NORTH-HOLLAND

Formal Semantics for Rule-Based Systems

S. Murrell and R. Plant

Departments of Mathematics and Computer Science and Computer Information Systems, University of Miami, Coral Gables, Florida

The existence of a formal description of any language is a prerequisite to any rigorous methods of proof, validation, or verification. This article develops a formal semantics in the denotational style for rule-based production systems. Primarily, a formal description of a generalized, hypothetical language for rule-based expert systems is developed as a base. Building on this base, it is possible to specify in detail the variations that distinguish individual real-world systems in a usable way.

1. INTRODUCTION

The use of production systems in the development of a knowledge-based application has been popular since they were demonstrated in the MYCIN (Shortliffe, 1974) and R1 (McDermott, 1980) systems to be both natural and intuitive in nature. This, combined with their modularity and declarative style, enabled the fast and easy development of otherwise complex or infeasible systems.

The popularity of this representational form brought about a new style of programming based on the paradigm of rule-based systems, along with many shells, environments, and languages dedicated to the creation of production systems. These languages and environments also simplified many programming tasks that were previously difficult to perform, such as the rapid prototyping of expert systems.

The combination of a new programming paradigm and new approaches to development had a beneficial effect in freeing the developer and programmer from trivial chores to build creative new systems. However the growth in rule-based environments was not supported by any theoretical model of these systems. This led to significant problems in the validation and verification of rule-based systems, and research is still active in that area (O'Leary, 1987; Preece, 1993). The development of an exact formal description of the new languages is a step toward reliable validation and verification, but research in this area has primarily been focused on the specification of the shells and their inference process (Gold and Plant, 1991).

Although formal semantic specifications have been developed for the general languages of artificial intelligence—LISP (Gordon, 1973; Henderson, 1980) and PROLOG (Jones and Mycroft, 1984; Nicholson and Foo, 1989)—and general-purpose goal-directed languages (Gudeman, 1992), their application to specific domain-oriented languages, particularly those for expert systems, has been left largely unexplored. In this article, a formal semantic description for rule-based systems is presented.

Initially, the semantic system is constructed around a simple but representative system. This provides a formal description of just one hypothetical rule-based language; as such, it would be of very limited usefulness. Much of the power of a useful semantic system is in its ability to describe, within a constant framework, the variations in different systems. The semantics of several such variations are also presented.

In Section 2, a simple language for describing rule-based expert systems is presented; this is not a description of any real language, just a generic syntactic base for the semantic definitions that follow. Then, in Section 3.1, using the well-established style and notation of denotational semantics (Milne and Strachey, 1976; Stoy, 1977), the semantic domains are defined for representations of conditions and system states, after which it is possible to give definitions (Section 3.2) of the basic semantic functions $\mathscr{E}$ and $\mathscr{D}$ for evaluating expressions and checking their domains, respectively. Building on these, in

Address correspondence to Prof. Stephen Murrell, Mathematics and Computer Science, University of Miami, P.O. Box 249085, Coral Gables, FL 33124.

© 1995 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/95/$9.50
SSDI 0164-1212(94)00108-Y

Section 3.3, three additional functions, $\mathscr{V}$, $\mathscr{F}$, and $\mathscr{Q}$, are used to determine which rules may be fired and what input is required from the user. Together with function $\mathscr{A}$, which defines (Section 3.5) the effect of carrying out the possible actions or conclusions, it is possible (Section 4) to specify the actions of the whole system. Once a basic definition for simple rule-base systems has been presented, additions to cover more complex possibilities are discussed (Section 5).

2. A SIMPLE RULE-BASED SYSTEM

To avoid the specific details and idiosyncrasies of any particular implementation, we initially assume a generic rule-based system. A program in this hypothetical system consists of a collection of rules and their associated declarations.

The following are the only metaconstructs used in the syntax specifications:

[A] means that A is optional

[A|B|C...] means a choice between A, B, or C

Furthermore,

```
⟨rule⟩ →  IF ⟨cond⟩ ⟨acts⟩ [BECAUSE
          ⟨expl⟩]
⟨cond⟩ →  ⟨identifier⟩
          | ⟨relat⟩
          | ⟨cond⟩ AND ⟨cond⟩|⟨cond⟩ OR
          ⟨cond⟩|NOT ⟨cond⟩
          | (⟨cond⟩)
⟨relat⟩→  ⟨expr⟩ = ⟨expr⟩|⟨expr⟩ <
          ⟨expr⟩|⟨expr⟩ ≤ ⟨expr⟩
          | ⟨expr⟩ ≠ ⟨expr⟩|⟨expr⟩ >
          ⟨expr⟩|⟨expr⟩ ≥ ⟨expr⟩
⟨expr⟩ →  ⟨identifier⟩|⟨constant⟩
          | (⟨expr⟩)
          | ⟨expr⟩ + ⟨expr⟩|⟨expr⟩ −
          ⟨expr⟩|⟨expr⟩
          | ⟨expr⟩ × ⟨expr⟩|⟨expr⟩ ÷
          ⟨expr⟩
⟨acts⟩ →  ⟨act⟩[AND ⟨acts⟩]
⟨act⟩  →  DEDUCE
          ⟨identifier⟩[=]⟨expr⟩
⟨expl⟩ →  ⟨string-constant⟩
          | ⟨expr⟩
          | ⟨expl⟩ + ⟨expl⟩
```

Examples:

```
IF weight > class × 2240
            DEDUCE tooheavy TRUE
              BECAUSE "weighs more
            than" +class + "tons."
IF unemployed DEDUCE salary = 0
```

2.1 Modes and Declarations

Although most rule-based systems do not provide type checking of operands, this specification does. It is easier to ignore an unwanted feature than to make use of an unprovided one. The equations in Section 3.2 leave explicitly undefined the behavior when type errors occur.

To enable type checking, a declaration is required for every variable used. The declaration associates a *type*, a *mode*, and an *accessibility* with each identifier. For the sake of completeness, declarations are permitted to specify an initial value for an identifier.

```
⟨decl⟩ → ⟨acc⟩⟨mode⟩⟨type⟩⟨defs⟩
⟨acc⟩  → [MODIFIABLE]
⟨mode⟩ → ASKED|DEDUCED
⟨type⟩ → NUMBER|BOOLEAN|STRING
⟨defs⟩ → ⟨defn⟩[, ⟨defs⟩]
⟨defn⟩ → ⟨identifier⟩[= ⟨expr⟩]
```

Examples:

```
MODIFIABLE DEDUCED NUMBER x, y = 0, z
ASKED BOOLEAN HasHeadaches
```

Type is used in its traditional sense to specify the set of values that an identifier may take on; naturally, the system may easily be extended to include more types than shown here. *Mode* is used to specify a basic but flexible interface with the user. An identifier that is declared as deduced may only be given a value as the result of executing an ⟨act⟩. An identifier that was declared as asked may also receive a value in this way, but if it is ever needed before it gets a value, the user may be queried.

Access simply defines whether or not an identifier's value may ever change. To conform to the most common rule-based systems, once an identifier has been deduced to be true, it remains permanently true, and any later attempt to give it a different value is a sign of inconsistency and therefore an error. However, it may be desirable to have identifiers that behave like variables, so the option is permitted.

2.2 Programs

With these definitions, it is sufficient to state that a program is simply an arbitrary mixture of rules and declarations. It may be the case that the order in which rules are entered has some effect on the system's conflict resolution strategy, but that is not reflected in the syntax.

```
⟨prog⟩      →  empty
               | ⟨component⟩⟨prog⟩
⟨component⟩ →  ⟨decl⟩
               | ⟨rule⟩
```

3. SEMANTICS

3.1 Domains

The behavior of any rule is dependent on the current state of the system (i.e., the values associated with identifiers), so before the semantics of rules may be defined, a representation for the state must be selected.

An identifier naturally has five components: its name, accessibility, mode, type, and value; additionally, it is possible to keep track of the reasons for the current value (in systems that do not support reasons, this component may be ignored). Only the value and reason may change while a program is running; the other components are fixed.

Identifiers are represented by a cartesian product:

```
    ID = NAME × ACC × MODE × TYPE × VAL
         × REASON
  NAME = String
   ACC = {Var, Fix}
  MODE = {Ask, Ded}
  TYPE = {Num, Bool, Str}
   VAL = Real + {True, False} + String +
         {Undefined}
REASON = String
```

Examples:

```
⟨"weight", Fix, Ask, Num, 3371, "user
input"⟩
⟨"class", Fix, Ask, Num, 1, "user in-
put"⟩
⟨"tooheavy", Fix, Ded, Bool, True,
"weighs more than 1 tons"⟩
```

In this definition, the VAL component is drawn from one of four basic domains, the first three representing the sets of real numbers, Boolean values, and strings. The fourth, which does not correspond to one of the user-declarable types, contains just one value, known as undefined; its only purpose is to take the place of the value for a variable that has none. It is not appropriate to use $\perp$ instead of undefined, because the condition that it represents must be detectable.

For each of the possible identifier types T, a membership function is_T is provided:

$$\mathrm{is}_{Num}(x) \equiv x \in \mathrm{Real} \vee x = \mathrm{Undefined}$$
$$\mathrm{is}_{Bool}(x) \equiv x \in \{\mathrm{True}, \mathrm{False}\} \vee x = \mathrm{Undefined}$$
$$\mathrm{is}_{Str}(x) \equiv x \in \mathrm{String} \vee x = \mathrm{Undefined}$$

The state of a system at any time is simply the set of all currently known identifier representations, with the requirement that there may be at most one unique record for each identifier in the state, and that its value is consistent with its type.

$$\begin{aligned}
\mathrm{STATE} &= \mathrm{set(ID)}\\
\mathrm{invariants:} &\ \forall s{:}\mathrm{State},\ \forall\langle n, a, m, t, v, r\rangle, \langle n', a', m', t', v', r'\rangle \in s,\\
&\qquad (n = n') \Rightarrow (a = a') \wedge (m = m') \wedge (t = t') \wedge (v = v') \wedge (r = r')\\
&\ \forall s{:}\mathrm{State}, \forall\langle n, a, m, t, v, r\rangle \in s, \mathrm{is}_t(v)\\
\mathrm{RULE} &= \langle\mathrm{cond}\rangle \times \langle\mathrm{acts}\rangle \times \langle\mathrm{expl}\rangle\\
\mathrm{PROG} &= \mathrm{set(RULE)}
\end{aligned}$$

Each rule is reduced to a triple containing its three essential syntactic components, and the whole program (for run-time purposes) is considered to be a set of such rules.

3.2 Evaluation Functions

Given a state, conditions and expressions may be evaluated. The function $\mathscr{E}$ defines the result of this evaluation. $\mathscr{E}$ is subscripted with the expected type of its result, a formal detail, which, as a side effect, provides a type-checking mechanism.

It is not possible for the evaluation of any expression to result in a change to the state of the system, so a straightforward applicative model is possible:

$$\begin{aligned}
\mathscr{E}_T{:}\langle\mathrm{expr}\rangle \times \mathrm{STATE} &\rightarrow \mathrm{VAL}\\
\mathscr{E}_T(k{:}\langle\mathrm{constant}\rangle)_s &= \text{intuitive meaning of } k\text{, if } \mathrm{is}_T(k)\\
&\ \text{not defined otherwise}\\
\mathscr{E}_T(n{:}\langle\mathrm{identifier}\rangle)_s &= \mathrm{lookup}_T(n, s)\\
\mathscr{E}_{Num}(e_1{:}\langle\mathrm{expr}\rangle + e_2{:}\langle\mathrm{expr}\rangle)_s &= \mathscr{E}_{Num}(e_1)_s + \mathscr{E}_{Num}(e_2)_s
\end{aligned}$$

In this traditional notation for semantic functions, the STATE parameter, s, is reduced to a subscript to make the important parameter—an expression—more prominent. Of course, the state is a vital parameter, but in terms of understanding, $\mathscr{E}$ is considered to be a function from syntax to meaning. The concepts of denotational semantics are covered fully in Stoy (1977).

There would normally be a partial definition for $\mathscr{E}$ corresponding to each of the possible syntactic forms for an ⟨expr⟩, but due to the confines of this format, only significantly different cases are shown here. Syntactic labels are retained to disambiguate the cases; the definition $\mathscr{E}_T(n{:}\langle\texttt{identifier}\rangle)_s = \texttt{lookup}_T\texttt{(n, s)}$ means that if n is syntactically an identifier, then $\mathscr{E}_T(n)_s = \texttt{lookup}_T\texttt{(n, s)}$. In other words, if a result of type T is expected, and n is an identifier, and the current state is represented by s, then the meaning of n (when seen as an expression) may be obtained by looking up the identifier n in the state s, demanding a result of type T.

$$\langle \texttt{n, a, m, t, v, r}\rangle \in \texttt{s} \wedge \texttt{v} \neq \texttt{Undefined} \Leftrightarrow \texttt{lookup}_t\texttt{(i, s)} = \texttt{v}$$

The state invariant makes this definition of lookup unambiguous; it also leaves the value of an unknown identifier undefined. Semantics for specific systems may specify different behaviors with minimal modification.

$$\begin{aligned}
\mathscr{E}_{\texttt{Bool}}(e_1{:}\langle\texttt{expr}\rangle \le e_2{:}\langle\texttt{expr}\rangle)_s &= \texttt{True}, \text{if } \mathscr{E}_{\texttt{Num}}(e_1)_s \le \mathscr{E}_{\texttt{Num}}(e_2)_s \\
& \texttt{False}, \text{otherwise} \\
\mathscr{E}_{\texttt{Bool}}(c_1{:}\langle\texttt{cond}\rangle \text{ AND } c_2{:}\langle\texttt{cond}\rangle)_s &= \texttt{True}, \text{if } \mathscr{E}_{\texttt{Bool}}(c_1)_s = \texttt{True} \wedge \mathscr{E}_{\texttt{Bool}}(c_2)_s = \texttt{True} \\
& \texttt{False}, \text{otherwise}
\end{aligned}$$

The definition of $\mathscr{E}$ for Boolean expressions does not specify whether *full* or *short-circuit* evaluation is to be used. Nor could it, because $\mathscr{E}$ simply specifies the correct result, which is the same in both cases. The distinction may be made in the design of $\mathscr{D}$, below.

The function $\mathscr{D}$ is used to check that an expression or condition is being evaluated within its domain and will not result in an invalid operation. This is not simply to make the system under definition more robust, which would be unrealistic modeling, but also to ensure that these semantic functions do not become inconsistent. The other semantic functions (particularly $\mathscr{E}$ and $\mathscr{A}$) have their domains implicitly restricted to only those states for which $\mathscr{D}$ would return True.

$$\begin{aligned}
\mathscr{D}_T{:}\ \langle\texttt{expr}\rangle \times \texttt{STATE} &\rightarrow \{\texttt{True, False}\} \\
\mathscr{D}_T(k{:}\langle\texttt{constant}\rangle)_s &= \texttt{True}, \text{if } \texttt{is}_T\texttt{(k)} \\
& \texttt{False}, \text{otherwise} \\
\mathscr{D}_T(n{:}\langle\texttt{identifier}\rangle)_s &= \texttt{True}, \text{if } \langle\texttt{n, a, m, T, v, r}\rangle \in \texttt{s} \wedge \texttt{v} \neq \texttt{Undefined} \\
& \texttt{False}, \text{otherwise} \\
\mathscr{D}_{\texttt{Num}}(e_1{:}\langle\texttt{expr}\rangle + e_2{:}\langle\texttt{expr}\rangle)_s &= \mathscr{D}_{\texttt{Num}}(e_1)_s \wedge \mathscr{D}_{\texttt{Num}}(e_2)_s \\
\mathscr{D}_{\texttt{Bool}}(e_1{:}\langle\texttt{expr}\rangle \le e_2{:}\langle\texttt{expr}\rangle)_s &= \mathscr{D}_{\texttt{Num}}(e_1)_s \wedge \mathscr{D}_{\texttt{Num}}(e_2)_s \\
\mathscr{D}_{\texttt{Bool}}(c_1{:}\langle\texttt{cond}\rangle \text{ AND } c_2{:}\langle\texttt{cond}\rangle)_s &= \mathscr{D}_{\texttt{Bool}}(c_1)_s \wedge (\mathscr{E}_{\texttt{Bool}}(c_1)_s = \texttt{False} \vee \mathscr{D}_{\texttt{Bool}}(c_2)_s)
\end{aligned}$$

The definition for AND specifies that A AND B is only valid if A is valid and false, or if A and B are both valid, that is, short-circuit evaluation, which is essential if *Askable* variables are to be usable. A similar definition holds for OR:

$$\mathscr{D}_{\mathtt{Bool}}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{OR}\ c_2{:}\langle\mathtt{cond}\rangle)_s = \mathscr{D}_{\mathtt{Bool}}(c_1)_s \wedge (\mathscr{E}_{\mathtt{Bool}}(c_1)_s = \mathtt{True} \vee \mathscr{D}_{\mathtt{Bool}}(c_2)_s)$$

Finally, the obviously wrong expressions must be excluded:

$$\begin{aligned}
\mathscr{D}_{\mathtt{Bool}}(e_1{:}\langle\mathtt{expr}\rangle + e_2{:}\langle\mathtt{expr}\rangle)_s &= \mathtt{False}\\
\mathscr{D}_{\mathtt{Num}}(e_1{:}\langle\mathtt{expr}\rangle \le e_2{:}\langle\mathtt{expr}\rangle)_s &= \mathtt{False}\\
\mathscr{D}_{\mathtt{Num}}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{AND}\ c_2{:}\langle\mathtt{cond}\rangle)_s &= \mathtt{False}
\end{aligned}$$

A nonsubscripted variant of $\mathscr{D}$ is used for checking actions when no result value is produced; this is defined in Section 3.5.

It should be noted that implementations are not required to check $\mathscr{D}$, which would represent a great deal of wasted computation. $\mathscr{D}$ is only used to restrict the domain of the semantic model so that the results of invalid operations are not specified.

3.3 Variables

The function $\mathscr{V}$: $\langle\mathtt{expr}\rangle \times \mathtt{State} \rightarrow \mathtt{set(Identifier)}$ provides a list of those identifiers that are required for the evaluation of an expression, but are still both undefined and askable. In other words, $\mathscr{V}$ determines which questions might usefully be asked.

$$\begin{aligned}
\mathscr{V}(n{:}\langle\mathtt{identifier}\rangle)_s &= \{n\} \text{ if } \langle\mathtt{n,\ a,\ Ask,\ t,\ Undefined,\ r}\rangle \in s\\
&\quad \{\ \} \text{ otherwise}\\
\mathscr{V}(k{:}\langle\mathtt{constant}\rangle)_s &= \{\ \}\\
\mathscr{V}(e_1{:}\langle\mathtt{expr}\rangle + e_2{:}\langle\mathtt{expr}\rangle)_s &= \mathscr{V}(e_1)_s \cup \mathscr{V}(e_2)_s\\
\mathscr{V}(e_1{:}\langle\mathtt{expr}\rangle \le e_2{:}\langle\mathtt{expr}\rangle)_s &= \mathscr{V}(e_1)_s \cup \mathscr{V}(e_2)_s\\
\mathscr{V}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{AND}\ c_2{:}\langle\mathtt{cond}\rangle)_s &= \mathscr{V}(c_1)_s \text{ if } \mathscr{V}(c_1)_s \ne \{\ \}\\
&\quad \mathscr{V}(c_2)_s \text{ if } \mathscr{V}(c_1)_s = \{\ \} \wedge \mathscr{D}_{\mathtt{Bool}}(c_1)_s \wedge \mathscr{E}_{\mathtt{Bool}}(c_1)_s = \mathtt{True}\\
&\quad \{\ \} \text{ otherwise}\\
\mathscr{V}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{OR}\ c_2{:}\langle\mathtt{cond}\rangle)_s &= \mathscr{V}(c_1)_s \text{ if } \mathscr{V}(c_1)_s \ne \{\ \}\\
&\quad \mathscr{V}(c_2)_s \text{ if } \mathscr{V}(c_1)_s = \{\ \} \wedge \mathscr{D}_{\mathtt{Bool}}(c_1)_s \wedge \mathscr{E}_{\mathtt{Bool}}(c_1)_s = \mathtt{False}\\
&\quad \{\ \} \text{ otherwise}\\
\mathscr{V}(\mathtt{NOT}\ c{:}\langle\mathtt{cond}\rangle)_s &= \mathscr{V}(c)_s
\end{aligned}$$

The definition for A AND B should be read thus: If A contains unknown values, then those are the unknown values required for A AND B (the unknowns in B are not necessarily required; A may yet evaluate to false). If A has no unknowns, is valid, and evaluated to True, then the unknowns of B are required.

The purpose of $\mathscr{V}$ is to determine which questions may be asked of the user as a precursor to the firing of a rule. $\mathscr{V}$ and $\mathscr{D}$ together determine whether a rule may even be considered for firing, and $\mathscr{E}$ (applied to the rule's condition) makes the final determination of firability and aids in the calculation of the effects of any rule that is fired.

3.4 Selection

In the context of a set of rules that are under consideration for possible firing, the subset of rules that may be fired immediately may be determined. They are those rules for which no still-to-be-asked variables appear in their conditions, the current state satisfies the domain constraints for all three parts (condition, actions, reason), and the condition evaluates to True.

$$\begin{aligned}
&\mathscr{F}\colon \mathtt{set(RULE)} \times \mathtt{STATE} \rightarrow \mathtt{set(RULE)}\\
&\mathscr{F}(\mathtt{rr})_s = \{\langle\mathtt{c,\ a,\ r}\rangle \in \mathtt{rr} \mid \mathscr{D}_{\mathtt{Bool}}(c)_s \wedge \mathscr{D}(a)_s \wedge \mathscr{D}_{\mathtt{Str}}(r)_s \wedge \mathscr{V}(c)_s = \{\ \} \wedge \mathscr{E}_{\mathtt{Bool}}(c)_s = \mathtt{True}\}
\end{aligned}$$

This specifies that a rule may not be fired if so doing would cause an error when evaluating the actions or the reason. Typical systems do not look this far ahead. For a more accurate model in these cases, the conditions may be relaxed.

In the same context, of a set of rules under consideration, the set of variables that could usefully be asked of the user may also be determined. If there are already firable rules, then there is no point in asking any questions; otherwise, any variable returned by $\mathscr{V}$ for one of the rules is a candidate.

$$\begin{aligned}
&\mathscr{Q}\colon \mathtt{set(RULE)} \times \mathtt{STATE} \rightarrow \mathtt{set}(\langle\mathtt{identifier}\rangle)\\
&\mathscr{Q}(\mathtt{rr})_s = \{\ \} \text{ if } \mathscr{F}(\mathtt{rr})_s \ne \{\ \}\\
&\qquad = \{i{:}\langle\mathtt{identifier}\rangle \mid \exists \langle\mathtt{c,\ a,\ r}\rangle \in \mathtt{rr}.\ i \in \mathscr{V}(c)_s\} \text{ otherwise}
\end{aligned}$$

In general, a system will ask just one of the questions returned by $\mathscr{Q}$ before reconsidering what step to take next, by reevaluating $\mathscr{F}$.

3.5 Actions

An addition to $\mathscr{D}$ is required to ensure the validity of a deduction. This variant is not subscripted, because there is no result and therefore no type to check.

The deduction of a new value for a variable is valid only if the type of that value is the same as the declared type of the variable. It is also required that the variable either has no current value or was declared as modifiable.

$$\begin{aligned}&\mathscr{D}(\texttt{DEDUCE}\ n{:}\langle\texttt{identifier}\rangle = e{:}\langle\texttt{expr}\rangle)_s\\ &\quad= \mathscr{D}_t(e)_s \text{ if } \langle \texttt{n, a, m, t, v, r}\rangle \in s\\ &\qquad\quad \text{and } (\texttt{v} = \texttt{Undefined})\\ &\qquad\qquad \vee(\texttt{a} = \texttt{Var})\\ &\quad \texttt{False} \text{ otherwise}\end{aligned}$$

Naturally, a multiple action is only valid if all of its components are valid. The first component's validity may be tested under the current state of the system, but all subsequent components will be executed in the state as modified by the prior execution of the first component. $\mathscr{A}$ models the result of performing a deduction, and must be used here.

$$\begin{aligned}&\mathscr{D}(a_1{:}\langle\texttt{act}\rangle \text{ AND } a_2{:}\langle\texttt{acts}\rangle)_s\\ &\quad= \texttt{False}, \text{ if } \mathscr{D}(a_1)_s = \texttt{False}\\ &\qquad (\mathscr{D}(a_2)\circ\mathscr{A}(a_1))_s \text{ otherwise}\end{aligned}$$

$\mathscr{A}$ itself has a relatively simple definition. The domain of $\mathscr{A}$ is assumed to contain only those combinations of actions and states for which $\mathscr{D}$ returns `True`, so no further checking is required.

The result of an action is a modification to the state of the system, so the result type of $\mathscr{A}$ is `STATE`. When multiple actions are performed, the first is performed in the initial system state, but all subsequent actions are performed in the resultant modified state, so a composition of functions is used, as with $\mathscr{D}$.

$$\begin{aligned}\mathscr{A}{:}\langle\texttt{acts}\rangle \times \texttt{STATE} &\to \texttt{STATE}\\ \mathscr{A}(a_1{:}\langle\texttt{act}\rangle \text{ AND } a_2{:}\langle\texttt{acts}\rangle)_s &= (\mathscr{A}(a_2)\circ\mathscr{A}(a_1))_s\\ \mathscr{A}(\texttt{DEDUCE}\ n{:}\langle\texttt{identifier}\rangle = e{:}\langle\texttt{expr}\rangle \texttt{BECAUSE}\ r)_s &= \texttt{Assign}(\texttt{n, e, r})_s\end{aligned}$$

$$\begin{aligned}&\langle \texttt{n, a, m, t, v, r}\rangle \in s \wedge \mathscr{D}_t(e)_s \wedge \mathscr{D}_{\texttt{Str}}(r)_s \Rightarrow\\ &\qquad \texttt{Assign}(\texttt{n, e, r})_s = s \oplus \langle \texttt{n, a, m, t, } \mathscr{E}_t(\texttt{e})_s\texttt{, } \mathscr{R}(\texttt{r})_s\rangle\end{aligned}$$

In the above definition, $\oplus$ is a simple updating operator that removes any existent record for a particular variable, replacing it with a new one. It could be formally defined thus:

$$\begin{aligned}&s \oplus \langle \texttt{n, a, m, t, v, r}\rangle =\\ &\qquad \{\langle \texttt{n', a', m', t', v', r'}\rangle \in s \mid n \neq n'\}\\ &\qquad \cup\{\langle \texttt{n, a, m, t, v, r}\rangle\}\end{aligned}$$

The definition of `Assign` specifies that only if the variable already exists and has the appropriate type may an assignment take place; when it does, the value and reason associated with the variable are replaced, but the other information remains unchanged. Such type checking is unnecessary in this context, because the prior use of $\mathscr{D}$ ensures that no type errors will occur, but in later uses of `Assign`, this protection is not available.

$\mathscr{R}$ is a simple string-evaluating function used to construct reasons. Using $\|$ as a string concatenation function:

$$\begin{aligned}\mathscr{R}{:}\langle\texttt{expl}\rangle \times \texttt{STATE} &\to \texttt{String}\\ \mathscr{R}(\texttt{k}{:}\langle\texttt{string-constant}\rangle)_s &= \texttt{k}\\ \mathscr{R}(e_1{:}\langle\texttt{expl}\rangle + e_2{:}\langle\texttt{expl}\rangle)_s &= \mathscr{R}(e_1)\|\mathscr{R}(e_2)\\ \mathscr{R}(e{:}\langle\texttt{expl}\rangle)_s &= \mathscr{R}_{\texttt{Int}}(\mathscr{E}_{\texttt{Num}}(e)_s) \text{ if } \mathscr{D}_{\texttt{Num}}(e)_s\\ &\quad\ \mathscr{R}_{\texttt{Bool}}(\mathscr{E}_{\texttt{Bool}}(e)_s) \text{ if } \mathscr{D}_{\texttt{Bool}}(e)_s, \text{ etc.}\\ \mathscr{R}_{\texttt{Int}}(n) &= \mathscr{R}_{\texttt{Digit}}(n) \text{ if } n < \texttt{10}\\ &\quad\ \mathscr{R}_{\texttt{Int}}(\lfloor n \div \texttt{10}\rfloor)\|\mathscr{R}_{\texttt{Digit}}(n \bmod \texttt{10}) \text{ otherwise}\\ \mathscr{R}_{\texttt{Digit}}(\texttt{0}) &= \text{“}\texttt{0}\text{”, etc.}\\ \mathscr{R}_{\texttt{Bool}}(\texttt{True}) &= \text{“}\texttt{True}\text{”}\\ \mathscr{R}_{\texttt{Bool}}(\texttt{False}) &= \text{“}\texttt{False}\text{”}\end{aligned}$$

3.6 Initial State

The initial state of the system may be obtained from $\mathscr{I}(\text{program})_{\{\ \}\{\ \}}$; $\mathscr{I}$ builds up an initial state and a representation of the rules in its second and third (subscripted) parameters as it scans the program, finally returning the two as a pair:

$$
\begin{aligned}
&\mathscr{I}\colon \langle\texttt{prog}\rangle \times \texttt{STATE} \times \texttt{PROG}\\
&\qquad\qquad\qquad \rightarrow \texttt{STATE} \times \texttt{PROG}\\
&\mathscr{I}(\texttt{empty})_{\texttt{s},\ \texttt{p}} = \langle \texttt{s},\ \texttt{p}\rangle\\
&\mathscr{I}(\texttt{IF c:}\langle\texttt{cond}\rangle\ \texttt{a:}\langle\texttt{acts}\rangle\\
&\quad \texttt{BECAUSE r:}\langle\texttt{expl}\rangle\ \texttt{pp:}\langle\texttt{prog}\rangle)_{\mathbf{s,p}}\\
&\qquad\qquad = \mathscr{I}(\texttt{pp})_{\texttt{s},\ (\texttt{p}\cup\{\langle \texttt{c, a, r}\rangle\})}\\
&\mathscr{I}(\texttt{a:}\langle\texttt{acc}\rangle\ \texttt{m:}\langle\texttt{mode}\rangle\ \texttt{t:}\langle\texttt{type}\rangle\\
&\qquad \texttt{n:}\langle\texttt{identifier}\rangle\ \texttt{pp:}\langle\texttt{prog}\rangle)_{\texttt{s},\ \texttt{p}}\\
&\ = \langle\{\ \},\{\ \}\rangle \text{ if } \langle \texttt{n},\ \texttt{a}',\ \texttt{m}',\ \texttt{t}',\ \texttt{v}',\ \texttt{r}'\rangle \in \texttt{s}\\
&\quad \mathscr{I}(\texttt{pp})_{(\texttt{s}\cup\{\langle \texttt{n,a,m,t,Undefined, ``\ ''}\rangle\}),\,\texttt{p}} \text{ otherwise}\\
&\mathscr{I}(\texttt{a:}\langle\texttt{acc}\rangle\ \texttt{m:}\langle\texttt{mode}\rangle\ \texttt{t:}\langle\texttt{type}\rangle\\
&\quad \texttt{n:}\langle\texttt{identifier}\rangle = \texttt{e:}\langle\texttt{expr}\rangle\texttt{pp:}\langle\texttt{prog}\rangle)_{\texttt{s,p}}\\
&\ = \langle\{\ \},\{\ \}\rangle \text{ if } \langle \texttt{n},\ \texttt{a}',\ \texttt{m}',\ \texttt{t}',\ \texttt{v}',\ \texttt{r}'\rangle \in \texttt{s}\\
&\quad \langle\{\ \},\{\ \}\rangle \text{ if } \mathscr{D}_{\texttt{t}}(\texttt{e})_{\{\ \}} = \texttt{False}\\
&\quad \mathscr{I}(\texttt{pp})_{(\texttt{s}\cup\{\langle \texttt{n,a,m,t,}\mathscr{E}_{\texttt{t}}(\texttt{e})_{\{\ \}},\texttt{``\ ''}\rangle\}),\,\texttt{p}} \text{ otherwise}
\end{aligned}
$$

In the above four clauses, the second adds a new rule to the rule base, the third adds a new variable after checking that the variable in question has not already been declared, and the fourth adds a new variable with an initial value after first checking that the expression for the value is valid. The first clause returns the accumulated results as a pair when the whole program has been processed.

4. THE EXECUTION CYCLE

The execution cycle of a typical system, after the initial state has been constructed, would be as follows:

$\mathscr{F}$ is used to determine which rules may fire immediately.
- if none, $\mathscr{Q}$ is used to determine which questions may be asked.
  - if none, execution halts.
  - otherwise, one question is asked, and $\mathscr{F}$ reevaluated.
- The process is repeated until a firable rule is found.

$\mathscr{A}$ is used to determine the results.

This procedure validly describes systems in which there is no concept of askable variables; in such cases, $\mathscr{Q}$ always returns { }.

To model a complete system, including user interactions, one final compound state is introduced: `SYS` is a compound of four parts, representing the current state (set of records for variables), the program under execution (a set of rules), the user input yet to be read (a list of values), and the output so far produced (a list of strings). As normal execution progresses, the state component is updated as the result of deductions, and rules are removed from the program as they fire. When interaction with the user is required, a question is added to the end of the output list, and an answer is taken from the head of the input list.

The input list is considered to be predetermined in that inputs are taken from it at the same time as outputs are put onto the output list. This may not seem to correctly model user interaction, but it should be borne in mind that the semantics are intended to specify the correct result from any given conditions; the time at which those conditions came into existence is not relevant.

```
SYS  = STATE × PROG × IN × OUT
 IN  = list(VAL)
OUT  = list(String)
```

The function `Step: SYS → SYS` modes a single step in the running of a system; it is defined with three mutually exclusive conditional clauses:

$$
\begin{aligned}
&\mathscr{F}(\texttt{p})_{\texttt{s}} \neq \{\ \} \Rightarrow \exists \texttt{r} \in \mathscr{F}(\texttt{p})_{\texttt{s}}.\\
&\quad \texttt{Step}\langle \texttt{s},\ \texttt{p},\ \texttt{i},\ \texttt{o}\rangle = \langle \mathscr{A}(\texttt{r})_{\texttt{s}},\ \texttt{p} \setminus \texttt{r},\ \texttt{i},\ \texttt{o}\rangle
\end{aligned}
$$

This first clause states that if there are any firable rules, then one of those rules is executed to update the state and removed from the program to prevent multiple firings.

$$
\begin{aligned}
&\mathscr{Q}(\texttt{p})_{\texttt{s}} \neq \{\ \} \Rightarrow \exists \texttt{v} \in \mathscr{Q}(\texttt{p})_{\texttt{s}}.\ \texttt{Step}\langle \texttt{s},\ \texttt{p},\ \texttt{i},\ \texttt{o}\rangle\\
&\quad = \langle \texttt{Assign}(\texttt{v}, \texttt{hd(i)}, \text{``}\texttt{input}\text{''})_{\texttt{s}},\\
&\qquad\qquad\qquad \texttt{p},\ \texttt{tl(i)},\ \texttt{o}\|\texttt{v}\|\text{``?''}\rangle
\end{aligned}
$$

The second states that if there are any askable questions (which implies that there are no firable rules), then one of those questions (i.e., variables) is appended to the output list as a question, and the head of the input list is assigned to be the new value of that variable in the updated state.

$$
\begin{aligned}
&\mathscr{F}(\texttt{p})_{\texttt{s}} = \{\ \} \wedge \mathscr{Q}(\texttt{p})_{\texttt{s}} = \{\ \} \Rightarrow\\
&\quad \texttt{Step}\langle \texttt{s},\ \texttt{p},\ \texttt{i},\ \texttt{o}\rangle = \langle \texttt{s},\ \texttt{p},\ \texttt{i},\ \texttt{o}\rangle
\end{aligned}
$$

The final clause states that if there are no firable rules and no askable questions, then nothing happens.

The function `Run` may now be defined as the least upper bound, or fixed point of `Step`, to describe a complete run of the system from an initial state. This is possible because each of the useful operations of `Step` is guaranteed to change the system.

$$\texttt{Run} = \bigsqcup \texttt{Step}$$

5. VARIATIONS

The specification presented above makes a number of assumptions that are not valid in most real systems. The variations required to describe the most common real cases are generally very slight.

5.1 Evaluation of Logical Expressions

Commonly, logical expressions are subject to short-circuit evaluation; the first operand of an AND or OR operation is always evaluated first, and the second operand is only evaluated if the value of the first makes it necessary. In some implementations, full evaluation is used. The two strategies are different in two ways: the former frequently results in faster execution (irrelevant to the semantics), and the latter may result in more run-time errors (consider the expression A AND B, when A is False and B is undefined. Short-circuit evaluation would return False; full evaluation would attempt to evaluate B and cause an error).

Full evaluation may be specified through some simple alterations to the functions $\mathscr{D}$ and $\mathscr{V}$. First, $\mathscr{D}$ is changed to insist that the second operand of AND or OR is valid, regardless of the value of the first operand:

$$\mathscr{D}_{\mathtt{Bool}}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{AND}\ c_2{:}\langle\mathtt{cond}\rangle)_s = \mathscr{D}_{\mathtt{Bool}}(c_1)_s \wedge \mathscr{D}_{\mathtt{Bool}}(c_2)_s$$
$$\mathscr{D}_{\mathtt{Bool}}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{OR}\ c_2{:}\langle\mathtt{cond}\rangle)_s = \mathscr{D}_{\mathtt{Bool}}(c_1)_s \wedge \mathscr{D}_{\mathtt{Bool}}(c_2)_s$$

Second, $\mathscr{V}$, which determines the set of askable variables that are essential in an expression, but still undefined, is similarly altered:

$$\mathscr{V}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{AND}\ c_2{:}\langle\mathtt{cond}\rangle)_s = \mathscr{V}(e_1)_s \cup \mathscr{V}(e_2)_s$$
$$\mathscr{V}(c_1{:}\langle\mathtt{cond}\rangle\ \mathtt{OR}\ c_2{:}\langle\mathtt{cond}\rangle)_s = \mathscr{V}(e_1)_s \cup \mathscr{V}(e_2)_s$$

5.2 Undefined Variables

The function $\mathscr{D}_T$ returns false if applied to an undefined identifier. This means that $\mathscr{E}_T$ is never applied to undefined identifiers, which in turn means that no attempt is ever made to evaluate undefined values; the possibility of run-time errors or default values does not arise. If this is not the true behavior of the system in question (and almost invariably, it is not), simple modifications to the specification are again possible.

The first is to remove the protection that $\mathscr{D}$ provides against ever looking at an undefined variable:

$$\mathscr{D}_T(n{:}\langle\mathtt{identifier}\rangle)_s = \begin{cases}\mathtt{True}\ \text{if}\ \langle\mathtt{n,\ a,\ m,\ T,\ v,\ r}\rangle \in \mathtt{s}\\ \mathtt{False}\ \text{otherwise}\end{cases}$$

If a run-time error is the correct behavior when an undefined identifier is used, then it is sufficient to specify $\perp$ ("bottom") as the result of lookup. As the bottom of the semantic domains, a result of $\perp$ will be propagated through any functions or operators that subsequently receive it as an operand.

$$\langle \mathtt{n,\ a,\ m,\ t,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} \neq \mathtt{Undefined} \Rightarrow \mathtt{lookup}_t(\mathtt{i,\ s}) = \mathtt{v}$$
$$\langle \mathtt{n,\ a,\ m,\ t,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} = \mathtt{Undefined} \Rightarrow \mathtt{lookup}_t(\mathtt{i,\ s}) = \perp$$

The same result could be obtained more explicitly by allowing lookup to return Undefined as its result, and redefining all of the functions that could use the result of a lookup to have Undefined as a 0. However, such a change would introduce a very large and unnecessary volume to the specification.

If a default value is to be returned for undefined identifiers, that, too, may be built into the definition of lookup:

$$\langle \mathtt{n,\ a,\ m,\ t,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} \neq \mathtt{Undefined} \Rightarrow \mathtt{lookup}_t(\mathtt{i,\ s}) = \mathtt{v}$$
$$\langle \mathtt{n,\ a,\ m,\ Num,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} = \mathtt{Undefined} \Rightarrow \mathtt{lookup}_{\mathtt{Num}}(\mathtt{i,\ s}) = 0$$
$$\langle \mathtt{n,\ a,\ m,\ Bool,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} = \mathtt{Undefined} \Rightarrow \mathtt{lookup}_{\mathtt{Bool}}(\mathtt{i,\ s}) = \mathtt{False}$$
$$\langle \mathtt{n,\ a,\ m,\ Str,\ v,\ r}\rangle \in \mathtt{s} \wedge \mathtt{v} = \mathtt{Undefined} \Rightarrow \mathtt{lookup}_{\mathtt{Str}}(\mathtt{i,\ s}) = \text{“ ”}$$

5.3 Conflict Resolution

The function Run is specified to just pick one possible firable rule, or one possible askable question, leaving unspecified the exact method used to make a choice (i.e., the conflict resolution strategy). Having a nondeterministic choice may be the best answer, but it is not the usual answer. Most systems give priority to the rule that was entered first.

This requires four simple changes:

1. The domain PROG must be redefined as list (RULE) instead of set(RULE).

   PROG = list(RULE)

2. The initial-state-generating function $\mathscr{I}$ must compile the PROG as a list, replacing the $\cup$ operator with a $\|$.

$$\mathscr{A}(\texttt{empty})_{\texttt{s, p}} = \langle \texttt{s}, \texttt{p} \rangle$$
$$\mathscr{A}(\texttt{IF c:}\langle\texttt{cond}\rangle\ \texttt{a:}\langle\texttt{acts}\rangle\ \texttt{BECAUSE r:}\langle\texttt{expl}\rangle\ \texttt{pp:}\langle\texttt{prog}\rangle)_{\texttt{s, p}} = \mathscr{A}(\texttt{pp})_{\texttt{s}, (\texttt{p}\|\{\langle \texttt{c, a, r}\rangle\})}$$

etc.

3. The function $\mathscr{F}$ should produce a list of firable rules instead of a set.

$$\mathscr{F}(\langle\rangle)_{\texttt{s}} = \langle\rangle$$
$$\mathscr{F}(\langle \texttt{c, a, r}\rangle \| l)_{\texttt{s}} = \langle \texttt{c, a, r}\rangle \| \mathscr{F}(l)_{\texttt{s}}$$
$$\text{if } \mathscr{D}_{\texttt{Bool}}(\texttt{c})_{\texttt{s}} \wedge \mathscr{D}(\texttt{a})_{\texttt{s}} \wedge \mathscr{D}_{\texttt{Str}}(\texttt{r})_{\texttt{s}}$$
$$\wedge \mathscr{V}(\texttt{c})\texttt{s} = \{\ \} \wedge \mathscr{E}_{\texttt{Bool}}(\texttt{c})_{\texttt{s}} = \texttt{True}$$
$$\mathscr{F}(l)_{\texttt{s}} \text{ otherwise}$$

4. The partial definition of `Step` for a nonempty set of immediately firable rules must be modified to expect a list of firable rules, and always take the first.

$$\mathscr{F}(\texttt{p})_{\texttt{s}} \neq \langle\rangle \Rightarrow \texttt{Step}\langle \texttt{s, p, i, o}\rangle = \langle \mathscr{A}(\texttt{r})_{\texttt{s}}, \texttt{p} \setminus \texttt{r}, \texttt{i}, \texttt{o}\rangle, \text{ where } \texttt{r} = \texttt{head}(\mathscr{F}(\texttt{p})_{\texttt{s}})$$

If similar transformations were applied to the selections of the question to be asked if there are no immediately firable rules, then the result would be an entirely deterministic specification. This would be a much more comfortable and satisfactory situation, but unfortunately not necessarily realistic.

5.4 Refirable Rules

The specification of `Step` shows that a rule is removed from the program once it has fired. This prevents multiple firings and provides a guarantee of progress during execution. Many systems behave in this way, but many do not. If refirability is required, then rules must no longer be removed from the program after each firing:

$$\mathscr{F}(\texttt{p})_{\texttt{s}} \neq \{\ \} \Rightarrow \exists \texttt{r} \in \mathscr{F}(\texttt{p})_{\texttt{s}}.$$
$$\texttt{Step}\langle \texttt{s, p, i, o}\rangle = \langle \mathscr{A}(\texttt{r})_{\texttt{s}}, \texttt{p, i, o}\rangle$$

However, this could result in a situation in which an application of `Step` to a state results in no change to that state. In a nondeterministic system, this would not imply that no further deductions are possible, but would mean that the unchanged state is a fixed point of the `Step` function, and therefore that `Run` may no longer be defined as the least fixed point of `Step`. In such a case, it is better to provide an explicit definition of `Run` stating that `Step` is repeatedly applied until there are no firable rules and no askable questions:

$$\mathscr{F}(\texttt{p})_{\texttt{s}} = \{\ \} \wedge \mathscr{Q}(\texttt{p})_{\texttt{s}} = \{\ \} \Rightarrow \texttt{Run}\langle \texttt{s, p, i, o}\rangle = \langle \texttt{s, p, i, o}\rangle$$
$$\mathscr{F}(\texttt{p})_{\texttt{s}} \neq \{\ \} \vee \mathscr{Q}(\texttt{p})_{\texttt{s}} \neq \{\ \} \Rightarrow \texttt{Run}\langle \texttt{s, p, i, o}\rangle = \texttt{Run}(\texttt{Step}\langle \texttt{s, p, i, o}\rangle)$$

6. CONCLUSION

This article has shown that a rigorous, formal description of a production system is well within reach. It is possible to give a complete definition of every detail of the execution of a rule-based expert system program, either for reference or to enable a proof of correctness, or complete verification and validation. Furthermore, the more subtle variations between different implementations may be clearly described as differences in the semantics.

For this development, the denotational (Milne and Strachey, 1976; Stoy, 1977) style of semantics was selected because it gives a very flexible framework on which to build a direct specification. Future work using an axiomatic approach is likely to be equally rewarding and perhaps more generally acceptable, because the notation used to express axiomatic definitions tends to be less intimidating to the uninitiated.

REFERENCES

Gold, D. I., and Plant, R. T., Towards a formal specification of an OPS5 production system architecture, *AAAI Workshop on Validation and Verification*, 1991.

Gordon, M. J. C., Models of Pure Lisp, Experimental Programming Reports 31, Department of Machine Intelligence, University of Edinburgh, 1973.

Gudeman, D. A., Denotational Semantics of a Goal-Directed Language, *ACM Trans. Progr. Lang. Syst.* 14 (1992).

Henderson, P., *Functional Programming*, Prentice-Hall International, London, 1980.

Jones, N. D., and Mycroft, A., A stepwise development of operational and denotational semantics for prolog, in *Proceedings of 1984 International Symposium on Logic Programming*, IEEE Computer Society Press, Washington, DC, 1984.

McDermott, J., R1: A Rule-Based Configure, Report CMU-CS-80-119, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1980.

Milne, R. E., and Strachey, C., *A Theory of Programming Language Semantics*, Chapman and Hall, London, 1976.

Nicholson, T., and Foo, N., A Denotational Semantics for Prolog, *ACM Trans. Progr. Lang.* 11 (1989).

O'Leary, D. E., Validation of Expert Systems—with Applications to Auditing and Accounting *Decis. Sci.* 18, 468–486 (1987).

Preece, A. D., Validation and Verification of Knowledge-Based Systems, Working Notes, AAAI, Menlo Park, California, 1993.

Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1974.

Stoy, J. E., *Denotational Semantics*, M.I.T. Press, Cambridge, Massachusetts, 1977.