Computational Intelligence, Volume ??, Number 000, 20??

UNIVERSALITY AND PREDICTION IN BUSINESS RULES

OLIVIER WANG
CNRS LIX, Ecole Polytechnique and *IBM France*
IBM France, 9 rue de Verdun, 94250 Gentilly, France

CHRISTIAN DE SAINTE MARIE, CHANGHAI KE
IBM France, 9 rue de Verdun, 94250 Gentilly, France

LEO LIBERTI
CNRS LIX, Ecole Polytechnique, 91128 Palaiseau, France

Business Rules have the form ⟨if *condition* then *action*⟩. A Business Rules program, which can be executed by means of an interpreter, is a sequence of Business Rules. Motivated by IBM use cases, we look at the problem of setting parameter values in a given Business Rules program so it will achieve a given average goal over all possible instances. We explore the following fundamental question: is there a general learning algorithm which addresses this issue? We prove the answer is negative. On the positive side, we derive operational semantics for Business Rules programs. As a proof of concept, we show empirically that these can be used to detect potential non-termination situations.

Key words: Business rules; Statistical learning; Operational semantics; Turing-completeness.

1. INTRODUCTION

Rules are among the oldest and best studied knowledge representation paradigms for automated reasoning (Brachman and Levesque, 2004). Business Rules (BR) are a newer concept, devised to allow corporations to digitally encode business process knowledge into a centralized storage and into a manageable form.

A more complete definition, given in natural (rather than formal) language, is given in the *BR manifesto* (Business Rules Group, 2017). The BR manifesto says, among other things, that BRs are sentences in natural language, aimed at business people, but susceptible to be parsed by a computer system into a BR management system. Their goal is to encode business processes into a set of easily and automatically managed digital objects.

The fact that most BR Management Systems (BRMS) are also able to *execute* those rules as if they were computer programs means that BRs, written by business people in natural language, are eventually turned into formal sentences, as no computer-executed code can ever be ambiguous at the machine level.

Here, we study the problem of parametrizing BRs so their output will conform to a certain prescribed average behaviour. This is necessary when business strategies must be constrained at the global level (by choice or regulatory necessity) without unduly influencing any of the local business processes encoded by BRs. We show that it is impossible to find a single computer algorithm which can solve this problem for any given BR, but we derive an operational semantics for proving formal properties for some BR subsets.

E-Mail:olivier.wang@polytechnique.edu
E-Mail:{csma,changhai.ke}@fr.ibm.com
E-Mail:liberti@lix.polytechnique.fr

© 20?? The Authors. Journal Compilation © 20?? Wiley Periodicals, Inc.

1.1. Scope

In this paper we shall consider BRs in their (restricted) interpretation as formal sentences of a programming language.

We note that there is no universally accepted "generic" language for expressing BR with (von Halle, 2001; Ross, 2003; Giurca et al., 2009). Since most of the theoretical part of this paper is concerned with an inexistence theorem, we do not need to concern ourselves with a multitude of dialects. It suffices to consider a rather minimal BR language variant consisting of BRs of the form "if *condition* then *action*". The *action* clause assigns a sequence of values to a corresponding sequence of variables, or executes a script which interacts dynamically with the environment.

We already mentioned that most BRMSs come with execution platforms, called *interpreters*. As for the BR languages themselves, there are many interpreters, more or less powerful (Ligeza, 2006; Hanson and Hasan, 1993), the classic one being the *Rete algorithm* (Forgy, 1982). For the same reasons as before, we do not need to consider all the variants; a minimal interpreter will suffice.

The research content reported in this paper was commissioned and co-authored by IBM, specifically the research group that develops and maintains the IBM BRMS, called *Operational Decision Manager* (ODM). Our minimal BR language and interpreter are both acceptable abstract models for the Rete-based engine used in ODM. While many users of industrial BRMS do not exploit the sophisticated Rete-based engine, it is nonetheless used by about twenty percent of BRMS applications, based on private IBM data. Example of rules that use Rete chaining include: price computation, fraud detection, and route planning. The fact that our work rests on an abstract model of ODM makes it relevant in most if not all major BRMS (FICO Blaze, JBoss Rules, Oracle BRE), as they all have a similar Rete-based engine. Similarly, some of the major free and/or open-source rule engines are also Rete-based, such as CLIPS (Culbert and Riley, 2003), Jess (on which Oracle BRE is based) (Friedman-Hill, 2003), or DROOLS (on which JBoss Rules is based) (Proctor, 2011). This means that potentially up to twenty percent of the whole Business Rules market motivates this work, and it is relevant for the whole of the field, not just IBM.

To summarize, we call a *BR program* a set of "if *condition* then *action*"-formatted BRs executed by a basic BR interpreter. We call the union of all valid BR programs the *BR programming language*.

1.2. Motivation and relevance of the main problem

In this paper we study the following problem: can a given BR program be parametrized so that it behaves statistically according to a prescribed goal on every input? More precisely, the question we answer is the following.

BUSINESS RULES WITH AVERAGE GOALS (BRAG).
Determine whether there exists an algorithm $\mathcal{A}$ which takes as an input:

- a BR program $\rho = P_p(x)$, where p is a parameter vector, x is an input vector, and ρ is a scalar output,
- the set X of all possible inputs of P,
- a prescribed set G of values for ρ,

and which returns, as an output, a vector $p^* = \mathcal{A}(P, G, X)$ such that:

$$\mathbb{E}(P_{p^*}(x) \mid x \in X) \in G, \tag{1}$$

where $\mathbb{E}(\cdot)$ denotes the expectation.

The BRAG problem arose out of the request of many users of the IBM ODM BRMS. Why is this question important to industry? We answer by way of a case scenario proposed by the IBM team which maintains ODM.

Consider a bank which has a process (encoded as a parametrizable BR program P_p) for deciding whether to grant a loan to a given customer. The test conditions in P_p may verify

anagraphic, work-related and credit ranking data about the customer (encoded in a vector x), and p is usually given by acceptable thresholds for customer data or functions thereof. The output ρ of P_p in this case might be a YES/NO/REVIEW type value, where REVIEW corresponds to cases requiring a personalized follow-up by a financial advisor. Of course, banks have to consider the volume of REVIEW outputs carefully. Accordingly, banks will try to choose p so that P_p automates only a given fraction $g \in [0, 1]$ of the decisions *over all loan requests on average*. An obvious way to deal with the issue could be to pick an appropriate training sample S and run some Machine Learning (ML) algorithm to learn the BR program configuration parameter vector p so as to satisfy Eq. (1). This, finally, begs the question above: is this direction even theoretically possible, in full generality? In this paper we will argue that this is not possible in general.

To achieve this negative result, we look at the question from the point of view of the expressiveness of the BR programming language. Within the panorama of programming languages, one can distinguish two main categories: imperative and declarative. Since both categories contain Turing-complete languages, a separation of the two categories according to computational expressive power is impossible. On the other hand, by looking at three basic building blocks present in all imperative languages (assignments, tests and loops), we can informally segment programming languages more finely: purely imperative languages have explicit constructs for all three blocks, and purely declarative languages do not have explicit constructs for any of those building blocks. One of the earliest computational models, the lambda-calculus (Church, 1932), embodies loops within the language itself by using recursion, while tests and assignments may be simulated using boolean variables and arithmetic operators; the situation of Prolog (Clocksin and Mellish, 1987) is similar. Constraint Programming (Apt, 2003) and Mathematical Programming (Williams, 1999) appear to be purely declarative, in the sense that the language itself does not provide constructs of assignments, tests or loops. Perhaps the purest form of a declarative language is given by systems of Diophantine equations, famously shown to be Turing-complete when Hilbert's 10th problem was solved in the negative (Matiyasevich, 1993). In this taxonomy, the BR language explicitly provides assignments and tests, but has no loop construct. Among the other programming languages we mentioned, recursive functions have been famously proved to be equivalent to Turing Machines (TM) by A. Turing himself (Turing, 1937). For inference rules, including Prolog, we refer the reader to (Sneyers et al., 2005). For Mathematical Programming, see (Liberti and Marinelli, 2014).

1.3. Contributions

This paper makes the following contributions: (a) it provides a constructive proof of Turing-completeness for the BR language; (b) it formally shows that this language is unlearnable in the Probably Approximately Correct (PAC) framework (see Sect. 4), and therefore that ML tools cannot be used in general to "learn the behavior" of BR programs statistically; (c) it proposes an operational semantics for the BR language. As a token of the practical applicability of our ideas, we show that on a few small but interesting examples our operational semantics can prove termination of entire sets of programs having some input parameter ranging over a given interval. We note that the same operational semantics can also be used to prove non-termination. Moreover, failure to prove termination can be taken as indicative of a risk of non-termination.

With respect to (Wang et al., 2016), the Turing-completeness proof presented here, the operational semantics, and the associated proof-of-concept implementation, are new. We remark that since the BR language itself is Turing-complete, the termination of its programs, which we claim can be computed by our operational semantics, is obviously an undecidable problem. As a consequence, operational semantics can prove termination and

non-termination of *some* sets of programs, but not of *every* program. We also remark that, as far as we could ascertain, ours is the first proof in the literature of the PAC-unlearnability of BR programs.

1.4. Structure of the negative result

In order to prove that the BRAG problem has no solution, we argue that the dynamics of a generic BR program cannot be machine-learned in a certain well-known abstract learning model (PAC learning, see Sect. 4). We achieve this by showing that certain functions witnessing non-PAC-learnability can be implemented using BR programs; and we prove this latter property by showing that the BR programming language is Turing-complete.

That certain declarative rule-based languages are Turing-complete is already known (Sneyers et al., 2005), but we provide here a new proof that also opens the door to the construction of a positive result: an operational semantics for BRs that can be used empirically to determine wether some given BR programs terminate or not (as mentioned above, since BRs are Turing-complete, determining termination in general is impossible — our operational semantics can prove termination/nontermination of certain instances, but certainly not all).

More specifically, our proof that the BR language is Turing-complete is centered around the idea that BR programs, when executed via an appropriate execution algorithm, behave like WHILE programs, which are known to be Turing-complete (Harel, 1980). From Turing-completeness, we conclude that any computable function can be simulated by a (universal) BR program, including functions that output pseudo-random numbers. This is the main idea behind the proof of PAC-unlearnability. Our operational semantics for BR programs is derived from the reduction from the WHILE language in the Turing-completeness proof.

1.5. Contents

The rest of this paper is organized as follows. Section 2 is used to formalize the notations and concepts of BRs we use in this paper. Section 3 exhibits our constructive proof of Turing-completeness, and recalls another, non-constructive one. Section 4 uses Pseudorandom Functions (PRF) and a chaotic map to prove PAC-unlearnability in both a weak and a strong sense. Section 5 showcases an operational semantics for BRs, discusses a proof-of-concept implementation, and illustrates some of its possible applications on small number of test cases.

2. PRELIMINARIES

Limited to the scope of this paper (Sect. 1.1), BR programming can be seen as programming for non-programmers. The two most difficult computer programming concepts for a layperson to understand appear to be loops and function calls. BR disposes of the former by automatically executing programs over a loop construct embedded in the interpreter, and of the latter by removing them entirely. In commercial BR management systems such as ODM function calls are replaced by entities called "meta-variables", which have no relevance to the present discussion.

2.1. Formalization

A BR program simply consists of an ordered set of rules. Given a variable sequence x, a rule is defined as shown in Alg. 1; T is the condition and A describes the action. While the variables are sometimes typed, we dispense with this distinction in the general discussion.

Algorithm 1 A Business Rule

```
if T(x) then
    x ← A(x)
end if
```

At least one of the variables, say x_1 without loss of generality, is selected to be the *output* of the BR program.

As stated in Sect. 1.1, there are many variations of BR interpreters, but we need only employ a very basic one. Most deterministic interpreters are semantically equivalent to the RIF formalization (de Sainte Marie et al., 2013). We therefore use the interpreter $\mathscr{I}_0$ consisting of the following algorithm:

(1) select the rules for which the condition is True, using the current values of the variables;
(2) execute the action of the first rule in the current selection or stop if there is no such rule;
(3) repeat from Step (1).

The selection of the rule to be executed in Step (2) is referred to as *conflict resolution*. Our algorithm has a simplistic conflict resolution strategy: whenever more than one rule instance could be executed, the one selected is obtained from a fixed total order on rule instances. An example of the execution of such an algorithm is described in Fig. 1.

While we have used a very simple execution algorithm where conflict resolution is based on a fixed ordering of rules, most BR execution algorithms are more complex. Any non-trivial interpreter, however, can be simulated by the basic one $\mathscr{I}_0$ (except in extreme cases, such as a conflict resolution strategy leading to a bounded number of execution loops).

The Variables
int x ← 1
int age ← 90

The Rules (in order)
In the total order considered by the algorithm, they are:
R_1:
  if(age $\geqslant 1 \wedge$ x $= 2$)
  then(age ← 0,x ← 0)
R_2:
  if(x $= 1$)
  then(x ←x$+1$)
R_3:
  if(age $\geqslant 60$)
  then(age ←age$+1$)

The Execution

Iteration 1:
Truth value of conditions:
$t(R_1) =$ False
$t(R_2) =$ True
$t(R_3) =$ True
Rule instances selected (in order): R_2
Rule executed: R_2
Variable values:
x $= 2$
age $= 90$

Iteration 2:
$t(R_1) =$ True
$t(R_2) =$ False
$t(R_3) =$ True
Rules selected: R_1, R_3
Rule executed: R_1
Variable values:
x $= 0$
age $= 0$

Iteration 3:
$t(R_1) =$ False
$t(R_2) =$ False
$t(R_3) =$ False
Rules selected: None
Rule executed: None

END

FIGURE 1. Example illustrating the execution algorithm

2.2. Some remarks on non-trivial BR program interpreters

The most common BR interpreters use an execution algorithm with a structure similar to ours, but with a different conflict resolution strategy in step 2. Common conflict resolution strategies combine at least the following three elements (de Sainte Marie et al., 2013):

- *Refraction* which prevents a rule instance from firing (being selected by the conflict resolution algorithm) again unless its condition clause has been reset.

• *Priority* which is a kind of partial order on rules, leading of course to a partial order on rule instances.
• *Recency* which orders rule instances in decreasing order of continued validity duration (when rule instances are created at run time, it is often expressed as increasing order of rule instance creation time).

We will see in the last section that these other interpreters do not preclude the use of the operational semantics techniques we introduce in this paper.

3. TURING-COMPLETENESS

3.1. Basic notions

A Universal Turing Machine (UTM) is a Turing Machine (TM) which can simulate any other TM on arbitrary input (Shannon, 1956; Turing, 1937). Let L be a programming language for the UTM U, described for example by its formal grammar. By means of a special program $\mathscr{I}$ called *interpreter*, programs written in L can be executed on U (Minsky, 1972). If a programming language L can be used to write all possible programs of a UTM so they are executed via an interpreter, then L is said to be Turing-complete.

Definition 1 (Turing-completeness): Let U be a UTM, which takes as input a string (T, x) consisting of a TM description and its input. A programming language L is *Turing-complete* if there exists an interpreter $\mathscr{I}$ such that for each possible input (T, x) of U there is a program p in L with $\mathscr{I}(p) = (T, x)$.

Note that in Defn. 1, $\mathscr{I}$ is seen as a function that maps programs p in L to the input (T, x) of the UTM U. It is assumed that U will then execute with (T, x) as input. The output of this computation is the effect that the program p has when executed on U via the interpreter $\mathscr{I}$.

We can replace "UTM" in Defn. 1 by any universal computer described in any Turing-complete language L', since interpreters can be composed via translators. Given two programming languages L, L' with interpreters $\mathscr{I}, \mathscr{I}'$ running on UTMs U, U', a *translator* is a pair of functions $(\mathcal{T}, \tau)$, where $\mathcal{T}$ maps programs of L to programs of L' and τ maps valid outputs of U' to valid outputs of U, such that:

$$\forall p \in L \quad U(\mathscr{I}(p)) = \tau(U'(\mathscr{I}'(\mathcal{T}(p)))). \tag{2}$$

In other words, the effect of running p through $\mathscr{I}$ is related to the effect of running the translation of p through $\mathscr{I}'$. Most commonly τ is the identity, i.e. we want the translated program to yield the same output as the same program.

The type of translators we are interested in are the *faithful* ones, i.e. those for which $\mathcal{T}$ is surjective and τ is bijective. The meaning of a surjective mapping $\mathcal{T}$ is technical (see the proof of Thm. 1) and can be relaxed. In particular, if one were to write $\mathscr{I}'$ as a program p' of L', mapping every p to the interpreter p' could suffice. The bijection on the ranges of the UTMs, on the other hand, is essential, and ensures universality. Such translators allow the "expressive power" of the corresponding languages to match. Next, we will state and prove a theorem that formalizes the concept of translation needed in this paper.

Theorem 1: Let L be a Turing-complete programming language with interpreter $\mathscr{I}$ running on a UTM U. Let L' be another programming language, U' another UTM, and $(\mathcal{T}, \tau)$ a faithful translator from L to L'. Then there exists an interpreter $\mathscr{I}'$ of L' on U' that makes L' Turing-complete.

Proof. Let (T', x') be an input of U'. Since τ is a bijection, there is an input (T, x) of U

such that

$$U(T,x) = \tau(U'(T',x')). \qquad (*)$$

Since L is Turing-complete, there is a program $p \in L$ such that $\mathscr{I}(p) = (T,x)$. Let $p' = \mathcal{T}(p)$, and define $\mathscr{I}'(p') = (T',x')$. Since $\mathcal{T}$ is surjective, $\mathscr{I}'$ is defined for all $p' \in L'$. Now by definition of translator we have $U(\mathscr{I}(p)) = \tau(U'(\mathscr{I}'(\mathcal{T}(p))))$. Since $\mathscr{I}(p) = (T,x)$, by $(*)$ the latter equality gives $\tau(U'(\mathscr{I}'(p'))) = \tau(U'(T',x'))$ which, since τ is a bijection, implies $U'(\mathscr{I}'(p')) = U'(T',x')$, which proves that the definition of $\mathscr{I}'$ is consistent with an interpreter. We have shown that for each input (T',x') of U' there is a program p' of L' that is interpreted to (T',x'), which makes L' Turing-complete. □

The computability concepts discussed above are at the basis of computability theory, and have been well known for at least 60 years. Thm. 1 could also be proved using a result of (Curtis, 1965), which states that since a UTM is defined as a TM which is able to simulate any other TM, L can be proven Turing-complete by showing that for any TM, L can be used to describe that TM via its interpreter. It would then suffice to give an explicit mapping between the formal grammars of L and L', leading to a semantics for L' in terms of L. We strongly suspect that even the statement and proof of Thm. 1 given above is not new, at least in its main traits, though we could not find it in the (overwhelmingly rich) literature on the matter. Most of these observations hold under the Church-Turing thesis (Church, 1936; Turing, 1939; Gandy, 1980), according to which any effectively computable function is Turing-computable. In other words, no device or program can compute a function that a UTM cannot.

3.2. WHILE programs

The Turing-completeness of WHILE-programs is well-known (Harel, 1980). We exploit this fact and the previous results to prove the Turing-completeness of BR programs. We show that any WHILE-program can be programmed using a set of BRs and the basic BR interpreter. Furthermore, any non-trivial BR interpreter involving a loop is also Turing-complete, as it can simulate the basic interpreter used in this article.

The WHILE programming language has been studied with or without a name as the simplest form of imperative programming since 1969 (Hoare, 1969). It has three simple syntactic elements: assignments, conditional actions (**if**. . . **then** blocks), and **while** loops.

A WHILE program has the canonical (recursive) form (Harel, 1980; Hirose and Oya, 1972):

while $\mathsf{T}_0(x)$ **do**
  ifblock$_1(\mathsf{T}_1, \mathsf{A}_1, x)$
  ...
  ifblock$_K(\mathsf{T}_K, \mathsf{A}_K, x)$
end while

where, for each $k \leqslant K$, **ifblock**$_k(\mathsf{T}_k, \mathsf{A}_k, x)$ is defined either as:

if $\mathsf{T}_k^1(x)$ **then**
  $x \leftarrow \mathsf{A}_k^1(x)$
  ifblock$(\mathsf{T}_k, \mathsf{A}_k, x)$
end if

or as an empty command. The interpretation of the symbols $\mathsf{T}_k^i(x)$ and $\mathsf{A}_k^i(x)$ is: T are (possibly jagged) tensors of Boolean conditions on the variables x, which evaluate to True or False, and A is a (possibly jagged) tensor of functions of x yielding values to be assigned to the variables.

In other words, a WHILE program is a single conditional loop containing a sequence of (possibly nested) test conditions followed by a conditional assignment action.

3.3. Translation of WHILE to BR

We prove that the BR language is Turing-complete (under the basic interpreter) by providing a translation of WHILE programs using BRs.

Theorem 2: Any WHILE program is computable by an equivalent BR program.

We prove this by showing that a generic WHILE program can be interpreted into a BR program. The only requirement of the interpretation is to be computable. We first prove this for WHILE programs without nested **if** statements, then we describe a sequence of syntactical steps on the symbols of a generic WHILE program which transforms it into a WHILE program without nested **if** statements.

Lemma 1: Any WHILE program without nested **if** statement can be translated to a BR program.

Proof. Given the following WHILE program without nested **if** statements:

```
1: while T_0(x) do
2:     if T_1(x) then
3:         x ← A_1(x)
4:     end if
5:     ...
6:     if T_K(x) then
7:         x ← A_K(x)
8:     end if
9: end while
```

We can write an equivalent BR program with $K + 1$ rules:

```
1: if ¬B_0(x_1, ..., x_n) then
2:     Stop
3: end if
4: if B_1(x_1, ..., x_n) then
5:     x ← A_1(x)
6: end if
7: ...
8: if B_K(x_1, ..., x_n) then
9:     x ← A_K(x)
10: end if
```

Using previous notations, rule R_0 has $T(x) = \neg \mathsf{B}_0(x)$ with $Stop$ as action, while for $k \in \{1, ..., K\}$ rule R_k has $T(x) = \mathsf{B}_k(x)$ and $A(x) = \mathsf{A}_k(x)$. This is obviously equivalent to the WHILE program considered. □

Lemma 2: Any WHILE program can be transformed into an equivalent WHILE program without nested **if** statements.

Proof. This is an easy proof, as the lemma amounts to saying any nested **if** statements can be unnested. A simple inductive reasoning on the depth of the nesting proves the result (possibly adding boolean variables to store the value of intermediate levels tests). The property applied to each **ifblocks** of a WHILE program transforms it into the form we want. □

We summarize the discussion in the following proposition.

Proposition 1: There is a faithful translator $(\mathcal{T}, \tau)$ from the WHILE language to the BR language.

Proof. Note that τ is the identity, which is a bijection. □

Finally, we invoke Thm. 1.

Corollary 1: The BR language is Turing-complete.

3.4. A direct proof

While the above proof is sufficient to justify the Turing-completeness of BRs, we can also use a much more direct proof by exhibiting a BR program that simulates a UTM. Such a BR program is exhibited in Fig. 2. We use meta-variables to make the BR program readable, as R_1 would otherwise be written as $Q \times 3 \times Q \times S$ distinct rules.

We suppose the variables include the following:
- many (static) state objects of type "state": $q_1, \ldots, q_Q$
- many (static) symbol objects of type "symbol": $s_1, \ldots, s_S$
- a (static) finite set of terminal states of type "terminal": T_{er}
- a (static) blank symbol of type "symbol": s_b
- a (static) set of Turing rules of type "rules", of the form

(state$_{\text{initial}}$, symbol$_{\text{initial}}$, right|left|stay, state$_{\text{next}}$, symbol$_{\text{written}}$):

$R = \{(q_r^i, s_r^i, \text{act}_r, q_r^f, s_r^f) \mid \text{act}_r \in \{\text{“left”}, \text{“right”}, \text{“stay”}\}\}_r$
- the current state of type "state": q
- the length of the visible tape data, of type "length": l
- the current visible tape data of type "tape": $T = \{(i, s_i) \mid i \in \mathbb{N}, 0 \leqslant i \leqslant l-1\}$ where l is the length of the visible tape data
- the current place on the tape of type "position": p

We use the following meta-variables in the BR program that simulates a UTM:
- α_{qf} of type "state"
- α_{sf} of type "symbol"

The BR program to simulate a UTM is then written in a compact form:

R_1:
if
  $(q, T(p), \text{act}, \alpha_{qf}, \alpha_{sf}) \in R$
then
  $q \leftarrow \alpha_{qf}$
  $T \leftarrow (T \setminus \{(p, T(p))\}) \cup \{(p, \alpha_{sf})\}$
  $\alpha_p \leftarrow (\text{"position"}, p \pm 1)$(Depending on the value of act)
  $\alpha_l \leftarrow (\text{"length"}, l \pm 1)$(Depending on the respective values of act, p and l)

R_2:
if
  $(q \in T_{er})$
then
  Stop;

FIGURE 2. A UTM written in the BR language.

The UTM in Fig. 2 terminates correctly for any valid input (T, x) where T is a TM and x is its input. We have made simplifications for the sake of clarity: R_1 should clearly be at least three different rules each replacing act with one of {"left", "right", "stay"}, and its complete formally correct form would in fact have two more rules, to be able to increase the length of the tape as needed (using the variable s_b as necessary).

4. PAC-UNLEARNABILITY

Because BRs are Turing-complete, as shown in Sect. 3, their computability is undecidable. We now introduce the main feature concerning the BRAG problem: the parameters.

Suppose BRs have tuning parameters, such as the threshold for α_1 in R_1 from Fig. 1. A common goal of BR users is to achieve an average result, such as having an average number of manually treated loan requests for a BR modeling a loan application process. This begs the following question: is there an algorithm for deciding the values of the parameters of a BR program in such a way that it statistically behaves according to a given target? As BR programs are Turing-complete, we know that this algorithm cannot exist in the most general terms, since BR programs might not even terminate in finite time.

We now look at those BR programs which *do* terminate. The question can be considered as a learnability problem. Does there exist an algorithm $\mathcal{A}$ that efficiently "learns" p, given as input: (a) a class $\mathscr{P}_p$ of terminating BR programs parametrized over p, (b) a data distribution $\mathcal{D}$ over its input domain X and (c) a goal g for the value of the average output $\mathbb{E}_{\mathcal{D}}(\mathscr{P}_p)$? We prove that the answer is still negative.

4.1. Basics of computational learning theory

The first thing to establish is the exact meaning of the word "learns", used informally above. Describing ML algorithms and their applicability to different problems is the concern of computational learning theory. There exist several approaches to this task, which can be broadly divided in two: those that specify that a successful learning algorithm must learn an unknown function f exactly from data observation, and those that accept learning an approximation of f. We focus on the second category, and, among the several abstract models of learning, we look at the well-established Probably Approximately Correct (PAC) framework (Valiant, 1984).

A PAC learning algorithm aims at identifying a *concept* (i.e. a function $X \to \{0,1\}$) in a *concept class* $\mathscr{C}$ (i.e. a family of concepts). For a concept $f \in \mathscr{C}$ and a list S of data points in X of length λ, an algorithm $\mathcal{A}$ is an (ϵ, δ)-PAC learning algorithm for $\mathscr{C}$ if for all sufficiently large λ it outputs a function h such that:

$$\mathbb{P}[h \text{ is an } \epsilon\text{-approximation of } f] \geqslant 1 - \delta.$$

We note that $\mathcal{A}$ has access to an oracle for f. Moreover:

- $\mathcal{A}$ is said to be efficient if the time complexity of $\mathcal{A}$ and h are polynomial in $1/\epsilon$, $1/\delta$ and λ;
- $\mathcal{A}$ is said to weakly learn $\mathscr{C}$ if there exist some polynomials $p_\epsilon(\lambda)$; $p_\delta(\lambda)$ for which $\epsilon \leqslant \frac{1}{2} - \frac{1}{p_\epsilon(\lambda)}$ and $\delta \leqslant 1 - \frac{1}{p_\delta(\lambda)}$.
- We say a concept class is PAC learnable if it is both efficiently and weakly learnable. Otherwise, it is unlearnable.

The question is now whether the BRAG is PAC learnable, i.e. whether the correct parametrizations of arbitrary BR programs leading to good control of the average can be learned weakly and efficiently. We shall prove that the answer is negative in the general case. Within the same notation and context, another question that arises is whether there even exists an algorithm which learns p with $\epsilon < \frac{1}{2}$ and $\delta < 1$. Again, the answer is negative in the general case.

The learning problem has been looked at within the context of other Turing-complete languages. Algorithms for learning some restricted classes of programs exist for inductive logic programming (Blockeel and De Raedt, 1998) or nonmonotonic inductive logic programming (De Raedt and Džeroski, 1994). Here, similarly to (Wang et al., 2016), we wish to examine the general case of learning in all terminating programs.

4.2. Pseudorandom Functions

Pseudorandom functions (PRF), introduced by Goldreich, Goldwasser and Micali (Goldreich et al., 1986), are indexed families of functions F_p for which there exists a polynomial

time algorithm to evaluate $F_p(x)$, but no probabilistic polynomial time algorithm can distinguish the function from a truly random function F_{rand} without knowing p, even if allowed access to an oracle.

It is known that PRF are unlearnable using PAC algorithms (Goldreich et al., 1986; Cohen et al., 2015). This allows us to prove that the general class of terminating BR programs is not PAC-learnable (i.e. that no PAC algorithm can learn the concept weakly and efficiently). We will then use a known chaotic map, the logistic map (defined by $f_{n+1}(x) = af_n(x)(1 - f_n(x))$ with $0 \leqslant a \leqslant 4$), to look at learnability in a stronger sense, and prove that BR programs cannot be learned by PAC learning algorithms *at all*, in full generality.

In the rest of this section, we consider F_p to be a PRF, and denote by $\text{Eval}_{p,x}(F_p(x))$ the complexity of evaluating $F_p(x)$.

4.3. Weak unlearnability

We call $(\mathscr{P}_p)_{p\in\pi}$ a class of terminating BR programs indexed by p, S a list of items from the input domain X with $|S| = \lambda$, and g a goal for the value of the average output $\mathbb{E}_S(\mathscr{P}_p)$. We consider C to be the concept class whose members are $f : (\mathscr{P}_p)_{p\in\pi} \to \{0, 1\}$.

Next, we prove that there is no practically viable algorithm that can learn a BR program out of a class of BR programs in the general case, even with access to a perfect oracle. This is a consequence of both the Turing-completeness of BR programs and the unlearnability of PRF.

Proposition 2: The concept $h \in C$ defined as $h(p) = 1$ iff $\mathbb{E}_S(\mathscr{P}_p) = g$ cannot be learned using a PAC learning algorithm in the general case.

Proof. As BR programs are Turing-complete, we let the family $(\mathscr{P}_p)_{p\in\pi}$ to consist of PRFs. Any algorithm which learns C also learns $(\mathbf{1}_f(p))_p \subset C$, where $\mathbf{1}_f(p) = 1$ iff $\mathscr{P}_p = f$. Learning the latter is trivially the same as learning a PRF, which is impossible by (Cohen et al., 2015). □

Corollary 2: The concept class C is unlearnable.

Hence, it is impossible to provide a single algorithm that is able to adjust the average behavior of BR programs according to a predefined goal.

Corollary 3: The BRAG is PAC-unlearnable.

4.4. Complete Unlearnability

We have used the fact that PRFs are not PAC learnable in the sense that no PAC algorithm can efficiently and weakly learn a PRF. We now demonstrate an example of a concept class that cannot be learned by PAC algorithms at all (i.e. not just because the "weakly" and "efficiently" qualifiers do not apply). This example is based on the intuition that chaos cannot be predicted, and so it cannot be learned.

We use the logistic map, as a known chaotic map. We note $f_{n+1}(x) = af_n(x)(1 - f_n(x))$, $f_0(x) = x$ and choose the parameter $a = 4$. We call $C_n(x)$ the concept class such that $C_n(x) = 1$ iff $f_n(x) \geqslant 0.5$ and $C_x(n) = 0$ otherwise, where $x \in [0, 1]$ follows the arcsine distribution, i.e. the probability density function is $p(x) = \frac{1}{\pi\sqrt{x(1-x)}}$, and with $n \in \mathbb{N}$ following the uniform distribution.

Theorem 3: The concept class $C_n(x)$ cannot be learned with any accuracy. To be precise, for all algorithms $\mathcal{A}$ calling the oracle $C_n(x)$ a finite number of times, we have:

$$\mathbb{P}_{n\in\mathbb{N}}(\mathbb{P}_{x\in X}(\mathcal{A}(C_n)(x) \neq C_n(x)) = 0.5) = 1.$$

Proof. The proof relies heavily on properties of the logistic map proved by Berliner (Berliner, 1992). From it, we know that as the logistic map is chaotic, each sequence $(f_n(x))_n$ is either eventually periodic or is dense in $[0, 1]$. We also know that as X follows the arcsine distribution, the $C_n(X)$ are independent identically distributed Bernoulli random variables, such that $\mathbb{P}_{x\in X}(C_n(x) = 1) = 0.5$.

Suppose $\mathcal{A}$ calls $C_n(x)$ for values of $x \in \{x_1, \ldots, x_k\}$. We call n^0 the value such that $\mathcal{A}(C_n) = C_{n^0}$. As $\mathbb{P}_{x\in X}(C_{n^0}(x) = 1) = 0.5$ does not depend on n^0, and the $C_n(X)$ are i.i.d., we have $\mathbb{P}_{x\in X}(C_{n^0}(x) \neq C_n(x)) = 0.5$ iff $n^0 \neq n$ and $\mathbb{P}_{x\in X}(C_{n^0}(x) \neq C_n(x)) = 0$ otherwise. The theorem is thus the same as saying that $\mathcal{A}$ almost certainly (in the probabilistic sense) cannot match n^0 to the exact value of n. We now prove that there almost always exists $n^1 \neq n$ which is indistinguishable from n by $\mathcal{A}$, i.e. $C_{n^1}(x_1) = C_n(x_1), \ldots, C_{n^1}(x_k) = C_n(x_k)$.

Let us call $Y_i^1 = C_i(x_1), \ldots, Y_i^k = C_i(x_l)$ with $i \in \mathbb{N}$. Some of the sequences Y^j are periodic after some rank, and some are not. Without loss of generality, we assume $Y^1, \ldots, Y^{k_1}$ are periodic, and $Y^{k_1+1}, \ldots, Y^k$ are not. Notice that, almost certainly, $(Y^1)_{i\geqslant n}$, $\ldots$, $(Y^{k_1})_{i\geqslant n}$ are periodic (for n large enough). Using $P \in \mathbb{N}$ to denote the smaller common multiple of the periods of those sequences, we also note that $C_{n+Pi}(x^1) = C_n(x^1)$, $\ldots$, $C_{n+Pi}(x^{k_1}) = C_n(x^{k_1})$, and that $y_i = n + Pi$.

As the sequences $Y^{k_1+1}, \ldots, Y^k$ are not periodic, we know that each sequence

$$(f_n(x^{k_1+1}))_n, \ldots, (f_n(x^k))_n$$

is dense in $[0, 1]$. Consequently, for any sequence of $k - k_1$ bits, there exists a countable number of $n^1 \in (Y_i)_{i\in\mathbb{N}}$ such that it is equal to $Y^{k_1+1}, \ldots, Y^k$. In particular, if this sequence is $C_n(x^{k_1+1}), \ldots, C_n(x^k)$, any of those n^1 different from n proves the theorem. □

Thm. 3 is a theoretical result: no practical application is ever likely to require the learning this type of concept class. Chaotic behaviour, however, is exhibited by many practical algorithms, so that results *of this type* (rather than *this result*) might have practical relevance. On the other hand, a key part of the proof is allowing the concept class to be infinite and indexed by an unbounded natural number. This is unlikely to happen in industrial settings. Another difficulty is representing and computing with real numbers, which can be done using Real RAM machines (Blum et al., 1989). In the case of the logistic map with parameter $a = 4$, the task is made slightly easier by the existence of an exact solution, but other chaotic maps would require expensive recursive computations. The existence of such extreme cases of unlearnability is nevertheless something to be careful of: unlikely does not mean impossible.

5. OPERATIONAL SEMANTICS APPLICATION

While a non-constructive proof of the Turing-completeness of BR programs is trivial, our constructive proof shows how easy it is to transition from BR programs to a more standard programming language. Indeed, we only need to construct a canonical WHILE form of BR programs to apply known operational semantics methods to BR programs, such as the standard Structural Operational Semantics (SOS) described by Plotkin in (Plotkin, 1981).

Current operational semantics for BR mostly follow the structural approach introduced

in that work in that they use small-step semantics. The standard semantics defined in W3C's Production Rule Dialect of the Rule Interchange Format (RIF-PRD) (de Sainte Marie et al., 2013) is mostly used to specify different variants of BR interpreters. Other attempts at creating operational semantics for BRs have focused on being compatible with either declarative semantics or ontologies (or both) (Rezk and Kifer, 2012; Zaniolo, 1994). Such semantics keep the structure of the BR program divided into rules, which helps with making sense of complex BR programs and creating better user experiences. On the other hand, existing BR semantics are not suitable for proofs.

We can use WHILE programs to obtain a SOS interpretation that is not unique to a syntax or execution algorithm. As shown in the rest of this section, using this semantics can help in proving properties of rule programs such as termination (Cousot, 1981).

5.1. WHILE form of BR programs

A canonical WHILE program equivalent to a BR program is easy to establish using the execution algorithm we consider. The main idea is to introduce an additional integer variable x_0 to serve as a control variable, and to write each rule instance explicitly in the WHILE program itself. The Fig. 5 shows the WHILE form of the example in Fig. 1. Note that our construction provides a "reverse translator" to the one used in Sect. 3.3.

For an ordered set of BRs $R_1, \dots, R_m$ with conditions $T_1, \dots, T_m$ and actions $A_1, \dots, A_m$, and a set of variables $x_1, \dots, x_n$, the WHILE program equivalent to the BR program with the above-mentioned execution mode is written as in Fig. 3. It uses a single additional integer variable x_0. Line 4 in the pseudocode of Fig. 3 consists of a sequence of m **ifblock**s, which

```
1: x0 ← −1
2: while x0 ≠ 0 do
3:     if x0 = −1 then
4:         if...then instructions s.t. x0 is the smallest i
               such that Ti(x1,...,xn) = True, or 0 if there is none
5:     else if x0 = i then
6:         (x1,...,xn) ← Ai(x1,...,xn)
7:         x0 ← −1
8:     else
9:         x0 ← 0
10:    end if
11: end while
```

FIGURE 3. Universal WHILE translation of BR programs.

we have separated for the sake of readability, see Fig. 4.

```
1: if T1(x1,...,xn) = True then
2:     x0 ← 1
3: else if T2(x1,...,xn) = True then
4:     ...
5: else if Tm(x1,...,xn) = True then
6:     x0 ← m
7: else
8:     x0 ← 0
9: end if
```

FIGURE 4. Code corresponding to line 4 in Fig. 3.

An interesting side note is that other BR program interpreters do not forbid the canonical conversion of a BR program to a WHILE program. While the specifics depend on each

The typed Variables (in order)
int x ← 1
int age ← 90

The WHILE program

```
1: x0 ← −1
2: while x0 ≠ 0 do
3:     if x0 = −1 then
4:         if age ⩾ 1∧x= 2 then
5:             x0 ← 1
6:         else if x = 1 then
7:             x0 ← 2
8:         else if age ⩾ 60 then
9:             x0 ← 3
10:        else
11:            x0 ← 0
12:        end if
13:    else if x0 = 1 then
14:        age ← 0
15:        x ← 0
16:        x0 ← −1
17:    else if x0 = 2 then
18:        x ←x+1
19:        x0 ← −1
20:    else if x0 = 3 then
21:        age ←age +1
22:        x0 ← −1
23:    else
24:        x0 ← 0
25:    end if
26: end while
```

FIGURE 5. Example: WHILE form of the BR program in Fig. 1

execution algorithm, we can remember the three most common elements found in such algorithms from Section 2.2:

- *Refraction* results in the use of an additional boolean variable per rule instance in the WHILE program.
- *Priority* partially decides the order of the if-then-else of the WHILE program.
- *Recency* is the most complicated. An easy workaround would add an incremental integer variable per rule instance in the WHILE program and use a max() function in the tests of the if-then-else.

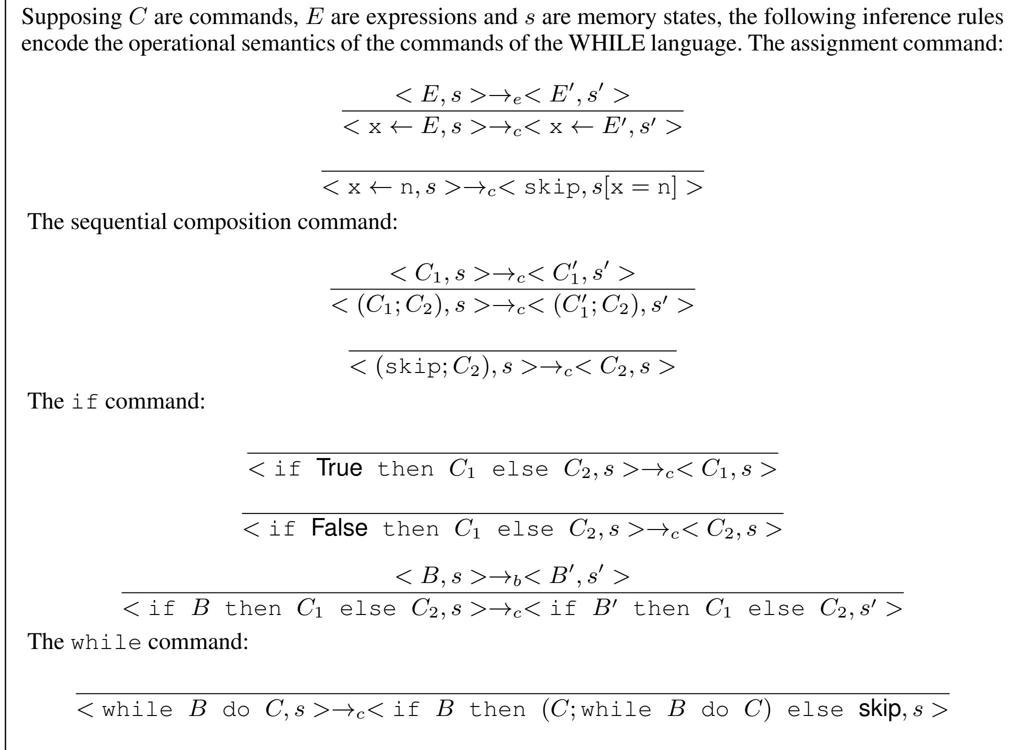
5.2. A structural operational semantics

Let us consider the SOS described by Plotkin in (Plotkin, 1981) applied to WHILE programs. It describes the execution of a program as a finite automaton, with each simple command corresponding to a state transition relation. Its usefulness in proving properties of programs, such as sets of input resulting in termination or non-termination, has been established (Burstall, 1969). Fig. 6 shows the state transition relations corresponding to the WHILE language under an easily readable form, using an Inference Rule syntax wherein the premises are listed above a horizontal line, the conclusion below, and the condition, if present, to its right.

We use an example to demonstrate that such a transformation of a BR program into a WHILE program can be used to prove properties of BR programs through those same SOS methods. Consider the simple example of a loan request application with three integer variables: amount, duration and income. We use the naive rule described in Fig. 7. The input value of amount is the requested loan value, while its final value is the total repaid sum.

We provide a proof-of-concept implementation of our SOS, and perform a few computational experiments on the issue of termination of BR programs. Obviously, since we now know that the BR language is Turing-complete, we do not mean that our SOS is able to *systematically* predict termination correctly. We simply mean to say that we test our SOS on a few *given* BR programs.

We use Prolog (Clocksin and Mellish, 1987) to code the SOS of the WHILE programming language as rules in a rule inference engine using the form displayed in Fig.6. Similar rules encode the evaluation of Boolean and Numerical variables. We then encode the WHILE



Supposing C are commands, E are expressions and s are memory states, the following inference rules encode the operational semantics of the commands of the WHILE language. The assignment command:

$$\frac{< E, s > \to_e < E', s' >}{< \mathtt{x} \leftarrow E, s > \to_c < \mathtt{x} \leftarrow E', s' >}$$

$$\frac{}{< \mathtt{x} \leftarrow \mathtt{n}, s > \to_c < \mathtt{skip}, s[\mathtt{x} = \mathtt{n}] >}$$

The sequential composition command:

$$\frac{< C_1, s > \to_c < C_1', s' >}{< (C_1; C_2), s > \to_c < (C_1'; C_2), s' >}$$

$$\frac{}{< (\mathtt{skip}; C_2), s > \to_c < C_2, s >}$$

The `if` command:

$$\frac{}{< \mathtt{if}\ \mathsf{True}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < C_1, s >}$$

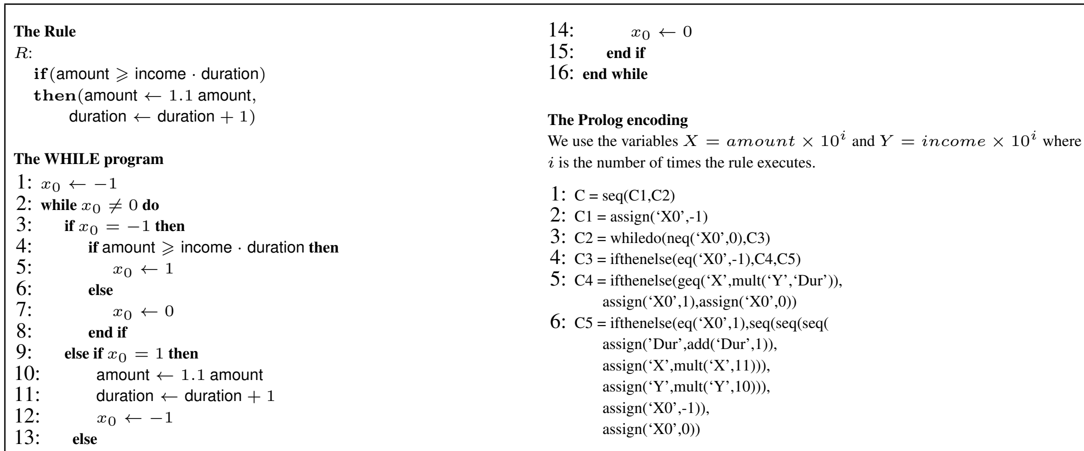
$$\frac{}{< \mathtt{if}\ \mathsf{False}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < C_2, s >}$$

$$\frac{< B, s > \to_b < B', s' >}{< \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s > \to_c < \mathtt{if}\ B'\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, s' >}$$

The `while` command:

$$\frac{}{< \mathtt{while}\ B\ \mathtt{do}\ C, s > \to_c < \mathtt{if}\ B\ \mathtt{then}\ (C; \mathtt{while}\ B\ \mathtt{do}\ C)\ \mathtt{else}\ \mathsf{skip}, s >}$$

FIGURE 6. The inference rules encoding the operational semantics of commands in WHILE, starting from a state s of the machine's memory



The Rule

R:
 if(amount $\geqslant$ income $\cdot$ duration)
 then(amount $\leftarrow$ 1.1 amount,
  duration $\leftarrow$ duration + 1)

The WHILE program

1: $x_0 \leftarrow -1$
2: **while** $x_0 \neq 0$ **do**
3: **if** $x_0 = -1$ **then**
4:   **if** amount $\geqslant$ income $\cdot$ duration **then**
5:    $x_0 \leftarrow 1$
6:   **else**
7:    $x_0 \leftarrow 0$
8:   **end if**
9: **else if** $x_0 = 1$ **then**
10:   amount $\leftarrow$ 1.1 amount
11:   duration $\leftarrow$ duration + 1
12:   $x_0 \leftarrow -1$
13: **else**
14:   $x_0 \leftarrow 0$
15: **end if**
16: **end while**

The Prolog encoding

We use the variables $X = amount \times 10^i$ and $Y = income \times 10^i$ where i is the number of times the rule executes.

```
1: C = seq(C1,C2)
2: C1 = assign('X0',-1)
3: C2 = whiledo(neq('X0',0),C3)
4: C3 = ifthenelse(eq('X0',-1),C4,C5)
5: C4 = ifthenelse(geq('X',mult('Y','Dur')),
       assign('X0',1),assign('X0',0))
6: C5 = ifthenelse(eq('X0',1),seq(seq(seq(
       assign('Dur',add('Dur',1)),
       assign('X',mult('X',11))),
       assign('Y',mult('Y',10))),
       assign('X0',-1)),
       assign('X0',0))
```

FIGURE 7. A naive BR program for Loan Applications

program itself as a series of rules in Prolog[1]. It is then a simple matter to ask Prolog about the termination of this program by simply asking about the truth value of assertions including '$X0_{fin}$' $= 0$ and constraint over the inputs.

[1]Notably, we use the variables $X = Amount \times 10^i$ and $Y = income \times 10^i$ where i is the number of times the rule executes, so as to benefit from SWI-Prolog's CLP(FD) library (Triska, 2014) and XSB-Prolog's `bounds` library.

5.3. Proving Termination

Termination of a program for a given input is one of the easiest properties of programs to look into using SOS. Because of the Turing completeness of the BR language, our methodology for proving (or disproving) termination will obviously not work on every BR program. We proceed as follows: given a BR program, we tranform it into its WHILE form, which we then use to derive the SOS. We implement the SOS as a set of constraints in two Prolog dialects (*SWI-Prolog* (Wielemaker et al., 2012) and *XSB Prolog* (Sagonas et al., 1994)), so as to use the corresponding interpreter to try and establish feasibility or infeasibility of the constraint set.

We coded the WHILE programs for three examples in Prolog. We first used *SWI-Prolog* to test termination for an interval range of inputs by using a Prolog query containing the input and the specific output $x_0 = 0$ (i.e. the WHILE loop ends). The results for the program in Fig. 7 show some of the advantages and limits of using basic Prolog. For some intervals of values for $amount$, we can prove in a few seconds of CPU time that for $duration = 5$ and $income = 10$, the BR program always terminates. On the other hand, we are unable to prove non-termination using *SWI-Prolog* as it lacks loop detection algorithms. We then used the tabling included in *XSB-Prolog* (Swift and Warren, 2012) to test non-termination. We observe that we can now detect non-terminating inputs, however given an interval we can only detect the *absence of a terminating input*, i.e. we cannot distinguish between a set of input that all terminate and another where only some values lead to termination. Furthermore, the results for the program in Fig. 9 show that even then, not all inputs can be determined to be terminating or non-terminating (this is expected as the halting problem is undecidable). We remark, however, that the most useful contribution of an SOS is to prove termination of programs (rather than non-termination), making even the basic Prolog application important. Amongst its many advantages, our semantics can be used for automatic validation of a BR program, for example. A search algorithm could also be used to identify the cutoff point of $amount \leqslant 62$ as containing every terminating input, if one were to take the monotonicity of the program into account.

Along with the example from Fig. 7, a slightly more realistic example is seen in Fig. 8, adding the boolean variable approval and the integer variable age to the BR program. On the other hand, Fig. 9 is a less trivial example: a BR program that chooses the interest rate interest depending on the credit score score using a made-up algorithm which simulates the logistic map, where score $\in [300, 850]$.

The results for the examples in Fig. 7 and Fig. 8 are displayed in Fig. 1, while the results for the example in Fig. 9 are in Fig. 2. Fig. 1 is indicative of a practical use of studying the termination of BR programs this way: a bank might wish to check that their loan application BR program terminates for a given range of values. Studying unexpected failures can point out a missing rule or variable in the BR program.

On the other hand, the results from Fig. 2 are somewhat more difficult to interpret. While the BR program in Fig. 9 is somewhat contrived, it shows that some failures can be not only unexpected, but unpredictable. The SWI-Prolog `ERROR: Out of local stack` looks like yet another inconclusive answer, but tracing the execution of the Prolog interpreter shows that some errors are the result of infinitely repeated goals, and a simple alteration to the Prolog interpreter can detect most infinite recursion loops of that kind (Van Gelder, 1987). This is proved by looking at the result of the tabling-enabled XSB-Prolog execution, which gives us proof of non-termination for the input $r = 3.82843$ (score $= 813.541375$). Such non-terminating inputs might be found by looking at the internal states of a classic BR engine, but the automation provided by existing research on SOS and Prolog remains valuable and time-saving. On the other hand, some values have a convergent but non-periodic behavior which we cannot detect using either of our implementations. The tested input $r = 3$ (score $= 637.5$) has such a behavior. The Prolog algorithm actually simulates the fixed

```
The Rules
R1:
  if (approval = True ∧ age < 18)
  then (approval ← False)
R2:
  if (approval = True ∧ duration > 10)
  then (approval ← False)
R3:
  if (approval = True ∧ amount ⩾ income · duration)
  then (amount ← 1.1 amount,
        duration ← duration + 1)

The Variables
boolean approval = True
int age, duration, income
float amount

The WHILE program
1: x0 ← −1
2: while x0 ≠ 0 do
3:   if x0 = −1 then
4:     if approval = True ∧ age < 18 then
5:       x0 ← 1
6:     else if approval = True ∧ duration > 10 then
7:       x0 ← 2
8:     else if approval = True ∧ amount ⩾ income · duration then
9:       x0 ← 3
10:    else
11:      x0 ← 0
12:    end if
13:  else if x0 = 1 then
14:    approval ← False
15:    x0 ← −1
16:  else if x0 = 2 then
17:    approval ← False
18:    x0 ← −1
19:  else if x0 = 3 then
20:    amount ← 1.1 amount
21:    duration ← duration + 1
22:    x0 ← −1
23:  else
24:    x0 ← 0
25:  end if
26: end while
```

FIGURE 8. A more realistic BR program for Loan Applications, the input should always have approval = True

```
The Rules
R1:
  if (n ⩽ 1000)
  then (x ← r · x(1 − x),
        n ← n + 1)
R2:
  if (x ⩾ 0.51)
  then (x ← r · x(1 − x))
R3:
  if (x ⩽ 0.49)
  then (x ← r · x(1 − x))

The Variables
float r = 4 score/850 ∈ [0, 4]
float x = 0.48 + interest ∈ [0, 1]
int n = 1

The WHILE program
1: x0 ← −1
2: while x0 ≠ 0 do
3:   if x0 = −1 then
4:     if n ⩽ 1000 then
5:       x0 ← 1
6:     else if x ⩾ 0.51 then
7:       x0 ← 2
8:     else if x ⩽ 0.49 then
9:       x0 ← 3
10:    else
11:      x0 ← 0
12:    end if
13:  else if x0 = 1 then
14:    x ← r · x(1 − x)
15:    n ← n + 1
16:    x0 ← −1
17:  else if x0 = 2 then
18:    x ← r · x(1 − x)
19:    x0 ← −1
20:  else if x0 = 3 then
21:    x ← r · x(1 − x)
22:    x0 ← −1
23:  else
24:    x0 ← 0
25:  end if
26: end while
```

FIGURE 9. A nontrivial BR program and corresponding WHILE program, the input should always have $n = 1$

points (after 1000 iterations) of the logistic map, defined as the sequence $x_{n+1} = r \cdot x_n(1 - x_n)$, and falls into a infinite recursion of goals if no fixed point is within $[0.49, 0.51]$. The bifurcation diagram in Fig. 10 is well-known, and represents those fixed points. While it makes the situation clear for $r \in [1.41, 3.57]$, i.e. score$\in [300, 758]$, the higher values fall within the chaotic part of the logistic map. In particular, some specific values have a periodic behavior, such as the tested $r = 3.82843$ (score $= 813.541375$). The use of SOS derived from the WHILE form of BR programs helps in this and similar cases by automatically identifying with certainty some non-terminating inputs, assuming Prolog is configured to identify infinite recursive goals.

BR program: interval domain for variable amount	$]-\infty, 50]$	$[50, 60]$	$[60, 70]$	$]-\infty, 62]$	$\{62\}$	$\{63\}$	$[63, +\infty[$
Naive program: SWI-Prolog (Fig. 7)	True	True	`ERROR: 'Out of global stack'`	True	True	`ERROR:'Out of global stack'`	`ERROR:'Out of global stack'`
Naive program: XSB-Prolog (Fig. 7)	yes	yes	yes	yes	yes	no	no
Realistic program: SWI-Prolog (Fig. 8)	True	True	True	True	True	True	True

TABLE 1. Results of the SWI-Prolog and XSB-Prolog queries containing the SOS of WHILE forms of the BR programs in Fig. 7 and Fig. 8 as well as facts about the input (duration $= 5$, income $= 10$, and age $= 20$ when relevant) and about the output ($x_0 = 0$)

BR program	r = 3	r = 3.22	r = 3.67	r = 3.82843
	score $= 637.5$	score $= 684.25$	score $= 779.875$	score $= 813.541375$
Custom interest rate: SWI-Prolog (Fig. 9)	`ERROR: Out of local stack`	True	True	`ERROR: Out of local stack`
Custom interest rate: XSB-Prolog (Fig. 9)	`Error: Query exhausted system memory`	yes	yes	no

TABLE 2. Results of the Prolog query containing the SOS of WHILE forms of the BR programs in Fig. 9 as well as facts about the input ($x = 5$, $n = 1$) and about the output ($x_0 = 0$)

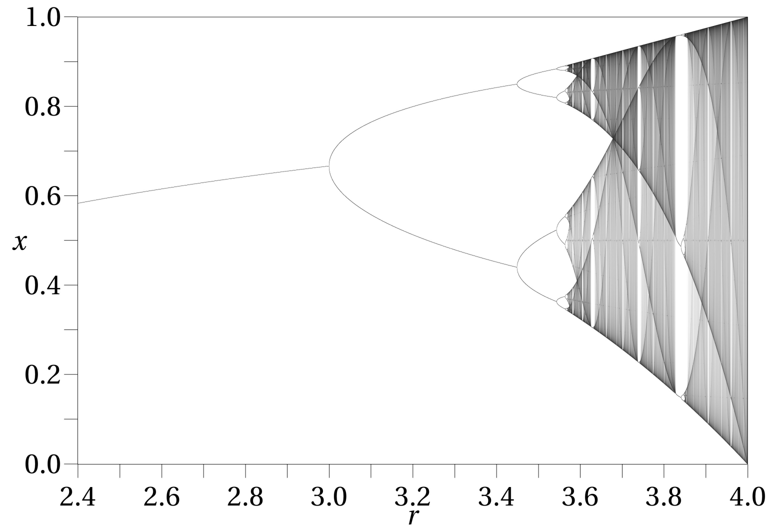


FIGURE 10. Bifurcation Diagram for the Logistic Map

6. CONCLUSION

Business Rules form an unexpectedly expressive programming language compared to their practical use in business. While BR programs are syntactically very simple, the language contains hidden complexity in the details of its interpreters. In particular, any interpreter that uses a looping algorithm can make the set of all BR programs Turing-complete. We hope that our formalization will open the doors to further research into BR as a program-

ming language (Wang, 2017). The Turing-completeness of BR programs using any looping interpreter is important to theoretical research into the properties of BRs.

We also proved the PAC-unlearnability of the concept class of BR programs. That proof is stronger than the usual PAC-learnability approach as the example of the logistic map shows that some BR programs are completely PAC-unlearnable regardless of how inefficient the learning algorithm is. This showcases how complex and nontrivial BR programs are. This invites future research into learning BR programs with statistical objectives as it may well be possible to achieve our stated purpose for less general, but still useful classes of BR programs. That is a problem that industrial BR systems would be very interested in. In (Wang and Liberti, 2017) we look at discrete distributions, for example, but many other possibilities exist.

The theoretical limits discussed in this paper do not imply that the BRAG problem is hopeless, but only that more research is needed in order to delimit a reasonable class of BRAG instances for which the BRAG can be solved, see e.g. (Wang et al., 2017).

The constructive proof we used to prove the Turing-completeness of BRs allows us to introduce the WHILE forms of BR programs. WHILE programs are simple programming languages that are easily linked to others, and can already provide some insight on their own. Using SOS analysis techniques on the transformed BR programs highlights a marked difference with the existing operational semantics of BRs. These might inspire new techniques for studying the properties of BR programs. In particular, the study of which inputs or sets of input allow for the termination or non-termination of BR programs is made more practical by the use of this standardized operational semantics.

Acknowledgments

The first author (OW) is supported by an IBM France/ANRT CIFRE Ph.D. thesis award.

REFERENCES

APT, K. 2003. Principles of Constraint Programming. Cambridge University Press, Cambridge.

BERLINER, L.M. 1992. Statistics, probability and chaos. Statistical Science, **7**(1):69–122.

BLOCKEEL, H., and L. DE RAEDT. 1998. Top-down induction of first-order logical decision trees. Artificial Intelligence, **101**(1):285–297.

BLUM, L., M. SHUB, and S. SMALE. 1989. On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions, and universal machines. Bulletin of the American Mathematical Society, **21**(1):1–46.

BRACHMAN, R., and H. LEVESQUE. 2004. Knowledge Representation and Reasoning. Elsevier, Amsterdam.

BURSTALL, R. M. 1969. Proving properties of programs by structural induction. The Computer Journal, **12**(1):41–48.

BUSINESS RULES GROUP. 2017. The business rules manifesto <www.businessrulesgroup.org/brmanifesto.htm>.

CHURCH, A. 1932. A set of postulates for the foundation of logic. Annals of Mathematics, **33**(2):346–366.

CHURCH, A. 1936. An unsolvable problem of elementary number theory. American Journal of Mathematics, **58**:345–363.

CLOCKSIN, W.F., and C.S. MELLISH. 1987. Programming in Prolog. Springer-Verlag, New York.

COHEN, A., S. GOLDWASSER, and V. VAIKUNTANATHAN. 2015. Aggregate pseudorandom functions and connections to learning. *In* Theory of Cryptography (TCC). *Edited by* Y. Dodis and J. Nielsen, Volume 9015 of *LNCS*. Springer, Berlin, pp. 61–89.

COUSOT, P. 1981. Semantic foundations of program analysis. *In* Program Flow Analysis: Theory and Applications. *Edited by* S. Muchnick and N. Jones. Prentice-Hall, Inc., pp. 303–342. Englewood Cliffs, New Jersey.

CULBERT, C., and G. RILEY. 2003. CLIPS Basic Programming Guide. CLIPS.

CURTIS, M.W. 1965. A Turing Machine simulator. Journal of the ACM, **12**(1):1–13.

DE RAEDT, L., and S. DŽEROSKI. 1994. First-order JK-clausal theories are PAC-learnable. Artificial Intelligence, **104**(1):375–392.

DE SAINTE MARIE, C., G. HALLMARK, and A. PASCHKE. 2013. Rif production rule dialect (second edition). Recommendation, W3C.

FORGY, C. 1982. Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence, **19**(1):17–37.

FRIEDMAN-HILL, E.J. 2003. JESS in Action. Manning Publications, Greenwitch, CT.

GANDY, R. 1980. Church's thesis and the principles for mechanisms. *In* The Kleene Symposium. *Edited by* J. Barwise, H. Keisler, and K. Kunen. North-Holland, pp. 123–148.

GIURCA, A., D. GAŠEVIĆ, and K. TAVETER editors. 2009. Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches. IGI Global, Hershey.

GOLDREICH, O., S. GOLDWASSER, and S. MICALI. 1986. How to construct random functions. Journal of the ACM, **4**(33):792–807.

HANSON, ERIC N., and MOHAMMED S. HASAN. 1993. Gator: An optimized discrimination network for active database rule condition testing. Technical Report 93-036, CIS Department University of Florida.

HAREL, D. 1980. On folk theorems. Communications of the ACM, **23**(7):379–389.

HIROSE, K., and M. OYA. 1972. Some results in general theory of flow charts. *In* Proceedings of the First USA-Japan Computer Conference, Tokyo, Japan, pp. 367–371.

HOARE, C.A.R. 1969. An axiomatic basis for computer programming. Communications of the ACM, **12**(10):576–580.

LIBERTI, L., and F. MARINELLI. 2014. Mathematical programming: Turing completeness and applications to software analysis. Journal of Combinatorial Optimization, **28**(1):82–104.

LIGEZA, A. 2006. Logical Foundations for Rule-Based Systems. Springer, Berlin.

MATIYASEVICH, Y. 1993. Hilbert's Tenth Problem. MIT Press, Boston.

MINSKY, M. 1972. Computation: Finite and Infinite Machines. Prentice-Hall, London.

PLOTKIN, G. 1981. A structural approach to operational semantics. Comp. Sci., Aarhus Univ. DAIMI FN-19. Technical Report Comp. Sci., Aarhus Univ. DAIMI FN-19, Computer Science Department, Aarhus University.

PROCTOR, M. 2011. Drools: a rule engine for complex event processing. *In* AGTIVE International Conference on Applications of Graph Transformations with Industrial Relevance. *Edited by* A. Schürr, D. Varró, and G. Varró. Springer, Berlin, pp. 2–2.

REZK, M., and M. KIFER. 2012. Formalizing production systems with rule-based ontologies. *In* Foundations of Information and Knowledge Systems. *Edited by* T. Lukasiewicz and A. Sali, Volume 7153 of *LNCS*. Springer, Berlin, pp. 332–351.

ROSS, R. 2003. Principles of the Business Rule Approach. Addison-Wesley, Boston.

SAGONAS, K., T. SWIFT, and D.S. WARREN. 1994. XSB as an efficient deductive database engine. *In* SIGMOD International Conference on the Management of Data. *Edited by* R. Snodgrass and M. Winslett. ACM Press, New York, pp. 442–453.

SHANNON, C. 1956. A universal Turing machine with two internal states. *In* Automata Studies. *Edited by* C. Shannon and J. McCarthy, Volume 34 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, pp. 157–165.

SNEYERS, J., T. SCHRIJVERS, and B. DEMOEN. 2005. The computational power and complexity of constraint handling rules. *In* Proceedings of the 2nd Workshop on Constraint Handling Rules, pp. 3–17.

SWIFT, T., and D.S. WARREN. 2012. XSB: Extending prolog with tabled logic programming. Theory and Practice of Logic Programming, **12**(1-2):157–187.

TRISKA, M. 2014. Correctness Considerations in CLP(FD) Systems. Ph. D. thesis, Vienna University of Technology.

TURING, A. 1937. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, **42**(1):230–265.

TURING, A. 1939. Systems of Logic Based on Ordinals. Ph. D. thesis, Princeton University.

VALIANT, L.G. 1984. A theory of the learnable. Communications of the ACM, **11**(27):1134–1142.

VAN GELDER, A. 1987. Efficient loop detection in prolog using the tortoise-and-hare technique. Journal of Logic Programming, **4**(1):23–31.

VON HALLE, B. 2001. Business Rules Applied: Building Better Systems Using the Business Rules Approach. Wiley, Chichester.

WANG, O. 2017. Analytics learning for rule-based systems. Ph. D. thesis, Ecole Polytechnique.

WANG, O., C. KE, L. LIBERTI, and C. DE SAINTE MARIE. 2016. The learnability of business rules. *In* Machine Learning, Optimization, and Big Data. *Edited by* P. Pardalos, P. Conca, G. Giuffrida, and G. Nicosia, Volume 10122 of *LNCS*. Springer, pp. 257–268.

WANG, O., and L. LIBERTI. 2017. Controlling some statistical properties of business rules programs. *In* Learning and Intelligent Optimization (LION), LNCS, Springer, Berlin.

WANG, O., L. LIBERTI, C. D'AMBROSIO, C. DE SAINTE MARIE, and C. KE. 2017. Controlling the average behaviour of business rules programs. *In* RuleML. *Edited by* J. A. et al., Volume 9718 of *LNCS*. Springer, Berlin, pp. 83–96.

WIELEMAKER, J., T. SCHRIJVERS, M. TRISKA, and T. LAGER. 2012. SWI-Prolog. Theory and Practice of Logic Programming, **12**(1-2):67–96.

WILLIAMS, H.P. 1999. Model Building in Mathematical Programming (4th ed.). Wiley, Chichester.

ZANIOLO, C. 1994. A unified semantics for active and deductive databases. *In* Rules in Database Systems. *Edited by* N. P. et al.. Springer, London, pp. 271–287.