Business Rule Sets as Programs: Turing-completeness and Structural Operational Semantics

Olivier Wang[1,2] Changhai Ke[1] Leo Liberti[2] Christian de Sainte Marie[1]

[1] IBM France, 9 Rue de Verdun, 94250 Gentilly, France
[2] CNRS LIX, Ecole Polytechnique, 91128 Palaiseau, France

olivier.wang@polytechnique.edu

Abstract

Production Rules or Business Rules are of the form if condition then action. *Sets of such rules can be executed on input data according to execution algorithms, or analyzed semantically using operational semantics, but the universality of systems or sets of such rules has never been characterized formally, to the best of our knowledge.*
Our goal is to show that for a simple execution algorithm, sets of Business Rule form a universal programming language. In other words, we show that Business Rules are Turing-complete. The proof consists in showing that a Business Rule program can be transformed in a WHILE program using additional variables. The implications of such a WHILE form include a type of structural operational semantics which may facilitate semantic analysis of Business Rule Sets.

Keywords

Business Rules, Production Rules, Turing-completeness, Structural Operational Semantics, WHILE program.

1 Introduction

In recent years, academic research on Rule Systems has mostly focused on knowledge representation and learning, through Classification Rules for example. In contrast, we look at Rule Systems as programs. In particular, Business Rules (BR) (also called Production Rules in RuleML [1]) have a behavior much closer to standard imperative programs (flowcharts, WHILE programs) than any other type of rules. They are notably used in decision automation through Business Rules Management Systems (BRMS). Where Induction Rules can be used to prove the internal consistency of an ontology and deduce theorems from axioms, Business Rules behave more like WHILE programs, when executed via an appropriate execution algorithm.
The rules we consider in this paper are BRs. BRs are written using meta-variables stored in a different symbol table than the input variables, which we will keep calling variables. The meta-variable table matches variables appearing in a BR to variable names, and the variable table matches the latter to stored values. Interpretation is in two stages: first, the BR program is turned into a set of *rule instances*, where the meta-variables are replaced by corresponding variables; then, rule instances are run by an execution algorithm, which includes a conflict resolution method to decide the order of rule instance execution. We remark that BR is a typed language. Every meta-variable and input variable must have a type, and arbitrary new types can be defined within the BR program itself. The matching from meta-variables to variables is typed: each meta-variable has a type and can only match to a variable of the same type. We only consider execution algorithms which include a main loop.
BR programming can be seen as programming for non-programmers, in the following sense. The two most difficult concepts for a layperson to understand are loops and function calls. BR disposes of the former by automatically executing programs over a loop construct, and of the latter by resorting to meta-substitutions (meta-variables are replaced by variable names), so that a rule *if* condition *then* action is automatically matched to many conditions and actions sharing the same structure.
There are many variations of BR execution algorithms. Our goal is not to characterize them or compare them. We use a very simple algorithm consisting of the following steps:

1. Match variables to type-appropriate meta-variables in rules to create all possible rule instances
2. Select the rule instances for which the condition is true, using the current values of the variables
3. Execute the action of the first rule instance in the current selection or stop if there is no such rule instance
4. Restart from step 1.

The order of rule instances used in step 3 is in our algorithm defined by the lexicographic order derived from a predefined order on the rules in the rule set to execute and a predefined order on input variables. Such an 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.

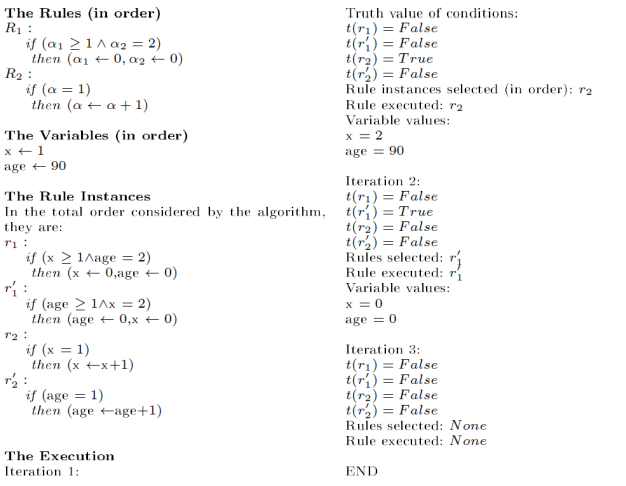

Figure 1: Example illustrating the execution algorithm

2 Turing-completeness

A Universal Turing Machine (UTM) is a Turing Machine (TM) which can simulate any other TM on arbitrary input [10, 11]. Let L be a programming language for the UTM U, described for example by its grammar. By means of a special program $\mathscr{I}$ called *interpreter*, programs written in L can be executed on U [6].

Definition 1. *If a programming language L can be used to program a UTM via an interpreter, then L is Turing-complete.*

We can replace "UTM" in Defn. 1 by any universal computer described in any Turing-complete language L', since interpreters can be composed. More precisely: (i) let U' be a program in L' describing a UTM, and $\mathscr{I}'$ is the interpreter from L' to the UTM; (ii) let U be a program in L describing $\mathscr{I}'(U')$, and $\mathscr{I}$ be an interpreter from L to L'. Then $\mathscr{I}(U) = \mathscr{I}'(U')$ is a UTM. Moreover, since a UTM is defined as a TM which is able to simulate any other TM, we can prove L Turing-complete by showing that for any TM, L can be used to describe that TM via its interpreter, as was done in [3]. According to the Church-Turing thesis [2, 12, 4], any effectively calculable function is Turing-computable. In other words, no device or program can compute a function that a UTM cannot.

2.1 Business Rule programs

Regardless of the execution algorithm, a BR program consists in a set of type declarations and a set of rules. A type declaration consists of either the creation of a type ($create\_new\_type$(type)) or the assignment of a type to a variable ($type$(var) $\leftarrow$ type), where var can be either a variable or a meta-variable. A rule is defined as follows. Given α the typed meta-variables and x the typed variables, a rule is written:

if $T(\alpha, x)$ **then**
$\alpha \leftarrow A(\alpha, x)$
$x \leftarrow B(\alpha, x)$
end if

where T is the condition and the couple (A, B) describes the action. At least one of the variables, say x_1 without loss of generality, is selected to be the *output* of the BR program.

The first part of a BR program interpreter is to compile the rule instances derived from each rule. At compile time, α is replaced by every type-feasible reordering of the x input variable vector. For $x \in \mathbb{R}^n$, the explicit set of rule instances compiled from this rule is the type-feasible part of the following code fragments, using $(\sigma_j \mid j \in \{1, ..., n!\})$ the permutations of $\{1, ..., n\}$:

if $T((x_{\sigma_1(1)}, ..., x_{\sigma_1(n)}), x)$ **then**
$(x_{\sigma_1(1)}, ..., x_{\sigma_1(n)}) \leftarrow A((x_{\sigma_1(1)}, ..., x_{\sigma_1(n)}), x)$
$(x_1, ..., x_n) \leftarrow B((x_{\sigma_1(1)}, ..., x_{\sigma_1(n)}), x)$
end if
...
if $T((x_{\sigma_n(1)}, ..., x_{\sigma_n(n)}), x)$ **then**
$(x_{\sigma_n(1)}, ..., x_{\sigma_n(n)}) \leftarrow A((x_{\sigma_n(1)}, ..., x_{\sigma_n(n)}), x)$
$(x_1, ..., x_n) \leftarrow B((x_{\sigma_n(1)}, ..., x_{\sigma_n(n)}), x)$
end if

The size of this set varies. The typing of the variables and meta-variables matters, and depending on T, A and B some of these operations might also be computationally equivalent. The number of rule instances compiled from a given rule can be 0 for an invalid rule ($T(\alpha, x) = false$), 1 for a static rule ($T(\alpha, x)$ and $B(\alpha, x)$ do not vary with α, $A(\alpha, x) = \alpha$), and up to $n!$ for some rules if every variable has the same typing.

An interpreter $\mathscr{I}$ for a BR program takes the Business Rules and values of the variables as input, and executes valid rule instances one at a time according to a conflict resolution strategy (this ranges from simple to complex algorithms) until it either executes a $Stop$ instruction or runs out of valid rule instances. It then returns the value of x_1. The most basic interpreter $\mathscr{I}_0$ uses the order given earlier to choose which rule instance to execute, executing assignment actions whenever conditions are True. In order to show Turing-completeness, we only consider the basic interpreter, meaning that we expect any more complicated ones to be able to simulate the most basic.

2.2 WHILE programs

A WHILE program has the canonical (recursive) form:

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

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

or as an empty command. The interpretation of the symbols $\mathsf{T}^i_k(x)$ and $\mathsf{A}^i_k(x)$ is: T are tensors of Boolean conditions on the variables x, which evaluate to True or False, and A is a 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 embedded test conditions followed by a conditional assignment action (note the action could be empty by setting the corresponding A function to the identity).

It is well known that WHILE programs are Turing-complete [5].

2.3 WHILE form of rule sets

A canonical WHILE program equivalent to a BR program is easy to establish using the execution algorithm we consider. Although this reduction plays no role in proving Turing-completeness of BR programs, it is still interesting to remark that BR programs *can* be written as WHILE programs, and having a standardized WHILE form for BR programs has some interesting applications. 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. This allows for an easier adaptation to different interpreters, such as those used in commercial BRMS. It must be noted that as explained in 2.1, each rule corresponds to a sequence of rule instances. The order of the instances in that sequence is determined by our algorithm: it is the order described in 2.1. The formal definition is somewhat more complicated, of course.

For a set of Business Rules $R_1, ..., R_m$ with conditions $T_1, ..., T_m$ and actions $(A_1, B_1), ..., (A_m, B_m)$, a set of typed variables $x_1, ..., x_n$, and a set of typed meta-variables $\alpha_1, ..., \alpha_n$, the WHILE program equivalent to the BR program with the above-mentioned execution mode is written as below. It uses a single additional integer variable x_0, and we note $(\sigma_j \mid j \in \{1, ..., n!\})$ the permutations of $\{1, ..., n\}$. The σ_j are ordered in lexicographic order on $\sigma_j(1, ..., n)$.

1: $x_0 \leftarrow -1$
2: **while** $x_0 \neq 0$ **do**
3: **if** $x_0 = -1$ **then**
4: $x_0 \leftarrow \min\{(i-1) \times n! + j$
5: $\mid T_i(x_{\sigma_j(1)}, ..., x_{\sigma_j(n)}) = true\}$
6: **else if** $x_0 = (i-1) \times n! + j$ **then**
7: $(x_{\sigma_j(1)}, ..., x_{\sigma_j(n)}) \leftarrow$
8: $A_i((x_{\sigma_j(1)}, ..., x_{\sigma_j(n)}), (x_1, ..., x_n))$
9: $(x_1, ..., x_n) \leftarrow$
10: $B_i((x_{\sigma_j(1)}, ..., x_{\sigma_j(n)}), (x_1, ..., x_n))$
11: $\triangleright$ x_0 is unique for each (i,j) couple
12: $x_0 \leftarrow -1$
13: **else**
14: $x_0 \leftarrow 0$
15: **end if**
16: **end while**

The use of the min() function and of the conditioned set is not strictly speaking allowed in WHILE programs, and the expression $T_i(x_{\sigma_j(1)}, ..., x_{\sigma_j(n)}) = true$ should be closer to checking that the substitution $\alpha_i \leftarrow x_{\sigma_j(i)}$ is type appropriate AND that the test is true. Even so, line 4 and line 5 could easily be replaced by series of try...catch... and if...then... instructions. For example, line 4 would be replaced by:

if $type(\alpha_1) = type(x_{\sigma_1(1)}) \wedge \cdots \wedge$
 $type(\alpha_n) = type(x_{\sigma_1(n)}) \wedge$
 $T_1((x_{\sigma_1(1)}, ..., x_{\sigma_1(n)}), (x_1, ..., x_n)) = true$ **then**
  $x_0 \leftarrow 1$
else if $type(\alpha_1) = type(x_{\sigma_2(1)}) \wedge \cdots \wedge$
 $type(\alpha_n) = type(x_{\sigma_2(n)}) \wedge$
 $T_1((x_{\sigma_2(1)}, ..., x_{\sigma_2(n)}), (x_1, ..., x_n)) = true$ **then**
  ...
else if $type(\alpha_1) = type(x_{\sigma_{n!}(1)}) \wedge \cdots \wedge$
 $type(\alpha_n) = type(x_{\sigma_{n!}(n)}) \wedge$
 $T_1((x_{\sigma_{n!}(1)}, ..., x_{\sigma_{n!}(n)}), (x_1, ..., x_n)) = true$ **then**
  $x_0 \leftarrow n!$
else if ... **then**
  ...
else if $type(\alpha_1) = type(x_{\sigma_{n!}(1)}) \wedge \cdots \wedge$
 $type(\alpha_n) = type(x_{\sigma_{n!}(n)}) \wedge$
 $T_n((x_{\sigma_{n!}(1)}, ..., x_{\sigma_{n!}(n)}), (x_1, ..., x_n)) = true$ **then**
  $x_0 \leftarrow n \times n!$
end if

The typed Variables (in order)
int x $\leftarrow$ 1
int age $\leftarrow$ 90

The WHILE program

1: $x_0 \leftarrow -1$
2: **while** $x_0 \neq 0$ **do**
3: **if** $x_0 = -1$ **then**
4: **if** x $\geq$ 1$\wedge$age = 2 **then**
5: $x_0 \leftarrow 1$ $\triangleright$ (i,j) = (1,1)
6: **else if** age $\geq$ 1$\wedge$x= 2 **then**
7: $x_0 \leftarrow 2$ $\triangleright$ (i,j) = (1,2)
8: **else if** x = 1 **then**
9: $x_0 \leftarrow 3$ $\triangleright$ (i,j) = (2,1)
10: **else if** age = 1 **then**
11: $x_0 \leftarrow 4$ $\triangleright$ (i,j) = (2,2)
12: **end if**
13: **else if** $x_0 = 1$ **then**
14: x $\leftarrow$ 0
15: age $\leftarrow$ 0
16: $x_0 \leftarrow -1$
17: **else if** $x_0 = 2$ **then**
18: age $\leftarrow$ 0
19: x $\leftarrow$ 0
20: $x_0 \leftarrow -1$
21: **else if** $x_0 = 3$ **then**
22: x $\leftarrow$x+1
23: $x_0 \leftarrow -1$
24: **else if** $x_0 = 4$ **then**
25: age $\leftarrow$age +1
26: $x_0 \leftarrow -1$
27: **else**
28: $x_0 \leftarrow 0$
29: **end if**
30: **end while**

Figure 2: WHILE form of the rule set in Fig. 1

The Fig. 2 shows the WHILE form of the example in Fig. 1.

2.4 Equivalence

We now prove that the programming language defined using business rules and the simple looping algorithm chosen is Turing-complete. This is based on creating a simulation of WHILE programs using rules. In other words, after having transformed a BR program in a WHILE program, we now make sure that the converse is possible for any WHILE program.

Theorem 2 (Equivalence with WHILE programs)**.** *The set of programs computable using WHILE programs is the same as the set of programs computable using Business Rules.*

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 embedded **if** statements, then we discover a sequence of syntactical steps on the symbols of a generic WHILE program which transforms it into a WHILE program without embedded **if** statements.

Lemma 3. *Any WHILE program without embedded* **if** *statement can be simulated using a BR program.*

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

1: **while** $\mathsf{T}_0(x)$ **do**
2: **if** $\mathsf{T}_1(x)$ **then**
3: $x \leftarrow \mathsf{A}_1(x)$
4: **end if**
5: ...
6: **if** $\mathsf{T}_K(x)$ **then**
7: $x \leftarrow \mathsf{A}_K(x)$
8: **end if**
9: **end while**

We can write an equivalent rule set with $K+1$ rules. They are static rules, because they do not make use of meta-variables and thus produce only one rule instance each:

if $\neg\mathsf{T}_0(x_1, ..., x_n)$ **then**
$Stop$
end if
if $\mathsf{T}_1(x_1, ..., x_n)$ **then**
$x \leftarrow \mathsf{A}_1(x)$
end if
...
if $\mathsf{T}_K(x_1, ..., x_n)$ **then**
$x \leftarrow \mathsf{A}_K(x)$
end if

Using previous notations, rule R_0 has $T_0(\alpha, x) = \neg\mathsf{T}_0(x)$ with $Stop$ as action, while for $k \in \{1, ..., K\}$ rule R_k has $T_k(\alpha, x) = \mathsf{T}_k(x)$, $A_k(\alpha, x) = \alpha$ and $B_k(\alpha, x) = \mathsf{A}_k(x)$. This is obviously equivalent to the WHILE program considered.

In the (admittedly rare) case where static rules are not allowed, this is still possible using variable types. In our case, using the same execution algorithm, we use n meta-variables $\alpha_1, ..., \alpha_n$ contained in a vector α to write the following non-static BR program:

for all $i \in \{1, \ldots, n\}$ **do**
$create\_new\_type(t_i)$
$type(x_i) \leftarrow t_i$
$type(\alpha_i) \leftarrow t_i$
end for
if $type(\alpha_1) = type(x_1) \wedge \cdots \wedge type(\alpha_n) = type(x_n) \wedge \neg\mathsf{T}_0(\alpha_1, ..., \alpha_n)$ **then**
$Stop$
end if
if $type(\alpha_1) = type(x_1) \wedge \cdots \wedge type(\alpha_n) = type(x_n) \wedge \mathsf{T}_1(\alpha_1, ..., \alpha_n)$ **then**
$\alpha \leftarrow \mathsf{A}_1(\alpha)$
end if...
if $type(\alpha_1) = type(x_1) \wedge \cdots \wedge type(\alpha_K) = type(x_K) \wedge \mathsf{T}_K(\alpha_1, ..., \alpha_n)$ **then**
$\alpha \leftarrow \mathsf{A}_K(\alpha)$
end if

Using previous notations, the typing conditions are part of the definition of rule instances, and rule R_0 has $T_0(\alpha, x) = \neg\mathsf{T}_0(\alpha)$ with $Stop$ as action, while for $k \in \{1, ..., K\}$ rule R_k has $T_k(\alpha, x) = \mathsf{T}_k(\alpha)$, $A_k(\alpha, x) = \mathsf{A}_k(\alpha)$ and $B_k(\alpha, x) = x$. This is also equivalent to the WHILE program considered, since the type declarations ensure that the only viable rule instances will be the ones where $\alpha = x$.
■

Lemma 4. *Any WHILE program can be transformed into an equivalent WHILE program without embedded* **if** *statements.*

Proof. We prove the lemma by reasoning on the **ifblocks**. We consider the assertion: Any **ifblock** can be transformed into a finite sequence of **ifblocks** without embedded **if** statements.
We reason inductively on the depth of the deepest embedded **if** statement. If it is 0 or 1, the property is trivial (one is a pass instruction and the other already of the correct form).
Let $n \in \mathbb{N}$ such that we can transform any **ifblock** of depth n into a sequence of **if** statements. Let us consider an **ifblock** of depth $n+1$. It can be written as:

if $\mathsf{T}_1(x)$ **then**
$x \leftarrow \mathsf{A}_1(x)$
if $\mathsf{T}_2(x)$ **then**
$x \leftarrow \mathsf{A}_2(x)$
ifblock$(\mathsf{T}_3, \mathsf{A}_3, x)$
end if
end if

where **ifblock**(T_3, A_3, x) is an **ifblock** of depth $n-1$. It is equivalent to the following:

if $\mathsf{T}_1(x) \wedge \mathsf{T}_2(x)$ **then**
$x \leftarrow \mathsf{A}_2(\mathsf{A}_1(x))$
ifblock$(\mathsf{T}_3, \mathsf{A}_3, x)$
end if
if $\mathsf{T}_1(x) \wedge \neg\mathsf{T}_2(x)$ **then**
$x \leftarrow \mathsf{A}_1(x)$
end if

Which is a sequence of two **ifblocks** of depth n. As we can tranform each of them into a sequence of **ifblocks** without

embedded **if** statements, we have the property for **ifblocks** of depth $n+1$, and the induction is valid.
The property applied to each **ifblocks** of a WHILE program transforms it into the form we wanted. ■

2.5 Turing-machine

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 rule set that simulates a universal Turing machine (UTM). Such a rule set is exhibited in Fig. 3.

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 $(\text{state}_{\text{initial}}, \text{symbol}_{\text{initial}}, \text{right}|\text{left}|\text{stay}, \text{state}_{\text{next}}, \text{symbol}_{\text{written}})$: $R = \{(q_r^i, s_r^i, \text{act}_r, q_r^f, s_r^f) \mid \text{act}_r \in \{\text{"left"}, \text{"right"}, \text{"stay"}\}\}_r$
- 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": $T = \{(i, s_i) \mid i \in \mathbb{N}, 0 \leq i \leq l-1\}$ 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 rule set to simulate a UTM is then written in a compact form:

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

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

Figure 3: Universal Turing Machine

This universal Turing machine has a straightforward input, and terminates correctly for any valid Turing machine. 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).

3 Applications

3.1 WHILE form with other execution algorithms

While we have used a very simple execution algorithm where conflict resolution is based on a fixed ordering of rules and variables, most BR execution algorithms are more complex. Almost all of those can simulate the basic algorithm (except in extreme cases, such as a conflict resolution strategy leading to a bounded number of execution loops). As such, they are also Turing-complete. For all of those, a WHILE form can be defined (although a canonical one might not always seem obvious).

The most common conflict resolution strategies combine at least the following three elements [9]:

- *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).

These elements do not forbid the conversion of a rule set to a WHILE program. While the specifics depend on each execution algorithm, the broad strokes are that:

- *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.

3.2 Structural Operational Semantics

Not all operational semantics for BR follow the structural approach introduced by Plotkin [7] in that they use small-step semantics. The standard semantics defined in W3C's Production Rule Dialect of the Rule Interchange Format (RIF-PRD) [9] do, but are tailored to the syntax and algorithm of the standard. Other attempts at creating operational semantics for BRs have focused on being compatible with either declarative semantics or ontologies (or both) [8, 13]. Such semantics keep the structure of the rule set divided into rules, which helps with making sense of complex rule sets and creating better user experiences.
At the price of losing the knowledge representation linked to the use of rules, we can use the canonical WHILE form to obtain a structural operational semantic interpretation that is not unique to a syntax or execution algorithm, which can allow for comparison of different BR programs. Furthermore, using this semantics makes proving properties of some rule programs easier, as the working memory will not include lists of rules anymore. In a way, this semantics is a tradeoff between data complexity (RIF-PRD has rule instances in working memory) and number of small steps taken (any semantics derived from a WHILE form will go through many steps corresponding to inactive rule instances). By keeping the conflict resolution separate from the evaluation of condition clauses, the execution of the rule set is algorithmically correct yet adds complexity to the semantics that is unnecessary in most cases. The comparison of a flowchart representation of each semantic analysis for the example of Fig. 1 is made in Fig. 4 to make

the difference easier to visualize. We keep the steps in both cases bigger than they could be, so that we do not get bogged down in the evaluation of condition clauses for example.

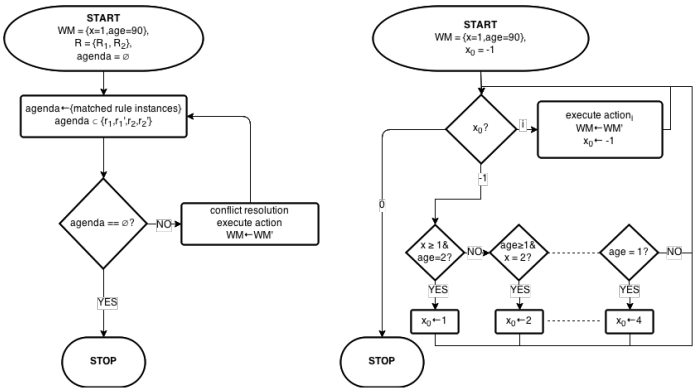

Figure 4: Representation of two semantic interpretations of the example in Fig. 1

4 Conclusion

Business Rules seem simple enough, repeatedly treating data according to a simple algorithm. The complexity of BR programs actually comes from the interpreters. In particular, almost any interpreter that uses a looping algorithm can make BR programs Turing-complete, as is the case with the simplistic algorithm we have presented in this paper. The proof of such is simple, yet it is a result that has been overlooked so far (to the best of our knowledge). The Turing-completeness of BR programs can have important theoretical implications: it links the usual Rules research on Inference Rules and ontologies with more traditional research on programming languages and computability.

We have further provided a tool to study this link in detail through the introduction of 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 structural operational semantic analysis techniques on the transformed BR programs highlights a marked difference with the existing operational semantics of Business Rules. These might inspire new techniques for studying the properties of BR programs. The use of WHILE forms also provides a way to compare BR programs that use different interpreters without needing to examine the details of either interpreter, as long as a canonical WHILE form is agreed upon.

Both directions, bringing programming to Rules via WHILE forms and bringing Rules to programming through theoretical work, seem interesting enough to merit further work.

Acknowledgments

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

References

[1] Boley, H., Paschke, A., Shafiq, O., 2010: RuleML 1.0: the overarching specification of web rules. In Proceedings of the 2010 international conference on Semantic web rules (RuleML'10), Dean M., Hall J., Rotolo A., and Tabet S. (Eds.). Springer-Verlag, Berlin, Heidelberg, 162-178.

[2] Church, A., 1936: An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58, 345-363.

[3] Curtis, M.W., 1965: A Turing Machine Simulator. Journal of the Association for Computing Machinery, Vol. 12, No. 1 (January, 1965), pp. 1-13.

[4] Gandy, R., 1980: Church's Thesis and the Principles for Mechanisms. In The Kleene Symposium, Barwise H.J., Keisler H.J., and Kunen K. (eds). North-Holland Publishing Company. pp. 123-148.

[5] Harel, D., 1980: On folk theorems. Communications of the ACM 23, 7 (July 1980), 379-389.

[6] Minsky, M., 1972: Computation: Finite and infinite machines. Prentice-Hall, London.

[7] Plotkin, G. 1981: A structural approach to operational semantics. DAIMI FN-19, Computer Science Department, Aarhus University.

[8] Rezk, M., Kifer, M., 2012: Formalizing Production Systems with Rule-Based Ontologies. In Foundations of Information and Knowledge Systems (pp. 332-351). Springer Berlin Heidelberg.

[9] de Sainte Marie, C., Hallmark, G., Paschke, A., 2013: RIF Production Rule Dialect (Second Edition). W3C Recommendation, www.w3.org/TR/2013/REC-rif-prd-20130205/

[10] Shannon, C., 1956: A universal Turing machine with two internal states. In Automata studies, volume 34 of annals of mathematics studies, Shannon C., McCarthy J. (eds). Princeton University Press, Princeton, pp 157-165

[11] Turing, A., 1937: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42(1), 230-265.

[12] Turing, A., 1939: Systems of Logic Based on Ordinals (Ph.D. thesis). Princeton University. p. 8.

[13] Zaniolo, C., 1994: A Unified Semantics for Active and Deductive Databases. In Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh, Scotland, pp 271-287. Springer London.