{2^n} \equiv_g X_1 \right\}.$$

If variable X_1 is instantiated to $\sharp$, then this mark will be propagated to all X_{2^n} such that n belongs to Σ. $\square$

*Remark 5.2. Let Σ be a recursively enumerable set; the above theorem associates a clause and a goal with Σ. By construction (see Proposition 5.2), because some variable (a list in this case) is written many times in the goal, the **same list** is eaten on all arguments (for each generated equality $X_{ai+b} = X_{a'i+b'}$). Then the speed of eating is linear (for each argument). But we can ensure that when the slowest eater has eaten the k first variables of the list, then all the paths, using equations $X_i = X_{g(i)}$ which do not use a number larger than k, have been built.*

The previous theorem and remark are crucial for the following results. The clause associated with a recursively enumerable set can be considered as a process of enumeration of this set. It suffices to init X_1 to $\sharp$ as explained and to apply iteratively this clause. Thus, we will build sets and use some results about them. The remark explains that the "construction" of the set is linear since the construction of the Conway relation $\equiv_g$, associated with Σ, is so.

6. THE HALTING PROBLEM

In this section, we will provide the answer to the first problem: Does the resolution of a binary recursive Horn clause when given a goal halt? While it has been established decidable in the ground [47] and linear [17] goal case, we will establish here the undecidability in the general case [19]. Already a right-linear rule is

sufficient. In order to complete the answer, we will show the decidability if the rule is left-linear.

6.1. The General Case

Theorem 6.1. The halting problem, according to SLD resolution, of a right-linear binary recursive Horn clause is undecidable. The resolution can be applied with or without occur-check.

PROOF. It is a direct consequence of Theorem 5.1 using a similar principle as in the example with the Collatz problem encoding in the previous section.

By initializing L in the goal to $[\sharp, X_2, \ldots, X_{2^n-1}, \flat \mid LL]$ where the mark $\flat$ is put on the (2^n)th position of L, then the resolution will stop if and only if equation $X_1 = X_{2^n}$, that is, $\sharp = \flat$, is generated during SLD resolution, that is, if and only if n is an element of Σ. Since it is undecidable for a given integer n and a recursively enumerable Σ, whether or not n belongs to Σ, the result is proved. It is easy to verify that the occur-check does not play any role in the proof.   □

Let us observe that the constructed clause depends only on Σ; it is only the goal that depends on n. So, if we fix any nonrecursive Σ, we get:

Theorem 6.2. There exists an explicitly constructable right-linear binary Horn clause for which the halting problem, according to SLD resolution, is undecidable. The resolution can be applied with or without occur-check.

6.2. Consequences

Some corollaries can be immediately established. In each case, it is possible to give a general version and an "explicitly constructable" version. We will give only the second one (since it includes the first one).

6.2.1. Finite Number of Solutions

Corollary 6.1. There is a particular program in the following form:

$$\begin{cases} p(\text{fact}) \leftarrow . \\ p(\text{left}) \leftarrow p(\text{right}). \end{cases}$$

where fact, left, right are terms such that it is undecidable whether or not, for a given goal, " $\leftarrow p(\text{goal})$.," there exists a finite number of answer-substitutions.

PROOF. If we consider the program built with

* the binary Horn clause and the goal defined in the previous proof,
* a fact "$p(X) \leftarrow .$" where X is a variable

each time that the fact is considered, we obtain a solution. This program will have a finite number of solutions if and only if the binary Horn clause stops for the given goal. This has been proven to be undecidable in Theorem 6.2.   □

6.2.2. Occur-Check. The previous results have been established over and above the occur-check. From this, we can assert that it is undecidable whether or not, when given a program, this occur-check must be applied during the resolution.

Corollary 6.2. There exists an explicitly constructable right-linear binary Horn clause for which it is undecidable whether or not, when given a goal, the occur-check will be necessary during the resolution.

PROOF. In the proof of Theorem 6.2, we replace the equalities

$$X_1 = \text{\textcrh} \text{ and } X_{2^n} = \flat$$

of the goal by

$$X_1 = h(Y, s(Y)) \text{ and } X_{2^n} = h(Z, Z)$$

where h and s are function symbol of arity 2 and 1, respectively. It is undecidable whether or not the program will stop because of the equalities $Z = Y$ and $Z = s(Y)$, that is, because of the occur-check. □

6.2.3. "Total Decoration" Here follows the last result which is a consequence of the undecidability of the halting problem. It concerns the property of *"total decoration"* [5] in the resolution of logic program. This property is used to optimize the step from logic programming to attribute grammars. The SLD resolution of a program is said to be totally decorated if and only if, at each step of the resolution, all Horn clauses of the program are applicable. The decidability of this problem was stated as open in [5]. In our case, there is only one clause. The property is then equivalent to the halting problem, or better, to the nonhalting problem, of this clause.

Corollary 6.3. There exists an explicitly constructable right-linear binary Horn clause for which it is undecidable whether or not, when given a goal, its resolution will be totally decorated.

PROOF. See the paragraph above. □

6.3. The Left-Linear Clause Case

At this point, we know that the halting problem is decidable if the goal is linear and undecidable if the rule is right-linear. We now consider the case where the rule is left-linear. The proof of the following result uses a completely different method from the previous one. It is based on the weighted graphs. We refer the reader to [16, 17, 32, 22].

Theorem 6.3. The halting problem of a left-linear binary Horn clause, when given a goal, is decidable.

6.4. Conclusion

We conclude that for a program of the following pattern:

$$\begin{cases} p(\text{left}) \leftarrow p(\text{right}). \\ \leftarrow p(\text{goal}). \end{cases}$$

the halting problem is decidable as soon as one of the terms *left* or *goal* is linear. These results are summarized in the following table:

goal	left	right	termination
ground	any	any	decidable [47]
linear	any	any	decidable [11]
any	linear	any	decidable
any	any	linear	undecidable

7. THE EMPTINESS PROBLEM

The second problem concerns the existence of at least one solution, also called the *emptiness problem*. Although it has been shown to be decidable in the ground [47] and linear [17] case, we will show in this section that it is undecidable in the general case. Two proofs will be presented. The first one is based on the Conway function encoding [20]. The second is based on the Post problem [28]. They have been established simultaneously and independently. The second proof is more elegant, but the first also works for a syntactical defined class of programs that is of some interest to us.

7.1. Preliminary Remark

Concerning the *existence* of solutions, the programs

$$\begin{cases} p(\textit{fact}) \leftarrow . \\ p(\textit{left}) \leftarrow p(\textit{right}). \\ \leftarrow p(\textit{goal}). \end{cases} \quad \text{and} \quad \begin{cases} p(\textit{goal}) \leftarrow . \\ p(\textit{right}) \leftarrow p(\textit{left}). \\ \leftarrow p(\textit{fact}). \end{cases}$$

are equivalent.

In the first case, the system of equations

$$\{\textit{goal} = \textit{left}_1, \textit{left}_{i+1} = \textit{right}_i (1 \leq i \leq n - 1), \textit{right}_n = \textit{fact}\}$$

must be solved; in the second,

$$\{\textit{goal} = \textit{left}_n, \textit{left}_i = \textit{right}_{i+1} (1 \leq i \leq n - 1), \textit{right}_1 = \textit{fact}\}.$$

According to a renaming of the variables ($\forall 1 \leq j \leq n$, X_j renamed in X_{n-j+1}), the two systems are equivalent. The existence of solutions for one provides the existence of solutions for the other. Then, when a result is established for some properties of the 4-tuple (*fact*, *left*, *right*, *goal*), the same result with the 4-tuple (*goal*, *right*, *left*, *fact*) can be deduced.

7.2. The Proof via Conway

This is probably the most complex proof in this paper. It requires a good understanding of how the encoding of the Conway functions into Horn clauses works. The principle is as follows: assume we have a linear Conway function (or its associated relation). In the associated program, the propagation of the mark ♯ in

Π_1 builds the list :

$\mathcal{L}_1 = [\natural, \natural, \flat, \natural, \flat, \cdots \natural, \cdots, \flat, \cdots]$ the $(n = 2^p)^{th}$ element is instantiated to $\natural$
$\qquad\qquad\uparrow\qquad\uparrow$ in less than 2^p iterations.
$\qquad\quad n = 2^p n \neq 2^p$

Π_2 builds the list :

$\mathcal{L}_2 = [\natural, -, \cdots \natural, \cdots]$ the $(n = 2^p)^{th}$ element is instantiated to
$\qquad\quad\uparrow$ $\natural$ in less than 2^p iterations if and only if
$\quad n = 2^p$ and $p \in \Sigma$ $p \in \Sigma$.

In Π, the fact impose at the nth iteration to unify the nth element of $\mathcal{L}_1$ with $\natural$ and the nth of $\mathcal{L}_2$ with $\flat$. At this step, three kinds of unification may occur

$$\boxed{n}$$
$$\natural$$

$n \neq 2^p \qquad \Rightarrow \mathcal{L}_1 = [\natural, \natural, \flat, \natural, \cdots, \flat, \cdots] \longrightarrow$ fail
$\qquad\qquad\qquad \mathcal{L}_2 = [\flat, X_2, \cdots, ? , \cdots]$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad \flat$

$\qquad\qquad\qquad\qquad\qquad\qquad\quad \natural$
$n = 2^p$ and $p \in \Sigma \Rightarrow \quad \mathcal{L}_1 = [\natural, \natural, \flat, \natural, \cdots, \natural, \cdots] \longrightarrow$ fail
$\qquad\qquad\qquad\qquad\quad \mathcal{L}_2 = [\flat, X_2, \cdots, \natural, \cdots]$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \flat$

$\qquad\qquad\qquad\qquad\qquad\qquad \natural$
$n = 2^p$ and $p \notin \Sigma \Rightarrow \mathcal{L}_1 = [\natural, \natural, \flat, \natural, \cdots, \natural, \cdots] \longrightarrow$ success
$\qquad\qquad\qquad\qquad\quad \mathcal{L}_2 = [\flat, X_2, \cdots, X_{2^p}, \cdots]$
$\qquad\qquad\qquad\qquad\qquad\qquad \flat$
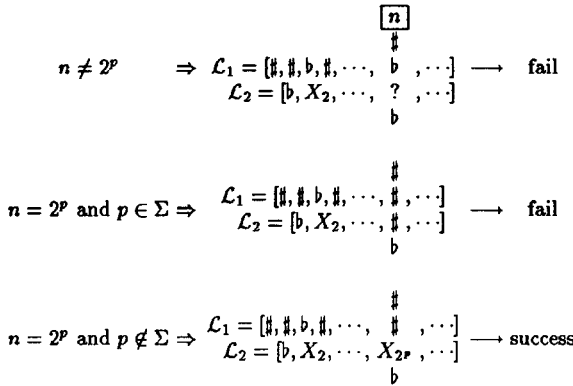
We will have a solution if and only if :

$$\exists p, \ n = 2^p \ \text{and} \ p \notin \Sigma$$

and no solution iff Σ is total, which is undecidable (see Corollary 4.1)

FIGURE 1. Theorem 7.1: principle of proof.

the list of the goal will be linear too. It is possible to write a program for which a solution at the (2^n)th step is equivalent to the fact that n does not belong to the linear recursive set. Then we can deduce from Corollary 4.1 the undecidability of the emptiness problem.

Let us examine the detailed proof. A less formal presentation is given in Figure 1.

Theorem 7.1. For a program of the form

$$\begin{cases} p(\, fact\,) \leftarrow . \\ p(\, left\,) \leftarrow p(\, right\,). \\ \leftarrow p(\, goal\,). \end{cases}$$

where fact and right are linear, the emptiness problem is undecidable.

For the proof of Theorem 7.1, we need the following lemma.

Lemma 7.1. For every linear recursively enumerable set Σ (containing 0), there exists a right-linear binary clause and a goal such that a natural integer, n, belongs to Σ if

and only if after at most 2^n SLD resolution steps, the first argument of the initial goal becomes instantiated to a list, the (2^n)th element of which is $\sharp$.

PROOF. Let Σ be a linear recursively enumerable set. By Definition 4.3 and its natural counterpart, there exists a linear and null Conway function whose domain is Σ. According to Theorem 5.1, there exists a clause $\mathscr{C}$ and a goal $\mathscr{G}$ that can be associated to Σ. As stated in Remark 5.2, the associated Conway relation is linearly computed by $(\mathscr{C})$ and $(\mathscr{G})$, with a linear coefficient equal to some α. In other words, mark $\sharp$ is linearly propagated in the first list-argument of the goal. It is now easy to define a linear clause $(\mathscr{C}')$ and a goal $(\mathscr{G}')$, with linear coefficient equal to 1, such that each resolution step of $\mathscr{C}'$ corresponds to α steps of $\mathscr{C}$. $\square$

PROOF OF THEOREM 7.1.   Consider the following program:

$$\begin{cases} p([X \mid L],[\_, X \mid LL]) \leftarrow p(L, LL). \\ \leftarrow p\left(\left[\sharp \mid L\right],\left[\sharp \mid L\right]\right). \end{cases}$$

It puts a mark $\sharp$ to all the (2^n)th positions of the list $[\sharp \mid L]$ of the goal. It is possible to modify it slightly such that it marks the other positions with another symbol $\flat$:

$$\Pi_1 \quad \begin{cases} p\left([X \mid L],[Y, X \mid LL],\left[\flat, Z \mid LLL\right]\right) \leftarrow p(L, LL, LLL). \\ \leftarrow p\left(\left[\sharp, \sharp \mid L\right],\left[\sharp, \sharp \mid L\right], L\right). \end{cases}$$

The clause generates the equality $X_i = X_{2i}$ for all $i \leq 1$; the goal produces $X_1 = \sharp$ and $X_{2i+1} = \flat$ for all $i \leq 1$ because of the L found three times. Hence, after n resolution steps, the first argument of the goal, L, becomes instantiated to

$$\mathscr{L} = [X_1, X_2, \ldots, X_n, \ldots]$$

where $\forall k < n$, $X_k = \sharp$ if k is a power of 2 and $X_k = \flat$ otherwise.

We now define a class of programs for which the existence of solutions is undecidable. Let Σ be any linear recursively enumerable set; the associated clause and goal defined in the previous lemma are denoted as follows:

$$\Pi_2 \quad \begin{cases} p(l_1, l_2, \ldots, l_k) \leftarrow p(r_1, r_2, \ldots, r_k). \\ \leftarrow p(g_1, g_2, \ldots, g_k). \end{cases}$$

Now, by merging Π_1 and Π_2, our particular class of programs follows:

$$\Pi \quad \begin{cases} p\left(Y_1, Y_2, \ldots, Y_k, \flat, Z, \left[\sharp \mid L\right], LL, LLL\right) \leftarrow . \\ p\left(l_1, l_2, \ldots, l_k, W, [U \mid V], [X \mid L], [Y, X \mid LL], \left[\flat, Z \mid LLL\right]\right) \\ \qquad\qquad\qquad\qquad \leftarrow p(r_1, r_2, \ldots, r_k, U, V, L, LL, LLL) \\ \leftarrow p\left(g_1, g_2, \ldots, g_k, X, g_1, \left[\sharp, \sharp \mid L\right], \left[\sharp, \sharp \mid L\right], L\right). \end{cases}$$

The k first arguments codify Σ. The $(k + 1)$th argument allows to extract, at the nth iteration, the nth argument of the list which characterizes Σ. The $(k + 2)$th argument is the list itself (because of the unification with g_1 in the goal) with its n first elements deleted at the nth iteration. Finally, the last three arguments codify the list of powers of 2. Because there is a solution at the nth step if and only if:

- n is a power of 2 (because of the three last arguments and the unification with $[\# \mid L]$ in the goal)

- the nth element of the list which characterizes Σ is not marked by $\#$ because it must be unifiable with b (the $(k + 1)$th argument of the fact and the variable U of the rule).

In other words, since we know that the marking (by $\#$) is linear (with $\alpha = 1$), there is a solution at the (2^n)th step if and only if the (2^n)th element of the list associated to Σ is not marked by $\#$, that is, if and only if n does not belong to Σ. Therefore, Π has no solution if and only if Σ is equal to $\mathbb{N}$. According to Corollary 4.1, this is undecidable. This proves the result. $\square$

By symmetry of the problem, according to Section 7.1, we immediately deduce an equivalent theorem with the terms *goal* and linear and any *right* and *fact* terms.

7.3. An Important Corollary

An important consequence of this theorem is that it allows to solve one open problem in first-order logic concerning the satisfiability of formulas. Indeed, for the first-order formulas with four subformulas such as

$$\forall X_i, \left(P(t_1) \wedge (Q(t_2) \vee R(t_3)) \wedge S(t_4) \right)$$

where P, Q, R, and S are literals and the X_i are the variables occurring in t_1, t_2, t_3, and t_4, a particular instance of this problem is

$$\forall X_i, \left(P(t_1) \wedge (P(t_2) \vee \neg P(t_3)) \wedge \neg P(t_4) \right)$$

for which the nonsatisfiability problem is equivalent to the existence of solutions for the program

$$\begin{cases} p(t_1) \leftarrow . \\ p(t_2) \leftarrow p(t_3). \\ \leftarrow p(t_4). \end{cases}$$

where t_1, t_2, t_3, and t_4 are any terms. Then we can assert:

Corollary 7.1. The satisfiability of the class of first-order formulas with four subformulas is undecidable.

PROOF. From Theorem 7.1 $\square$

This result can be connected to an old open problem: the satisfiability of formulas in pure quantification theory (that is, without function symbol and with an eventually infinite number of constants):

$$\forall t \; \exists u \; \forall v \; \cdots \; \forall w (\mathscr{A}_1 \wedge \mathscr{A}_2 \vee \cdots \wedge \mathscr{A}_n)$$

where the $\mathscr{A}_i$ are atomic positive or negative formulas; the satisfiability of the 5-subformula case has been shown to be undecidable in [23]. It is established that this problem is equivalent to the halting problem of 2-counter machines (which is undecidable). The 3- and 4-subformula problems remain open.

7.4. The Proof via Post

This proof [28] is not based on the Conway functions, but on the better known Post problem [42].

7.4.1. The Post Problem. Let us consider a finite alphabet Σ. A *Post correspondence system* over Σ is a nonempty finite set $\mathscr{S} = \{(l_i, r_i) \mid i \in [1, \dots, m]\}$ where the l_i, r_i are words over Σ. A nonempty sequence of indices $1 \le i_1, \dots, i_n \le m$ is called a solution of $\mathscr{S}$ if and only if

$$l_{i_1} \cdots l_{i_m} = r_{i_1} \cdots r_{i_m}.$$

It is well known that the Post correspondence problem, that is, "Does there exist a solution for a given system ?," is, in general, undecidable if the alphabet contains at least two symbols.

7.4.2. Encoding of the Post Problem. Elements a_i of the alphabet Σ will be represented as unary function symbols, and a word $w = a_1 \cdots a_n$ over Σ thus becomes a term $a_1(a_2(\dots(a_n(\epsilon))\dots))$ where ϵ is a constant corresponding to the empty word. So, the composition of words is associative since the composition of functions is associative. For convenience, we also write $w(\epsilon)$ instead of $(a_1(a_2(\dots(a_n(\epsilon))\dots)))$ and $u(v(\epsilon)) = uv(\epsilon)$ where u and v correspond to words over Σ. For instance, if $w_1 = ab$, $w_2 = ba$, $v_1 = a$, and $v_2 = bba$, then $w_1(w_2(t)) = a(b(b(a(t)))) = abba(t) = v_1 v_2(t)$ for any term t.

To append something to a list using unification, we use the concept of *difference lists*. To explain the encoding of a Post correspondence problem, we adopt SLD resolution as an operational semantics for the logic problem. The search space of possible sequences of indices inherent to a Post correspondence problem is not encoded in the and/or tree of the logic program. Instead, we encode it in two (difference) lists L and R. At the beginning of the computation, L and R are $[l_1(\epsilon), \dots, l_m(\epsilon) \mid X] - X$ and $[r_1(\epsilon), \dots, r_m(\epsilon) \mid Y] - Y$, respectively. This encodes all possible sequences of indices of length 1 (i.e., $1, 2, \dots, m$). In the next step, we select the sequence 1 and replace it by all sequences that have length 2 and as suffix 1. In terms of the lists L and R, we remove $l_1(\epsilon)$ and $r_1(\epsilon)$ (representing the sequence 1 of length 1) and append $[l_1(l_1), \dots, l_m(l_1) \mid X]$ and $[r_1(r_1), \dots, r_m(r_1) \mid Y]$, respectively (representing the sequences $11, 12, \dots, 1m$ of length 2).

In the general case, we select in each step a sequence $i_1 \cdots i_j$ of indices and replace it by all sequences that have length $j + 1$ and $i_1 \cdots i_j$ as suffix. Always selecting the heads of L and R and appending the extensions is a fair strategy, i.e., it ensures that successively all possible sequences appear as heads of the two difference lists.

Given a Post correspondence problem as above, the following binary program has an SLD refutation iff the Post correspondence problem has a solution.

$$
\left\{
\begin{array}{l}
P([E \mid H_1] - H_2, [E \mid H_3] - H_4) \leftarrow . \\
P([C \mid L] - [l_1(C),\ldots,l_m(C) \mid X], \\
\quad [D \mid R] - [r_1(D),\ldots,r_m(D) \mid Y] \leftarrow P(L - X, R - Y). \\
\leftarrow P([l_1(\epsilon),\ldots,l_m(\epsilon) \mid X] - X, \\
\quad [r_1(\epsilon),\ldots,r_m(\epsilon) \mid Y] - Y).
\end{array}
\right.
$$

The fact checks whether the heads of the lists in the current goal are equal, i.e., encode a solution of the Post correspondence problem. In Figure 2, the sequence of goals is depicted which is induced by SLD resolution, with a search rule always taking the binary rule for the next SLD resolution step.

Since the Post correspondence problem is undecidable, the existence of solutions for this program is undecidable too. □

$$
\begin{aligned}
&\leftarrow P([l_1(\epsilon),\cdots,l_m(\epsilon) \\
&\qquad\qquad |X_0] - X_0, \quad [r_1(\epsilon),\cdots,r_m(\epsilon)|Y_0] - Y_0) \\
&\leftarrow P([l_2(\epsilon),\cdots,l_m(\epsilon) \\
&\qquad ,l_1 l_1(\epsilon),\cdots,l_m l_1(\epsilon) \\
&\qquad\qquad |X_1] - X_1, \quad [r_2(\epsilon),\cdots,r_m(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_1(\epsilon),\cdots,r_m r_1(\epsilon)|Y_1] - Y_1) \\
&\leftarrow P([l_3(\epsilon),\cdots,l_m(\epsilon) \\
&\qquad ,l_1 l_1(\epsilon),\cdots,l_m l_1(\epsilon) \\
&\qquad ,l_1 l_2(\epsilon),\cdots,l_m l_2(\epsilon) \\
&\qquad\qquad |X_2] - X_2, \quad [r_3(\epsilon),\cdots,r_m(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_1(\epsilon),\cdots,r_m r_1(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_2(\epsilon),\cdots,r_m r_2(\epsilon)|Y_2] - Y_2) \\
&\cdots \\
&\leftarrow P([l_1 l_1(\epsilon),\cdots,l_m l_1(\epsilon) \\
&\qquad ,l_1 l_2(\epsilon),\cdots,l_m l_2(\epsilon) \\
&\qquad ,\cdots \\
&\qquad ,l_1 l_m(\epsilon),\cdots,l_m l_m(\epsilon) \\
&\qquad\qquad |X_m] - X_m, \quad [r_1 r_1(\epsilon),\cdots,r_m r_1(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_2(\epsilon),\cdots,r_m r_2(\epsilon) \\
&\qquad\qquad\qquad ,\cdots \\
&\qquad\qquad\qquad ,r_1 r_m(\epsilon),\cdots,r_m l_m(\epsilon)|Y_m] - Y_m) \\
&\leftarrow P([l_2 l_1(\epsilon),\cdots,l_m l_1(\epsilon) \\
&\qquad ,l_1 l_2(\epsilon),\cdots,l_m l_2(\epsilon) \\
&\qquad ,\cdots \\
&\qquad ,l_1 l_m(\epsilon),\cdots,l_m l_m(\epsilon) \\
&\qquad ,l_1 l_1 l_1(\epsilon),\cdots,l_m l_1 l_1(\epsilon) \\
&\qquad\qquad |X_{m+1}] - X_{m+1}, \quad [r_2 r_1(\epsilon),\cdots,r_m r_1(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_2(\epsilon),\cdots,r_m r_2(\epsilon) \\
&\qquad\qquad\qquad ,\cdots \\
&\qquad\qquad\qquad ,r_1 r_m(\epsilon),\cdots,r_m l_m(\epsilon) \\
&\qquad\qquad\qquad ,r_1 r_1 r_1(\epsilon),\cdots,r_m r_1 r_1(\epsilon)|Y_{m+1}] - Y_{m+1}) \\
&\cdots
\end{aligned}
$$

FIGURE 2. A goal sequence induced by the logic program.

7.5. Some Particular Cases

First, we will establish the decidability of the emptiness problem in the cases where three characteristic elements of our programs are linear, and where only one is ground. Then we will prove that the result remains undecidable if the clause is (left and right) linear.

7.5.1. Decidability. The proofs of the following theorems will appear in [22].

Theorem 7.2. *The emptiness problem for the class of programs*

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

where three of the terms left, right, fact, and goal are linear, is decidable.

Theorem 7.3. *The emptiness problem for the class of programs*

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

is decidable as soon as one of the terms left, right, goal, or fact is ground.

7.5.2. Undecidability. Here, we will consider the case where the left *and* right terms of the rule are linear. We shall transform the nonlinear clauses that code the Conway functions into linear to show that the proof of Theorem 7.1 can be applied. Now, we are no longer able to use two occurrences of the same variable in "*left*" to ensure that two elements of the list L are equal. Instead, we built out of the elements of L new lists LLU and LLV such that their corresponding elements are supposed to be equal. For linearity reasons, we cannot force the equality of LLU and LLV during the resolution, so we postpone it, and check the equality while unifying with the fact.

Theorem 7.4. *For the class of programs*

$$\begin{cases} p(\mathit{fact}) \leftarrow . \\ p(\mathit{left}) \leftarrow p(\mathit{right}). \\ \leftarrow p(\mathit{goal}). \end{cases}$$

where left, right are linear and fact, goal are arbitrary, the emptiness problem is undecidable.

PROOF. Let us consider the following program:

$$\begin{cases} p\left([X \mid LX], \left[\overbrace{U, \_, \ldots, \_}^{a} \mid LU\right], \left[\overbrace{V, \_, \ldots, \_}^{c} \mid LV\right]\right) \leftarrow p(LX, LU, LV). \\ \leftarrow p\left(L, \left[\underbrace{\_, \ldots, \_}_{b} \mid L\right], \left[\underbrace{\_, \ldots, \_}_{d} \mid L\right]\right). \end{cases}$$

It produces the equalities

$$\begin{cases} X_{ai+b} = U_i \\ X_{ci+d} = V_i. \end{cases}$$

If we slightly modify this program and add a fact as in

$$\begin{cases} p(\_,\_,\_,L,L) \leftarrow . \\ p\left([X\,|\,LX], \left[\overbrace{U,\_,\dots,\_}^{a}\,|\,LU\right], \left[\overbrace{V,\_,\dots,\_}^{c}\,|\,LV\right], LLU, LLV\right) \\ \qquad\qquad\qquad\qquad \leftarrow p(LX, LU, LV, [U\,|\,LLU], [V\,|\,LLV]). \\ \leftarrow p\left(L, \left[\underbrace{\_,\dots,\_}_{b}\,|\,L\right], \left[\underbrace{\_,\dots,\_}_{d}\,|\,L\right], [\ ], [\ ]\right). \end{cases}$$

the two last arguments become instantiated to the lists $[U_1,\dots,U_n\,|\,\_]$ and $[V_1,\dots,V_n\,|\,\_]$. Then, because of the nonlinearity of the fact, we add the equality $U_i = V_i$, and deduce that we have for X

$$X_{ai+b} = X_{ci+d}.$$

And no other different relation on X is defined.

It is easy to create n other equalities on X. For example, if n is 2,

$$\begin{cases} p(\_,X,\_,\_,L1,L1,\_,\_,L2,L2) \leftarrow . \\ p\Big([X\,|\,LX], Y, \left[\overbrace{U1,\_,\dots,\_}^{a_1}\,|\,LU1\right], \left[\overbrace{V1,\_,\dots,\_}^{c_1}\,|\,LV1\right], LLU1, LLV1, \\ \qquad \left[\underbrace{U2,\_,\dots,\_}_{a_2}\,|\,LU2\right], \left[\underbrace{V2,\_,\dots,\_}_{c_2}\,|\,LV2\right], LLU2, LLV2\Big) \\ \qquad\qquad \leftarrow p(LX, X, LU1, LV1, [U1\,|\,LLU1], [V1\,|\,LLV1], \\ \qquad\qquad\qquad\qquad LU2, LV2, [U2\,|\,LLU2], [V2\,|\,LLV2]) \\ \leftarrow p\Big(\big[\mathbin{\#}|\,L1\big], \_, \left[\overbrace{\_,\dots,\_}^{b_1},\mathbin{\#}\,|\,L1\right], \left[\overbrace{\_,\dots,\_}^{d_1},\mathbin{\#}\,|\,L1\right], [\ ], [\ ], \\ \qquad \left[\underbrace{\_,\dots,\_}_{b_2},\mathbin{\#}\,|\,L2\right], \left[\underbrace{\_,\dots,\_}_{d_2},\mathbin{\#}\,|\,L2\right], [\ ], [\ ]\Big). \end{cases}$$

In the same way as previously, this program produces the equalities

$$X_{a_1 i + b_1} = X_{c_1 i + d_1} \quad \text{and} \quad X_{a_2 i + b_2} = X_{c_2 i + d_2}.$$

Let us note that in the fact, after p iterations, X is X_p. The extension for any $n \geq 2$ is now obvious; then we can code any Conway function, that is, any linear recursive set Σ (containing $\{0\}$). Indeed, if $p = 2^k$, then in the fact, $X = X_p = X_{2^k} = \mathbin{\#}$ if and only if $k \in \Sigma$. Let us call Π_Σ such a program associated with Σ.

Now, we consider another program Π':

$$\begin{cases} p(\_,\_,L,L). \\ p\Big([X\,|\,LX],\big[\_,Y,^\flat\,|\,LLY\big],LLX,LLY\Big) \\ \quad \leftarrow p\Big(LX,\big[^\flat,LY\big],[X\,|\,LLX],[Y\,|\,LLY]\Big). \\ \quad \leftarrow p\Big(\big[\sharp,\sharp\,|\,L\big],\big[\sharp,\sharp\,|\,L\big],[\ ],[\ ]\Big). \end{cases}$$

It is the linear equivalent to the program Π in proof of Theorem 7.1. It builds the list

$$L = [X\,|\,LX] = [X_1, X_2, \ldots, X_p, \ldots]$$

with the relations $X_p = \sharp$ if p is a power of two and $X_p = ^\flat$ otherwise.

As in the general case (see proof of Theorem 7.1), by merging the Horn clauses and goals of the above program and of some Π_Σ, and by choosing a fact such as

$$p\Big(\underbrace{\_,^\flat,\_,\_,L1,L1,\ldots,\_,\_,LN,LN}_{\Pi_\Sigma\ part},\underbrace{\_,\_,\big[\sharp\,|\,L\big],\big[\sharp\,|\,L\big]}_{\Pi'\ part}\Big) \leftarrow .$$

we obtain a program with one linear binary rule which will have at least one solution if and only if Σ is not equal to $\mathbb{N}$. This property is undecidable. □

7.6. Conclusion

We have established the undecidability of the emptiness problem in the general case. Moreover, we have proven that the emptiness problem is decidable as soon as three characteristic terms are linear or as soon as only one is ground.

goal	left	right	fact	Emptiness
ground	any	any	ground	decidable [47]
linear	any	any	linear	decidable [11]
ground	any	any	any	decidable
any	any	any	ground	
linear	linear	linear	any	decidable
any	linear	linear	linear	
any	any	linear	linear	undecidable
linear	linear	any	any	
any	linear	linear	any	undecidable

8. COMPUTATIONAL POWER

In this section, we will prove the main result. We prove that *append*-like programs have the same computational power as Turing machines [21].

We will use the principles of the two previously presented proofs of the existence of solutions. Roughly speaking, the following proof consists of building a

Prolog meta-interpreter with only one binary Horn clause. First, we will build a word generator, then a pseudo-meta-interpreter which never stops, and last we will add the termination. The only difficult point is the technical one which uses the Conway functions in order to guarantee the termination of our meta-interpreter.

8.1. A Word Generator

In this section, we show how to build a one binary Horn clause program which generates all words over the alphabet $\{a, b\}$. These words are represented by a list (e.g., $[a, b, b, a, b]$ stands for *abbab*). The encoding is similar to these previously presented to encode the Post Correspondence Problem:

$$\begin{cases} gen([Word \mid R] - RR, Word) \leftarrow \\ gen([Word \mid R] - [[a \mid Word], [b \mid Word] \mid RR], AWord) \\ \quad \leftarrow gen(R - RR, AWord). \\ \leftarrow gen([[\,] \mid R] - R, Word). \end{cases}$$

The behavior of the difference list is the following:

$$\left[Word, \overbrace{\underbrace{\_, \ldots, \_, [a \mid Word], [b \mid Word]}_{R - RR}, \underbrace{\_, '', \ldots}_{RR}} \right].$$

Consider the first steps of this program. By unifying the goal and the fact, we obtain the solution $Word = [\ ]$. Using the binary clause once results in the new goal

$$gen([[a], [b] \mid RR_1] - RR_1, AWord_1)$$

producing by unification with the fact the solution $Word = [a]$. Resolving this new goal with the binary clause instead of the fact results in the goal

$$gen([[b], [a, a], [b, a] \mid RR_2] - RR_2, AWord_2)$$

resulting in the solution $Word = [b]$, etc. Observe that a and b serve as prefixes of two new words such that the suffix of these words is the first element of the list. These two words are concatenated to the tail of the list generated so far. In other words, the difference-list can be seen as a FIFO (First In First Out) pipe.

The principle is strictly similar to the one of the Post problem encoding. It is clear that this generator can be easily extended for any finite alphabet.

8.2. A First Meta-Interpreter

Let us begin with the study of a meta-program. It is made of one fact, two binary recursive rules, and one goal [41]. An equivalent form, with two facts, one ternary rule, and one goal can be established.

Let Π be the set of Horn clauses

$$\Pi = \{clause_1, clause_2, \ldots, clause_n\}$$

and $\leftarrow g_1, \ldots, g_k$ a goal. The following meta-program generates the same answer-substitutions in the same order as a standard breadth-first SLD interpreter:

$$\begin{cases} solve([\ ],[\ ]) \leftarrow . \\ solve([Goal \mid RestOfGoals], [[Goal \mid Body] - RestOfGoals \mid L]) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \leftarrow solve(Body, \mathscr{P}). \\ solve(Goals, [Clause \mid Rest]) \leftarrow solve(Goals, Rest). \\ \leftarrow solve(\mathscr{G}, \mathscr{P}). \end{cases}$$

$\mathscr{G}$ denotes the list $[g_1, g_2, \ldots, g_n]$ and $\mathscr{P}$ the list of difference lists which encode the clauses of the program Π, $\mathscr{P} = [clause_1, \ldots, clause_n]$. A clause $a \leftarrow b_1, \ldots, b_m$ of Π is encoded by the difference list $[a, b_1, \ldots, b_m \mid R] - R$. In this meta-program, the first binary clause is used to choose the first clause in the current clause list and check if its head part can be unified with the current goal. The second clause discards the first clause in the current clause list. It is easy to check that SLD resolution is achieved.[7]

This meta-program is studied in detail in [40]. The proof of its equivalence with the object program Π with respect to standard SLD resolution (through a depth-first, left to right, traversal of the SLD tree) is presented in [22]. Its complexity—as defined below—is shown to be linearly dependent on that of the original program Π.

Let us call the *complexity* of a solution-node sol_n the number of crossed nodes of the SLD tree (through a depth-first, left to right, traversal) before reaching sol_n. We will denote by φ_p the complexity function of program p, and N_p its number of rules. p will take the values o for the object program, and m for the meta-program.

At best, the goal unifies with the first rule of the original program, and this rule is a fact. At worst, it unifies with the last one. We obtain

$$\varphi_o(sol_n) + N_o \le \varphi_m(sol_n) \le (\varphi_o(sol_n) \times N_o) + N_o.$$

Example 8.1. The code of the meta-program associated to the "*append*" program is as follows:

```
1. solve([ ], [ ]).
2. solve([Goal|L1], [[Goal|L3] – L1|List_of_Rules]) : –
            solve(L3, [[append([],Lapp1,Lapp1)|L] – L,
                [append([X|Lapp2],Lapp3,[X|Lapp4]),
                append(Lapp2,Lapp3,Lapp4)|LL] – LL]).
3. solve(List_of_Goals,[Rule|List_of_Rules]) : –
            solve(List_of_Goals,List_of_Rules).
```

[7]Note that the fact cannot be simplified into $solve([\ ], -)$ because, in this case, any goal $\leftarrow$ $solve([\ ], Prog)$ could unify with either the fact or the second binary clause. So each solution in $\mathscr{P}$ would produce n solutions for $\leftarrow solve(\mathscr{G}, \mathscr{P})$.

For the initial goal append([1],[2,3],L), the corresponding goal in the meta-program will be

```
: – solve([append([1],[2,3],L1)],
        [[append([],Lapp1,Lapp1)|L] – L,
        [append([X|Lapp2],Lapp3,[X|Lapp4]),
        append(Lapp2,Lapp3,Lapp4)|LL] – LL]).   □
```

For the encoding of an arbitrary program Π, both the first binary clause and the goal have to be adapted in an appropriate way.

Now, assume that Π is a meta-interpreter. Then this encoding allows to define the explicitly constructible meta-interpreter *MI* with the right pattern. In order to interpret any program with the help of Π, we just have to encode the appropriate goal for Π, and therefore for *MI*.

To summarize, we have built a meta-interpreter for Horn clause languages with one fact, two binary recursive clauses, and one goal.

8.3. An SLD Tree Generator

Assume a program Π consists of the two binary clauses "$left_1 \leftarrow right_1$" and "$left_2 \leftarrow right_2$," one goal "$\leftarrow goal$," and one fact "$fact \leftarrow$." Consider the word-generator where $\mathscr{A} = right_1, left_1$ and $\mathscr{B} = right_2, left_2$.

$$
\begin{cases}
meta([W \mid R] - RR, W) \leftarrow . \\
meta([W \mid R] - [[\mathscr{A} \mid W], [\mathscr{B} \mid W] \mid RR], [H \mid RRR]) \\
\quad \leftarrow meta(R - RR, [H, X, X \mid RRR]). \\
\leftarrow meta([[goal \mid L] \mid R] - R, [fact \mid LL]).
\end{cases}
$$

After n times using the binary clause for resolving, we obtain as the second argument of the new goal the list

$$
[fact, X_1, X_1, X_2, X_2, \ldots, X_n, X_n \mid T].
$$

Furthermore, according to what we have seen concerning the word generator, we obtain, after some iterations, as the head of the first argument

$$
[right_{i_m}, left_{i_m}, \ldots, right_{i_1}, left_{i_1}, goal \mid T].
$$

The i_j are either 1 or 2, and the variables are renamed before each resolution step. By resolving a current goal with the fact of the above program, we obtain the unification problem

$$
\begin{array}{cccccccc}
[right_{i_m}, & left_{i_m}, & right_{i_{m-1}}, & \ldots, & right_{i_1}, & left_{i_1}, & goal, & \_] \\
\updownarrow & \updownarrow & \updownarrow & \cdots & \updownarrow & \updownarrow & \updownarrow & \updownarrow \\
[fact, & X_1, & X_1, & \ldots, & X_{m-1}, & X_m, & X_m, & \ldots, X_n, Y \mid \_]
\end{array}
$$

which has a solution iff the following system has one.

$$\begin{cases} fact = right_{i_m} \\ left_{i_k} = right_{i_{k-1}}, \qquad 2 \le k \le m \\ goal = left_{i_1}. \end{cases}$$

It is important to note that, by construction, it is assured that the list containing the fact has a sufficient length to obtain these equations. All the possible lists

$$\left[right_{i_m}, left_{i_m}, \ldots, right_{i_1}, left_{i_1}, goal \mid Q' \right]$$

are selected; therefore, SLD resolution is complete. This system will be solvable if and only if there exists a corresponding refutation of the original program Π using the resolution order imposed by the equations (i.e., by the i_k).

Indeed, this program works as an SLD tree generator, and achieves the resolution relative to this tree with a breadth-first strategy as described in Figure 3.

Each time a node is selected, the two new nodes corresponding to the inferences with the rules $\mathscr{A}$ and $\mathscr{B}$ are added at the end of the "to be examined nodes" list (that is, at the end of the "$R - RR$ list"). Then the following node is considered, etc.

8.4. A Binary Nonstopping Meta-Interpreter

In this section, the results of the two previous sections are combined.

Our meta-interpreter MI of Section 8.2 satisfies the pattern of the program Π of the previous section. Hence, it is possible to encode it as above. Therefore, one can associate with any logic program an equivalent program (i.e., with the same solutions) containing a binary clause, one fact and one goal. Unfortunately, this encoding does not preserve termination. Thus, we have a "never stopping" meta-interpreter, say M_{nS}.

In a next section we will show how to construct from this non-terminating interpreter a terminating one. This requires a technical preliminary.

FIGURE 3.
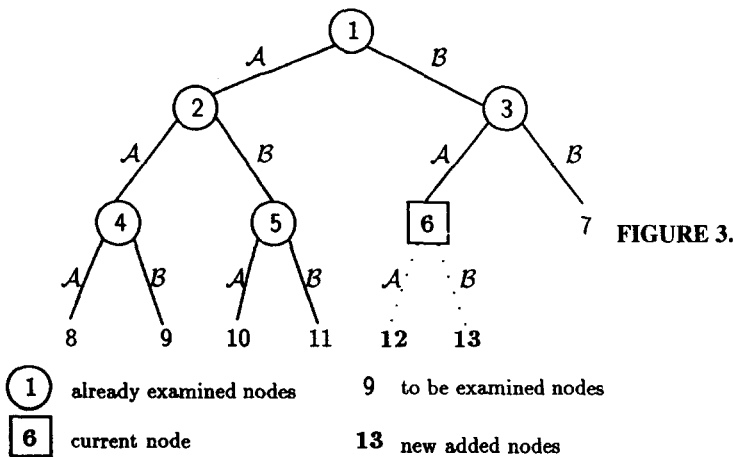
8.5. The Technical Preliminary

Our aim is to cause the termination of M_{nS}. As in the proof of the halting problem, the termination will be caused by a failing unification. But here, we need that the halting happens after a certain number of resolution steps since we want the program to produce the answer-substitutions first.

Remark 8.1. In the following, the term "program" corresponds to the intuitive meaning. The reader can consider it also to denote "a machine (in the sense of a Turing machine) which computes a partial recursive function."

Proposition 8.1. For every program Π with input I, there exist a binary Horn clause $\mathscr{R}_\Pi$ and a goal, depending on I, such that $\mathscr{R}_\Pi$ stops after at least n iterative applications if Π stops after n elementary steps with input I, and does not stop otherwise.

PROOF. Let Π be given; let g be the Conway function associated with Π (or with the Minsky machine related to Π). g is characterized by its "period" p (in fact, the period of $g(n)/n$) and the rational numbers $a_0, a_1, \ldots, a_{p-1}$. Let $\mathscr{R}$ be the associated binary Horn clause as in Proposition 5.1. By construction, at each iteration, $\mathscr{R}$ build p new equalities $X_i = X_{g(i)}$. In fact, at the ith iteration, the program builds the equalities

$$\forall 0 \le k \le d - 1, \qquad X_{p(i-1)+k+1} = X_{a_k \times (p(i-1)+k+1)}.$$

According to Section 4.2.3, since g is a null Conway function, there is only one path from 2^n to 2^0, if any. The Horn clause produces as well positive and negative iterates. Then at each iteration, $\mathscr{R}$ creates p_i positive and n_i negative equalities of the series

$$\left(X_{g^{(i)}(2^n)} = X_{g^{(i+1)}(2^n)} \right)_{i \in \mathbb{N}}$$

with $p_i + n_i \le d$. This is depicted as follows:

$$2^n \underline{\quad} p_1 \xrightarrow{g^*} \cdot \underline{\quad} \cdots \underline{\quad} \cdot \xrightarrow{g^{(-1)*}} n_1 \underline{\quad} 2^0.$$

Consequently, if it takes k_n iterations from $g(2^n)$ to 1, the equality $X_{2^n} = X_1$ will be generated in at least k_n/p iterative applications of the Horn clause. By adding some extra variables, it is possible to slow down $\mathscr{R}$ p times such that the speed (the complexity) of $\mathscr{R}$ is at best the same as the one of g. The resulting clause is called $\mathscr{R}_\Pi$.

Now, as in the proof of the undecidability of the halting problem, for any value I, we can choose a goal for $\mathscr{R}_\Pi$ such that it stops if and only if the relation $X_{2^I} = X_{2^0}$ occurs. And we can guarantee that the number of iterations before termination is greater than the number of elementary steps of Π with input I before halting. $\square$

Now, let us build a particular program from M_{nS}. We will apply Proposition 8.1 to program Π, which takes a Horn clause program P as input, and is defined as follows:

1. read P
2. evaluate P by a breadth-first strategy and keep the solutions in $\mathscr{S}_1$
3. compute $M_{nS}(P)$ and keep the solutions in $\mathscr{S}_2$; stop as soon as $\mathscr{S}_2 = \mathscr{S}_1$ and write 0.

It is clear that Π stops if and only if P stops. The stopping time (that is, the number of steps before termination) with input P is greater than the time used by $M_{nS}(P)$ to produce all the solutions of P.

Now, according to the previous proposition, there exists a clause $\mathscr{R}_{\Pi}$. By adding a general fact and a goal depending on the input, we can build the program M_S:

$$M_S \begin{cases} stop(\,fact_S\,) \leftarrow . \\ stop(\,left_S\,) \leftarrow stop(\,right_S\,). \quad (\mathscr{R}_{\Pi}) \\ \leftarrow stop(\,goal_S\,). \end{cases}$$

M_S stops with input a program P (in $goal_S$) (based, in fact, on the Gödel number of P, for example) if and only if P stops. The stopping time of M_S is greater than the one required by $M_{nS}(P)$ to produce all solutions.

8.6. The Meta-Interpreter

Combining M_{nS} and M_S, we can state:

Theorem 8.1. *There exists a meta-interpreter for Horn clauses in the form of a program with only one binary Horn clause, a fact and goal, which, given as input a Horn clause program P, has the same solutions as P and terminates if and only if P terminates.*

PROOF. Let us denote M_{nS} as follows:

$$M_{nS} : \begin{cases} meta(\,fact_{nS}\,) \leftarrow . \\ meta(\,left_{nS}\,) \leftarrow meta(\,right_{nS}\,). \\ \leftarrow meta(\,goal_{nS}\,). \end{cases}$$

We merge M_{nS} and M_S in a new meta-interpreter:

$$M\mathscr{S} : \begin{cases} TheMeta(\,fact_{nS}, fact_S\,) \leftarrow . \\ TheMeta(\,left_{nS}, left_S\,) \leftarrow TheMeta(\,right_{nS}, right_S\,). \\ \leftarrow TheMeta(\,goal_{nS}, goal_S\,) \end{cases}$$

such that, with input a Horn clause program P, it produces all the solutions of P (because of the M_{nS} part), and then will stop if and only if P terminates (because of the M_S part).

Thus, we have a meta-interpreter, with one binary recursive clause, one fact, and one goal, which preserves the solutions (produced in the same order as in a breadth-first strategy) and the termination of any Horn clause program given as input. □

This result can be seen as the equivalent of the Böhm–Jacopini theorem for logic programming.

Corollary 8.1. *The class of programs with only one binary Horn clause and two unit clauses has the same computational power as Turing machines.*

PROOF. Since we have a meta-interpreter for Horn clauses containing only one binary recursive clause, we can assert that this class of programs has the same

computational power as Horn clause programs, and consequently as Turing machines. □

The previous two main results (Theorems 6.2 and 7.1) are, of course, corollaries of this result.

Corollary 8.2. For append-like programs, halting and emptiness problems are undecidable.

We recall briefly the notations of Section 8.2. φ_p denotes the complexity function for program p and N_p its number of rules. p will take the value u for the universal program (the one of Theorem 8.1), m for the meta-interpreter MI of Section 8.2, and o for the meta-interpreter Π encoded in MI. Since the universal meta-interpreter u achieves the resolution of the SLD tree of m as described in Figure 3 with a breadth-first strategy, its complexity φ_u is bounded by

$$\varphi_u(sol_n) \leq 2^{\varphi_m(sol_n)} \leq 2^{(\varphi_o(sol_n) \times N_o) + N_o}.$$

φ_u is the complexity of the universal program for obtaining the solutions. For halting, the bound of the number of crossed nodes is greater since we add the complexity due to the halting technique with the Conway functions. Hence, when the complexity of the first meta-interpreter is only linearly dependent, the complexity of the universal program is at least exponential with respect to the complexity of the original program.

9. DISCUSSION: TERNARY CLAUSES AND IMPLICATION

9.1. Horn Clause Implication

"The solution of the implication $\mathscr{A} \Rightarrow \mathscr{B}$ of two clauses $\mathscr{A}$ and $\mathscr{B}$ is usually interpreted as the formula $(\forall x_1, \ldots, x_n \; \mathscr{A}) \Rightarrow (\forall y_1, \ldots, y_m \; \mathscr{B})$, where $\{x_1, \ldots, x_n\}$ are the variables occurring in $\mathscr{A}$ and $\{y_1, \ldots, y_m\}$ are the variables in $\mathscr{B}$ (where, by hypothesis, the clauses $\mathscr{A}, \mathscr{B}$ are variable disjoint). Clause implication is equivalent to the non-satisfiability problem of a clause set consisting of clause $\mathscr{A}$ and ground unit clauses that are obtained from the negation of the clause $\mathscr{B}$. Hence the undecidability result holds also for the satisfiability problem of such clause sets" [47].
 In particular, in the case of the Horn clauses, let us explain the equivalence between Horn clause implication and the satisfiability problem of logic programs. First, assume that

$$\mathscr{A} = A \vee \neg A_1 \vee \cdots \vee \neg A_n$$

and

$$\mathscr{B} = B \vee \neg B_1 \vee \cdots \vee \neg B_m$$

and we shall denote by $\tilde{x}$ variables occurring in $\mathscr{A}$, and $\tilde{y}$ variables in $\mathscr{B}$.

$$\forall \tilde{x}(A \vee \neg A_1 \vee \cdots \vee \neg A_n) \Rightarrow \forall \tilde{y}(B \vee \neg B_1 \vee \cdots \vee \neg B_m)$$

$\Leftrightarrow$

$$\neg(\neg(\exists \tilde{x} \neg(A \vee \neg A_1 \vee \cdots \vee \neg A_n) \vee \forall \tilde{y}(B \vee \neg B_1 \vee \cdots \vee \neg B_m)))$$

$\Leftrightarrow$

$$\neg(\forall \tilde{x}(A \vee \neg A_1 \vee \cdots \vee \neg A_n) \wedge \exists \tilde{y}(\neg B \wedge B_1 \wedge \cdots \wedge B_m))$$

$\Leftrightarrow$

$$\neg(\forall \tilde{x}(A \vee \neg A_1 \vee \cdots \vee \neg A_n) \wedge \exists \theta(\neg \theta B \wedge \theta B_1 \wedge \cdots \wedge \theta B_m))$$

where θ is a ground substitution on $\tilde{y}$

$\Leftrightarrow$

$$\forall \theta \text{ (ground)} \neg \left(\left(\begin{array}{l} \theta B_1. \\ \vdots \\ \theta B_m. \\ A \leftarrow A_1, \ldots, A_n. \\ \leftarrow \theta B. \end{array} \right) \text{has a solution} \right).$$

Let us note that in the case where $n = m = 1$, we are back to our small binary program scheme, where the goal and the fact are ground. This case was considered by Schmidt-Schauß, who proved it also to be decidable. He had also shown that it becomes undecidable if $\mathscr{A}$ is a four-literal clause [47].

Later, Marcinkowski and Pacholski proved the three-literal case ($n = 2$) to be undecidable as well. They proved this result for Horn clauses [36,35].

Now, let us consider that $n = 2$ and $m = 1$. Then the class of programs to be satisfied becomes

$$\begin{cases} \theta B_1 \leftarrow . \\ A \leftarrow A_1, A_2. \\ \leftarrow \theta B. \end{cases}$$

which is clearly close to the studied structure.

We are optimistic that the results and/or methods of the previous sections can help to establish the status of the satisfiability of this pattern. Thus, we will provide another proof of the result in [36] with a restriction on the size of $\mathscr{B}$ ($m = 1$).

In the following section, we give a first step in this direction. The proof is established through program transformations.

9.2. Results on Ternary Programs

Theorem 9.1. There is a particular explicitly constructible program in the following form:

$$\begin{cases} p(fact_1) \leftarrow . \\ p(fact_2) \leftarrow . \\ p(left) \leftarrow p(right_1), p(right_2). \end{cases}$$

where fact$_1$ is ground, and either

- *left and right$_1$ are linear*
- *left and right$_2$ are linear*
- *right$_2$ and fact$_2$ are linear*

for which it is undecidable if, for a ground goal " ← goal.," the program halts and if there are some answer-substitutions.

PROOF. We show that any *append*-like program (so, in particular, we can choose the smallest meta-interpreter of Section 8)

$$\begin{cases} p(\,fact\,) \leftarrow . \\ p(\,left\,) \leftarrow p(\,right\,). \\ \leftarrow p(\,goal\,). \end{cases}$$

can be encoded by

$$\begin{cases} p(\,fact, [1], [\ ]) \leftarrow . \\ p(\,goal_2, [\ ], [\ ]) \leftarrow . \\ p(\,left, [X|L], [Y|LL]) \leftarrow p(\,right, [X], LL), p(\,goal, L, Z). \\ \leftarrow p(\,goal_1, [1,1], [1]). \end{cases}$$

where

1. *goal$_2$* is a ground instance of *goal*
2. *goal* does not share any variable with *left* and *right*
3. X, Y, Z, L, LL are new variables not appearing in *left, right, goal*
4. *goal$_1$* is a ground instance of *left* by the substitution σ, such that σ unifies *right* and *fact*: since we consider the emptiness problem, we do not care about the trivial case where *fact* and *right* do not unify (this case does not alter the halting problem). Since we can assume that there exists a most general unifier θ for *fact* and *right*, σ is chosen as an instance of θ which makes *left* ground.

Clearly, the first three conditions are syntactical ones.

Now, we show that from the third step of resolution ($p(goal, [1], Z)$), the derivation of both programs can be the same until the first success: the second program will stop iff the first will, and the second program will stop with a success iff the first one has at least one solution.

After the second derivation step, we obtain the goal $\leftarrow p(goal, [1], Z)$ which has the same derivation as $\leftarrow p(goal)$ in the binary program, except that at each unification with the third clause (the ternary one), a new atom $p(goal, [\ ], -)$ is generated. But these atoms will unify only with the second fact ($p(goal_2, [\ ], [\ ])$) because of the second argument. If, during the resolution, there is a unification of the first goal atom with the first clause, all other goal atoms will be removed after a unification with the second clause. This is shown in Figure 4.

As there is no condition for the original program, our proof is correct, in particular, for all the classes of programs for which the halting and emptiness problem is undecidable. □
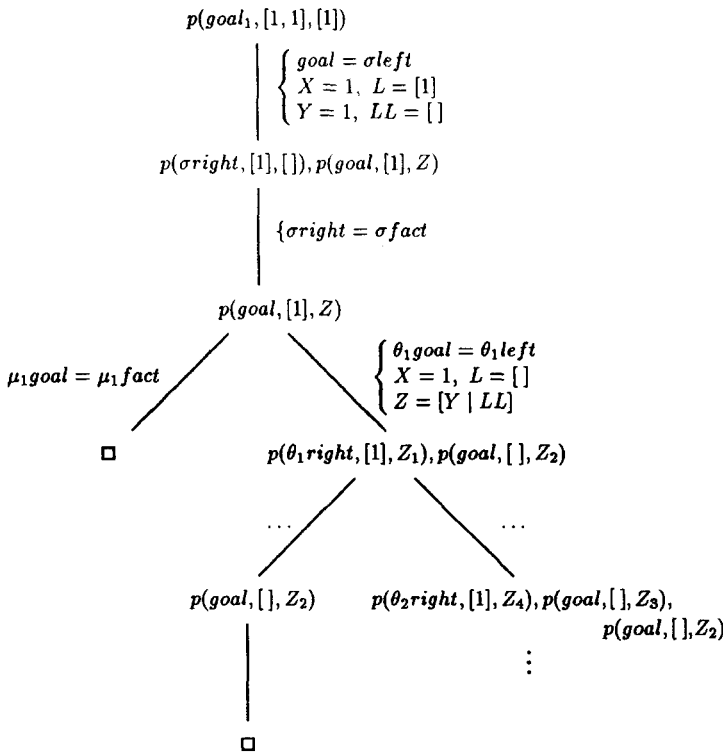
$$p(goal_1, [1,1], [1])$$

$$\begin{cases} goal = \sigma left \\ X = 1, \ L = [1] \\ Y = 1, \ LL = [] \end{cases}$$

$$p(\sigma right, [1], []), p(goal, [1], Z)$$

$$\{\sigma right = \sigma fact\}$$

$$p(goal, [1], Z)$$

$$\mu_1 goal = \mu_1 fact \qquad\qquad \begin{cases} \theta_1 goal = \theta_1 left \\ X = 1, \ L = [] \\ Z = [Y \mid LL] \end{cases}$$

$$\square \qquad\qquad p(\theta_1 right, [1], Z_1), p(goal, [], Z_2)$$

$$\dots \qquad\qquad \dots$$

$$p(goal, [], Z_2) \qquad p(\theta_2 right, [1], Z_4), p(goal, [], Z_3),$$
$$p(goal, [], Z_2)$$
$$\vdots$$

$$\square$$

FIGURE 4. Beginning of the resolution.

If we consider only the halting problem, we have just to encode a program such as

$$\begin{cases} p(left) \leftarrow p(right). \\ \leftarrow p(goal). \end{cases}$$

The coding for a standard Prolog selection rule (depth-first, leftmost atom) is

$$\begin{cases} p(fact_1, [1,1], [\ ]). \\ p(left, [X|L], [Y|LL]) \leftarrow p(right, [X|L], LL), p(goal, L, Z). \\ \leftarrow p(goal_1, [1,1], [1]). \end{cases}$$

where

- $goal_1$ and $fact_1$ are ground instances of, respectively, *left* and *right*
- *goal* does not share any variable with *left* and *right*
- X, Y, L, LL, Z are new variables not appearing in *left*, *right*, and *goal*.

The difference between these two programs is that, if the first case, the second atom (of the ternary clause) will be derived only when the first has been removed. At the second step of resolution, if the first atom is selected, it will only unify with $fact_1$: it will not alter the rest of the resolution. If the second atom is chosen, it will

be the start of a derivation similar to that of the original program, except that an atom $p(goal, [\ ], -)$ will be generated at each step of derivation. If these atoms are not selected before the first atom (standard computation rule), then they will not be: the program never stops (*left* always unifies with *right*), or stops with a failure (because of the second argument, *right* cannot unify with $fact_1$). If any other computation rule is used, then these atoms must be removed (unified with a success), so a second fact must be added to the program to ensure that these atoms will not prune an infinite derivation. Thus, the coding is

$$\begin{cases} p(fact_1, [1,1], [\ ]). \\ p(fact_2, [\ ], [1,1]). \\ p(left, [X|L], [Y|LL]) \leftarrow p(right, [X|L], LL), p(goal, L, Z). \\ \leftarrow p(goal_1, [1,1], [1]). \end{cases}$$

where $fact_2$ is a ground instance of *goal*.

10. CONCLUSION

The two tables below summarize the known results about the halting and emptiness problems depending on the form of the characteristic elements *goal*, *fact*, *left*, and *right* of *append*-like programs:

$$\begin{cases} p(fact) \leftarrow . \\ p(left) \leftarrow p(right). \\ \leftarrow p(goal). \end{cases}$$

goal	*left*	*right*	Termination
ground	any	any	decidable [47]
linear	any	any	decidable [11]
any	linear	any	decidable
any	any	linear	undecidable

goal	*left*	*right*	*fact*	Emptiness
ground	any	any	ground	decidable [47]
linear	any	any	linear	decidable [11]
ground	any	any	any	decidable
any	any	any	ground	decidable
linear	linear	linear	any	decidable
any	linear	linear	linear	decidable
any	any	linear	linear	undecidable
linear	linear	any	any	undecidable
any	linear	linear	any	undecidable

Linearity seems to state the border between decidability and undecidability. For both problems, the groundness of one term ensures decidability. The halting problem becomes decidable as soon as *goal* or *left* are linear. The emptiness problem is decidable if three terms are linear.

The technic based on our encoding of the Conway functions provides a consistent framework for the study of the binary recursive Horn clauses. Indeed, it allows to solve the halting and emptiness problems and many other properties.

The main consequence of the undecidability of the emptiness problem is that the satisfiability for the class of first-order formulas containing four subformulas is undecidable too.

We have shown in this paper that *append*-like programs have the same computational power as Turing machines since we prove that there exists a universal *append*-like program. This result can be seen as an extension of the Böhm–Jacopini theorem [3] to logic programming. As in imperative languages, the simplest nontrivial program scheme can express any partial recursive function. As in the Böhm–Jacopini proof, the transformation can be done automatically.

The results on undecidability justify pragmatic or heuristic approaches to logic programming analysis, as in abstract interpretation or type inference. There is no way to define formal and complete methods to control the most basic recursive pattern. Even in such restrictive classes of programs, most of the interesting properties to provide a more efficient compilation technic are undecidable.

Finally, the proof method based on Conway functions appears to be a powerful and efficient tool for encoding hard problems. As an example, consider [37, 38]. Therein, Marcinkowski proves (among many other results) that uniform boundedness is undecidable for single rule Datalog programs by using Conway functions.

REFERENCES

1. Abramson, H. and Rogers, M. H. (eds.), *Meta-Programming in Logic Programming*, Logic Programming series, MIT Press, 1989.
2. Bibel, W., Hölldobler, S., and Würtz, J., Cycle Unification, *CADE* 94–108 (June 1992).
3. Böhm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM* 9:366–371 (1966).
4. Blair, H. A., The Recursion-Theoretic Complexity of Predicate Logic as a Programming Language, *Information and Control* 54:25–47 (1982).
5. Bouquard, J. L., Logic Programming and Attribute Grammars, Ph.D. Thesis, Orléans 1992.
6. Bratko, I., *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 1986.
7. Chen, H. and Hsiang, J., Recurrence Domains: Their Unification and Application to Logic Programming, Technical Report, Stony Brook (available by anonymous ftp on "sbcs.sunysb.edu"), July 1991.
8. Conway, J. H., Unpredictable Iterations, in: *Proc. 1972 Number Theory Conference*, University of Colorado, pp. 49–52, 1972.
9. Courcelle, B., Fundamental Properties of Infinite Trees, *J. TCS* 17:95–169 (1983).
10. Dauchet, M., Simulation of Turing Machines by a Regular Rewrite Rule, *J. Theoretical Computer Science* 103:409–420 (1982).
11. Dauchet, M., Devienne, P., and Lebègue, P., Weighted Graphs: A Tool for Logic Programming, in: *11th CAAP86*, 1986.

12. Dauchet, M., Devienne, P., and Lebègue, P., Weighted Systems of Equations, *Informatika 91, Grenoble, Special Issue of TCS* (1991).

13. Delahaye, J.-P., Sémantique Logique et Dénotationnelle des Interpréteurs Prolog, *Informatique Théorique et Applications* 22(1):3–42 (1988).

14. De Schreye, D., Verschaetse, K., and Bruynooghe, M., A Practical Technique for Detecting Nonterminating Queries for a Restricted Class of Horn Clauses, Using Directed Weighted Graphs, in: *Proc. ICLP'90*, Jerusalem, MIT Press, June 1990, pp. 649–663.

15. De Schreye, D. and Decorte, S., Termination of Logic Programs: The Never-Ending Story, *J. Logic Programming* (1994).

16. Devienne, P., Les Graphes Orientés Pondérés: Un Outil pour l'étude de la Terminaison et de la Complexité dans les Systèmes de Réécritures et en Programmation Logique, Ph.D. Thesis, Lille, 1987.

17. Devienne, P., Weighted Graphs—Tool for Studying the Halting Problem and Time Complexity in Term Rewriting Systems and Logic Programming, *J. Theoretical Computer Science* 75:157–215 (1990).

18. Devienne, P., Lebègue, P., and Routier, J. C., Weighted Systems of Equations Revisited, in: *Analyse Statique, Actes WSA'92, Bordeaux, BIGRE* 81–82, Sept. 1992, pp. 163–173.

19. Devienne, P., Lebègue, P., and Routier, J. C., Halting Problem of One Binary Horn Clause is Undecidable, in: *Proc. STACS'93*, Würzburg, Springer-Verlag, Feb. 1993.

20. Devienne, P., Lebègue, P., and Routier, J. C., The Emptiness Problem of One Binary Recursive Horn Clause is Undecidable, in *Proc. ILPS'93*, Vancouver, MIT Press, Nov. 1993, pp. 250–265.

21. Devienne, P., Lebègue, P., Routier, J. C., and Würtz, J., One Binary Horn Clause is Enough, in: *Proc. STACS'94*, Caen, Springer-Verlag, Feb. 1994, pp. 21–32.

22. Devienne, P., Lebègue, P., Parrain, A., Routier, J. C., and Würtz, J., Smallest Horn Clause Programs (extended), Technical Report, LIFL, June 1995.

23. Goldfarb, W. and Lewis, H. R., The Decision Problem for Formulas with a Small Number of Atomic Subformulas, *J. Symbolic Logic* 38(3):471–480 (1973).

24. Gaifman, H. and Mairson, H., Undecidable Optimisation Problems for Database Logic Programs, in: *Symp. on Logic in Computer Science*, New York, 1987, pp. 106–115.

25. Gallagher, J. and Bruynooghe, M., Some Low-Level Source Transformations for Logic Programs, in: *Proc. 2nd Workshop on Meta-Programming in Logic*, Leuven, Belgium, Apr. 1990, pp. 229–244.

26. Guy, R. K., Conway's Prime Producing Machine, *Mathematics Magazine*, 56:26–33 (1983).

27. Harel, D., On Folk Theorems, *Commun. ACM* 7:379–389 (1980).

28. Hanschke, P. and Würtz, J., Satisfiability of the Smallest Binary Program, *Inform. Processing Lett.* 45(5):237–241 (Apr. 1993).

29. Hansson, Å. and Tärnlund, S. Å., Program Transformation by Data Structure Mapping, in: *Logic Programming*, K. L. Clark and S. Å. Tärnlund (eds.), *APIC Studies in Data Processing*, Academic Press, 1982, pp. 117–122.

30. Lagarias, J. C., The $3x + 1$ Problem and Its Generalizations, *Amer. Math Monthly* 92:3–23 (1985).

31. Lagarias, J. C., Annotated Bibliography on Collatz Problem," Private Communication, 1992.

32. Lebègue, P., Contribution à l'Etude de la Programmation Logique par les Graphes Orientés Podérés, Ph.D. Thesis, Lille, 1988.

33. Lewis, H. R., The Decision Problem for Formulae with a Bounded Number of Atomic Subformulae, *Notices Amer. Math. Soc.* 20 (1973).

34. Lloyd, J. W., *Foundations of Logic Programming* (2nd, ext. ed.), Springer-Verlag, 1987.

35. Marcinkowski, J., A Horn Clause that Implies an Undecidable Set of Horn Clauses, Private Communication, 1993.

36. Marcinkowski, J. and Pacholski, L., Undecidability of the Horn-Clause Implication Problem, in: *FOCS*, 1992.
37. Marcinkowski, J., The 3 Frenchmen Method Proves Undecidability of the Uniform Boundedness for Single Recursive Rule Ternary DATALOG Programs, Submitted (1995).
38. Marcinkowski, J., Undecidability of Uniform Boundedness for Single Rule Datalog Programs, Submitted (1995).
39. Minsky, M., *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967.
40. Parrain, A., Transformations de Programmes Logiques et Sémantique Opérationnelle, Ph.D. Thesis, Lille, Feb. 1994.
41. Parrain, A., Devienne, P., and Lebègue, P., Prolog Programs Transformations and Meta-Interpreters, in: *Logic Program Synthesis and Transformation, LOP-STR'91*, Manchester, Springer-Verlag, 1991, pp. 228–241.
42. Post, E. M., A Variant of a Recursively Unsolvable Problem, *Bulletin Amer. Math. Soc.* 46:264–268 (1946).
43. Rogers, H., *Theory of Recursive Functions and Effective Computability*, MIT Press, 1987.
44. Routier, J. C., Termination, Satisfiability and Computational Power of One Binary Horn Clause, Ph.D. thesis, Lille, Feb. 1994.
45. Salzer, G., Solvable Classes of Cycle Unification Problems, in: *IMYCS, Smolenice (CSFR)*, 1992.
46. Shmueli, O., A Single Recursive Predicate is Sufficient for Pure Datalog, *Inform. and Computation* 117:91–97.
47. Schmidt-Schauß, M., Implication of Clauses is Undecidable, *J. Theoretical Comput. Sci.* 59:287–296 (1988).
48. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, in: S.-Å. Tärnlund (ed.), *2nd Int. Logic Programming Conf.*, Uppsala, 1984, pp. 127–138.
49. Tärnlund, S. Å., Horn Clause Computability, *BIT* 172:215–226 (1977).
50. Vardi, M., Decidability and Undecidability Results for Boundedness of Linear Recursive Queries, in: *Symp.. Principles of Database Systems*, Austin, TX, 1988, pp. 341–351.
51. Würtz, J., Unifying Cycles, in: *Proc. European Conf. on Artificial Intelligence*, Aug. 1992, pp. 60–64.