On the Semantics and Interpretation of Rule Based Programs with Static Global Variables

Manfred Schmidt-Schauß

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail:schauss@informatik.uni-frankfurt.de

Abstract. A rule based program is a set of production rules of the form "when condition then assignment". We propose to use a restricted form of rules as a rule-based programming paradigm. We sacrifice the possibility of explicitely assigning new values to a variable. The gain is that the final result of running a program is independent from the order of rule execution and that rule execution can be parallelized without restrictions. A further gain is that rules are really declarative, and thus local changes have a more predictable effect than in general rule based programs.
We propose as a natural semantics a semi-lattice ordered by information content. Using this semantics we are able to show that rules can be independently executed, even in parallel in a distributed environment, if only minor restrictions are satisfied. This semantics also gives clear hints on the implementation of default rules that use meta-predicates like unknown? in their precondition.
There are several interesting specialisations. One is that the information content of a variable may only be "unknown" or "completely known", i.e. the truth values of a three-valued logic. Another specialization is interval arithmetic on numbers. This models the situation where the value of a variable is known to be within some interval.

Keywords: knowledge representation, semantics, rule-based program, production rules, expert systems, default rules, parallel execution, three-valued logic

1 Introduction

Production rule based programming has been in use since the early expert systems ([BFKM85, BS84]) and can be seen as an forerunner of the declarative programming style. Nowadays, knowledge based systems are based on an integration of two or more programming styles, like integration of logic and functional programming, which permits to use features of different programming styles, but may loose the declarativeness of rules. The goal of this paper is to demonstrate that the production rule style of programming remains useful and powerful, if

the usage of global variables is restricted. Informally, the necessary restriction is to view global variables as containing knowledge about an entity in the world, and the program execution generates more knowledge using the available knowledge. This excludes for example the usage of counters or variables that gather evidence by manipulating a global variable.

Dropping general assignments appears to be rather restrictive, since most programming techniques used in rule based systems cannot be used, as they depend on mutable global variables or on the execution order of rules. However, there are several advantages. The rule-based program is independent of the flow of control. On one hand this permits local verification of the soundness of rules, on the other hand the compiler can rearrange the rule execution in order to improve efficiency or to execute the rules concurrently.

There are two techniques to compute values, one is evaluation of expressions, like in applicative programming, and the other is to gather information in global variables by overwriting unknown variables with their value, when this value becomes known. Hence there is an implicit way of modifying global variables. A natural generalization is an interval arithmetic. The idea is that a program computes constraints on the values of global variables. This is a practically useful extension, since in general the input is not an exact value, for example measured values are in general exact within some given error range.

A successful instance of the programming paradigm is the system Pro.M.D. ([PT88], [TP90]) that is in everyday use in a clinical laboratory. Pro.M.D. is a rule based expert system shell that is successfully used with several different knowledge bases for routine diagnosis and clinical chemistry analyses. It is implemented in Prolog, and compiles rules into Prolog predicates. This paper justifies and clarifies the programming paradigm that was used in Pro.M.D. Furthermore a semantically correct usage and interpretation of default rules is developed.

2 Syntax of Mini-RBL

We introduce a small rule-based language Mini-RBL, which can also be seen as a subset of the Pro.M.D-language. Every program consists of a set of rules of the form "IF <cond> THEN <action>".

```
<cond>   ::= <b-expr>
<b-expr> ::= NOT <b-expr> | <b-expr><b-op> <b-expr>
             | F | T | UNKNOWN | <c-expr> | <b-var>
             (known? <b-expr>)| (known? <n-expr>)
<b-op>   ::= AND | OR | EQUIV | IMPL
<c-expr> ::= <n-expr><n-cmp><n-expr>
<n-cmp>  ::= GT | GE | EQ | LE | LT
<n-expr> ::= <numbers> | <n-expr><n-op><n-expr>
             | UNKNOWN | <n-var>
<n-op>   ::= + | - | * | /
<action> ::= <b-var> := <b-expr> | <n-var> := <n-expr>
```

<b-var> and <n-var> are variable names, and <numbers> are the usual numbers. For ease of notation, we write "x := t" for rules of the form "IF T THEN x := t".

Syntactically, UNKNOWN is a constant. Its meaning is that the value exists, but is not known. The values of the Boolean and arithmetic functions for unknown arguments can be determined in a straightforward way. For Boolean operators, this gives truth tables using three truth values. The usual behaviour of arithmetic operators is that they evaluate to UNKNOWN, if at least one argument is UNKNOWN (cf. section 6). The syntax implies that there is a finite set of global variables. Depending on the purpose some of them may be viewed as input, output or auxiliary.

A *rule-based Mini-RBL program* is a set of rules that corresponds to the syntax above. The execution of a rule based program consists of firing rules, if their condition is true, until rule firing does not modify global variables anymore. If the condition evaluates to unknown or false, then the rule does not fire. This execution of Mini-RBL programs is guided by the principle that the execution sequence of rules should not influence the result. A consequence is that explicit assignment to variables is forbidden. Variables having the value unknown may be overwritten with a value, an assignment of the same value to a variable is also permitted. If a variable is already assigned a value (except unknown), and a further assignment tries to assign a different value, then the program stops with error "conflicting facts". This error is seen as a programming error that is dynamically detected.

3 Semantics of production rules with static global variables

n order to provide a denotational semantics [Sto81] for production rules with static global variables (and thus also for Mini-RBL programs), we model the information content of variables. Therefore, we use an upper, complete semilattice as a semantic domain. $x \leq y$ means that y contains more informatin than x.

There is some related work in the area of asynchronous and parallel processes (see [Gla90] and [GK91]), where an environment using the principle of single-assignment variables is investigated. However, this work does not consider three-valued logic nor a lattice of information content.

Definition 3.1 *An admissable semantic domain D is a partially ordered set, such that*

i.) There is a least element, called unknown in D.

ii.) For $a, b \in D$, if there is some c with $a \leq c$ and $b \leq c$, then there is a unique least element d with $a, b \leq d \leq c$. This element is called least upper bound of a, b (denoted $lub(a, b)$).

iii.) Furthermore for every increasing chain $a_1 \leq a_2 \leq \ldots$ of elements, there exists a least upper bound a with $a_i \leq a$ for all i.

iv.) There are two elements $false, true \in D$*, such that* $false$ *and* $true$ *are maximal elements, and* $\{x \in D | x < false\} = \{x \in D | x < true\} = \{unknown\}$.

D can be described as a complete upper semilattice that contains the domain of a three valued logic (see [Häh93], [Urq86]). Furthermore there is no greatest element in D. A motivation for the omission of a greatest element is that the domain of boolean values could only be extended if some algebraic rules are sacrificed.

The ordering on D^n is the product ordering $(v_1, \ldots, v_n) \leq (w_1, \ldots, w_n)$ iff $v_i \leq w_i$ for all i. This product ordering satisfies conditions i)-iii) of Definition 3.1. In order to avoid dealing with partial functions, we extend the domain D^n to be a lattice by a top element $\top$ (which could be interpreted as inconsistent knowledge) that is greater than all other elements. The corresponding set with the extended ordering is denoted by $(D_n)^\top$. Elements that are not equal to $\top$ are called proper elements.

Definition 3.2 *Let D be an admissable semantic domain. A function $\varphi : (D_n)^\top \to (D_n)^\top$ is* monotonic*, iff $\varphi\top = \top$ and $v \leq w \Rightarrow \varphi v \leq \varphi w$. A function $\varphi : (D_n)^\top \to (D_n)^\top$ is* strongly monotonic*, iff it is monotonic and $v \leq \varphi v$ for all $v \in (D_n)^\top$.*

We do not enforce that functions are continuous in the sense of lattice theory.

Definition 3.3 *Let D be an admissable semantic domain. Let Φ be a set of functions $\varphi : (D_n)^\top \to (D_n)^\top$. A* fair *sequence of applications of Φ is a sequence $\varphi_1, \varphi_2, \ldots$ with $\varphi_i \in D$ such that every function in Φ occurs infinitely often in this sequence.*

Let $\Phi : (D_n)^\top \to (D_n)^\top$ be a finite set of functions. For $w_0 \in (D_n)^\top$, we inductively define a set W with $w_0 \in W$, and if $w \in W$, then $\varphi(w) \in W$ for all $\varphi \in \Phi$. We abbreviate this set as $\Phi^*(w_0)$.

Lemma 3.4 *Let D be an admissable semantic domain and let $\Phi : (D_n)^\top \to (D_n)^\top$ be a finite set of strongly monotonic functions. Let $W := \Phi^*(w_0)$ for some $w_0 \in (D_n)^\top$. Consider a chain $c_0, c_1, c_2, \ldots$ with $c_0 = w_0$ that is derived with a fair sequence $\varphi_1, \varphi_2, \ldots$ of applications of functions $\varphi_i \in \Phi$, such that $c_i = \varphi_i c_{i-1}$ for $i = 1, 2, \ldots$.*

Then the limit of c_i is a least upper bound of W and vice versa.

This limit element is also denoted as $lub(\Phi^*(w_0))$

Proof. Note that by the strong monotonicity criterion, $(c_i)_i$ is an increasing chain.

1.) First we show that for every element in W there is some greater element c_i in the chain. We make induction on the number of applications. Assume that for some $w \in W$, we have an element $c_k \geq w$. Now consider φw. Since the chain is fair, there is an index m in the chain $(c_i)_i$, such that $c_{m+1} = \varphi c_m$,

and $c_m \geq c_k$. This implies $c_m \geq w$ and hence $c_m = \varphi_0 c_{m-1} \geq \varphi_0 w$. Thus we have shown that every element in W is dominated by some element of the chain $(c_i)_i$.

2.) The converse is trivial, since every element in c_i is also in W.

3.) Note that due to the definition of $(D_n)^\top$, the set W always has an upper bound. The limit of the chain c_i is also a least upper bound of W, since $c_i \in W$ and (c_i) dominates every element of W. □

Definition 3.5 *We define* D-programs *on* $(D_n)^\top$.

The operator $IFTHEN_i(cond,t)$ *for* $1 \leq i \leq n$ *accepts two functions as arguments:* $cond : (D_n)^\top \to \{unknown, true, false\}$, *and* $t : (D_n)^\top \to D$. *For* $w = (v_1, \ldots, \ldots, v_n) \in (D_n)^\top$, *then*

$$IFTHEN_i(cond,t)(w) = \begin{cases} lub(w, (v_1, \ldots, v_{i-1}, t(w), \ldots, v_n)) & if\ cond(w) = true \\ w & if\ cond(w) \neq true \end{cases}$$

A D-program *is a set* Φ *of functions of the form* $IFTHEN_i(cond,t)$.

a.) A D-program Φ *is* admissable, *iff all* $\varphi \in \Phi$ *are strongly monotonic.*

b.) Let w_0 *be the input to the program and let* $W = \Phi^* w_0$. *Then there are two cases*

i.) W has only $\top$ *as upper bound. Then the program is* contradictory *on* w_0.

ii.) W has a proper upper bound. Then the output of P is $lub(W)$. *We denote this output as* $[[P]]$.

c.) An element $w \in (D_n)^\top$ *is a fix-point for the D-program* Φ, *iff for all* $\varphi \in \Phi$, *we have* $\varphi(w) = w$.

d.) A D-program Φ terminates *on the input* $w_0 \in (D_n)^\top$, *iff there is some fix-point w for* Φ *in the set* $\Phi^*(w_0)$.

This definition interprets rules as follows. If the condition of the rule is satisfied, then the assignment $v := t$ in the assignment part does not replace the value of the variable v by the value of t, but computes the least upper bound of the old and new value, i.e., $v_{new} := lub(v_{old}, t)$. This conforms with the intuition of information gathering. The interesting point is that on the right hand side, every rule φ locally satisfies the restriction $w \leq \varphi(w)$.

The semantics of a Mini-RBL program is a D-program, where the domain D is constructed from the domain of a three-valued logic, and the numbers, where unknown is the smallest element, and numbers cannot be compared. Every rule is interpreted as a function.

Corollary 3.6 *The result of an admissable D-program is independent of the sequence, in which rules are tried and fired, as long as the sequence is a fair one. This holds for the three possible types of behaviour: Termination with success, termination with failure, and non-termination. In the case of non-termination the program approximates the result.*

Corollary 3.7 *An admissable D-program terminates, if every properly increasing chain in D is finite. An admissable rule-based program terminates on input w_0, if every properly increasing chain starting with w_0 in D^n is finite.*

Proof. If every properly increasing chain in D is finite, then this holds for $(D_n)^\top$. Obviously, if properly increasing chains starting with w_0 are finite in $(D_n)^\top$, then every infinite application

Corollary 3.8 *The result of an admissable rule-based program is independent of the sequence, in which rules are tried and fired, as long as the sequence is a fair one. This holds for the three possible types of behaviour: Termination with success, termination with failure, and non-termination.*

Example 3.9 *The following rules are taken from a rule based knowledge base for the diagnostics of blood coagulation. The result is a certain combination of symptoms.*

IF quick ≤ 0.7	*THEN quick-patho := true*
IF ptt > 40	*THEN ptt-patho := true*
IF ptt ≤ 40	*THEN ptt-normal := true*
IF ptt-patho AND tzt-patho AND tzy-patho AND bzt-patho	*THEN result-combination-I := true*
.....	

Here knowledge is structured in a decision tree and thus is more of a propositional type. The program terminates, since at some point in time, all variables are computed.

Example 3.10 *An example of a non-terminating, but sensible D-program consists of the following two rule to compute the square-root of some number. We assume that values are numbers that are restricted by intervals. The lub-operation in the domain is the intersection of intervals. The functions that extract the values of upper and lower bounds of the intervals are permitted only in the expression on the right hand side of the assignment. We denote the lower and upper bound of an interval variable by the suffixes low and high. Let v be the input variable and q be the desired output. The rules of the program are:*

- $q := [1, v]$
- $q := [(v - {q_{low}}^2)/(q_{low} + q_{high}) + q_{low}, 0.5 * (v/q_{high} + q_{high})]$

These two unconditional rules form a valid program to approximate the square-root of the input v. Since these two rules are unconditional, our interpretation guarantees that the program is admissable.

4 Concurrent Execution of Rules

The parallel execution model of rules in which we have shared memory and rule execution can be in principle sequentialized, is already covered by the previous

paragraph. In the following we consider the more general siuation, where we can model concurrent access in shared memory on the same variable, or with distributed memory, where the same variable may be in the system more than once on different processors. If we consider such a situation, it may not be possible to simulate the execution by a sequentialized rule execution. Since we concentrate on proving correctness of execution, not on efficiency, we will not use a model, where processors and channels are explicit, but a simplified model that only makes the different values at different places of the same variable explicit.

The idea of our model is to assume that every processor has its own memory for all the global variables. Then all operations can be modelled using three kinds of transformations. The first is that two processors communicate and exchange part of their knowledge to construct the *lub* of their values. The second is that some processor fires some rule. The third is that some processor starts a new processor with a copy of its memory.

Definition 4.1 *Let D be an admissable domain. The model for the distributed memory is a multiset M of elements in $(D_n)^\top$. The program is represented by a set Φ of strongly monotonic functions on $(D_n)^\top$. There are three transformation rules for processing:*

i.) Let b_1 and b_2 be two elements from M. Select some index j and replace b_2 by $lub(b_2, (b_{2,1}, \ldots, b_{2,i-1}, b_{1,i}, b_{2,i+1}, \ldots, b_{2,n}))$.
ii.) Take some $\varphi \in \Phi$, some $b \in M$ and replace b by $\varphi(b)$.
iii.) Select some $b \in M$ and add a copy of b to M.

This produces a sequence of computations, consisting of multisets M_i. Such a sequence is called a parallel execution sequence*. For better syntactic manipulations, we could also remember the exact operations that lead from M_i to M_{i+1}, but this is not necessary for our purposes.*

From the sequence $(M_i)_i$, we can define a sequence $(c_i)_i$ as $c_i := lub(M_i)$. We call this the compressed *sequence corresponding to $(M_i)_i$.*

Definition 4.2 *A parallel execution sequence $(M_i)_i$ is* fair *iff for all i, $b \in M_i$ and $\varphi \in \Phi$ there is some j and some $c \in M_j$ such that $c \geq \varphi(b)$.*

The fairness condition is rather natural and forces every function φ to be applied as often as required. It prevents, for example, useless computations which are dominated by copy-operations

Theorem 4.3 *Let D be an admissable semantic domain, let Φ be a set of strongly monotonic functions on $(D_n)^\top$, let w_0 be some element from $(D_n)^\top$ (the input), and let $(M_i)_i$ be a fair parallel execution sequence. Then the corresponding compressed sequence is an ascending chain with the limit $lub(\Phi^*(w_0))$.*

Proof. Let $c_1, c_2, \ldots$ be the compressed sequence corresponding to M_i. Now construct a fair sequential sequence $(b_i)_i$ from M_i as follows:

- $b_1 := c_1$

- If M_{i+1} is constructed from M_i by a lub or copy operation then $b_{i+1} := b_i$.
- If M_{i+1} is constructed from M_i by application of φ, then $b_{i+1} := \varphi(b_i)$.

The sequence $(b_i)_i$ is fair in the sequential sense, since it is also fair in the parallel sense. Now we show that the two sequences (c_i) and (b_i) have the same limit.

1.) $b_i \geq c_i$ for all i: By induction on i. We have $b_1 = c_1$, hence the relation holds. If M_{i+1} is constructed from M_i by a lub or copy operation, then $c_i = c_{i+1}$ and $b_i = b_{i+1}$, hence $b_{i+1} \geq c_{i+1}$. If M_{i+1} is constructed from M_i by an application of a function φ, then $b_{i+1} = \varphi(b_i) \geq \varphi(c_i) \geq c_{i+1}$, since φ is monotonic. The last inequation holds, since $c_i = lub(M_i)$, and $c_{i+1} = lub(M_i \setminus \{d\}, \varphi(d))$ for some d.
Since every φ must occur infinitely often, we can construct a fair sequential sequence that dominates the parallel sequence of compressed environments.

2.) For every i there exists some j such that $c_j \geq b_i$:
By the fairness condition, we can construct a sequence $(d_i)_i$ as follows. We let $d_1 = c_1$ and d_{i+1} the element that is greater than $\varphi(d_i)$ for some $\varphi \in \Phi$. We can choose the same sequence of φ's as for the sequence $(b_i)_i$. Thus we get $d_i \geq b_i$ by induction on i, since $d_1 = b_1$ and $d_{i+1} \geq \varphi(d_i) \geq \varphi(b_i) = b_{i+1}$. Since d_i is contained in some M_{j_i}, we also have $c_{j_i} \geq b_i$

Now we have shown that a fair parallel sequence has the same limit as a fair sequential sequence, hence the limit is equal to $lub(\Phi^*(w_0))$.

Theorem 4.3 shows that if an admissable rule-based program is concurrently executed in a distributed environment, then the result is the same as in the sequential case, if some fairness conditions hold. A termination condition in the parallel execution case could be to stop, if there is some processor that fulfills a termination condition similar to the sequential one, i.e., there is no more program rule that makes progress in the compressed sequence. This condition is not effective as a distributed algorithm. The invstigation of pragmatical termination tests is beyond the scope of this paper. Using stronger conditions, it is possible to show that the knowledge of the whole distributed computation is also eventually available on some single processor

5 Monotonic Operators and Truth-Preserving Conditions

In this section we exhibit some general conditions which are sufficient to ensure the monotonicity of the functions that correspond to rules. The first condition is that the expression t on the right hand side of an assignment must correspond to a monotonic function. This excludes, for example, the use of functions like *known*? in the assignment part, since *known*? is not monotonic. A further condition which we must have is that every *cond* in a rule remains true, after it was evaluated to true. In the following we investigate these requirements in more detail.

Definition 5.1 *A function $\varphi : D^n \to D$ is called* truth-preserving, *iff $v \leq w$ and $\varphi(v) = true$ implies that $\varphi(w) = true$.*

Let $\preceq$ be an extension of the order $\leq$ on D and D^n, such that $x \leq y \Rightarrow x \preceq y$ for $x, y \in D$ and $false \preceq true$ for the boolean values $false$ and $true$, such that $\preceq$ is the smallest such ordering. We will also freely use this notions for tuple-valued functions, in this case every component must satisfy the truth-preserving condition.

Using truth-preserving conditions we can give a rather general condition for a rule to be strongly monotonic:

Theorem 5.2 *Let R = "IF cond THEN $v_i = t$" be a rule. Consider cond as a function: $(D_n)^\top \to D$, and let the assignment be interpreted as $v_{new} := lub(v_{old}, t)$.*

If cond is truth-preserving and t is monotonic, then R is strongly monotonic on $(D_n)^\top$.

Proof. Let φ be a truth-preserving condition and let $v \leq w$. Then $(cond(v), cond(w))$ $\in \{(unknown, unknown),\ (unknown, false),\ (false, false),\ (unknown, true),$ $(false,\ true),\ (true,\ true)\}$. In the first three cases the function φ is the identity. In the last three cases truth-preservation requires that $v \leq\ lub(w, t(v))$ or $lub(v, t(v)) \leq lub(w, t(w))$. The two equations hold, since $v \leq w$ and $t(v) \leq t(w)$.

It is instructive to consider the forbidden cases that $(cond(v), cond(w))$ is $(true, false)$ or $(true, unknown)$. In these cases the inequation $lub(v, t(v)) \leq w$ is required, which holds only in the very restricted case that $t(v) \leq w$.

Lemma 5.3
- *The functions $\vee$, $\wedge$ and $\neg$ are monotonic.*
- *the functions $+, -, *, /$ and arithmetic comparisons $<, \leq, =, \geq, >$ are monotonic, where division by zero results in a global error, i.e., $\top$.*

Lemma 5.4 *The following functions are $\preceq$-monotonic:*

i.) tupling, projection, and composition of $\preceq$-monotonic functions: $D^n \to D^m$.
ii.) the known?-function that is false for unknown, otherwise true
iii.) The function proved?, that is true for true, otherwise false.
iv.) The functions $\vee$ and $\wedge$.
v.) The function uuf, that is false for true, otherwise unknown.
vi.) In the context of interval arithmetic, the function exact(d, I), that is true if the interval I is less or equal d.

The following lemma describes some truth-preserving functions that are permitted as conditions in admissable rules.

Lemma 5.5 *The following functions $\varphi : D^n \to \{unknown, false, true\}$ are truth-preserving:*

i.) $\preceq$-monotonic functions.

ii.) Monotonic functions

iii.) The composition $\lambda x.f(g(x))$ of a $\preceq$-monotonic function $f : D^k \to D$ and a monotonic function $g : D^h \to D^k$.

Proof. We prove only the last part, the other ones are trivial. Let $f : D^k \to D$ be $\preceq$-monotonic and $g : D^h \to D^k$ be monotonic. Let $v \leq w$ be elements in D^h. Then $g(v) \leq g(w)$, hence also $g(v) \preceq g(w)$. Since f is also $\preceq$-monotonic, we have $f(g(v)) \preceq f(g(w))$, hence the composition is truth-preserving.

The function $\neg$ is monotonic, but not $\preceq$-monotonic and not truth-preserving. The same holds for logical implication. The function *known*? is not monotonic, but $\preceq$-monotonic and truth-preserving. The function $\lambda(x).\neg(known?(x))$ is also not truth-preserving.

Corollary 5.6 *The following conditions in Mini-RBL are truth-preserving.*

- *An arbitrary condition, where only the boolean operators $\vee, \wedge, \neg$ are used*
- *A condition, which can be seen as the composition of a function that is composed solely of $\vee, \wedge$, and known?, and another tuple-valued Boolean function that is monotonic.*

An example for a truth-preserving condition is $known?(x) \wedge (known?(y) \vee \neg(x > y))$, which is composed as $known?(x_1) \wedge (known?(x_2) \vee x_3) \circ (x, y, \neg(x > y))$. It is not hard to syntactically check these conditions during compilation of a Mini-RBL program.

Example 5.7 *These examples show that composition is not compatible with truth-preservation and with $\preceq$.*

- *The composition of two truth-preserving functions may be not truth-preserving. For example, let f be defined such that $f(unknown) = false$ $f(false) = unknown$ and $f(true) = true$. Then f is truth-preserving. The composition $known?(f(x))$ is not truth-preserving, since $known?(f(false)) = false$ and $known?(f(unknown)) = true$.*
- *the composition of a monotonic with a $\preceq$-monotonic function may be not $\preceq$-monotonic. The standard example is $\neg(known?(x))$.*

6 Application to fuzzy values

If we apply the method to numbers which are known to be in some interval, then the semi-lattice is simply the lattice of intervals ordered by the superset ordering, and the empty interval is omitted. It is not hard to determine the operation of functions on intervals given functions on elements, the functions are simply lifted to sets. This method ensures that functions on intervals are monotonic. If an interval contains only one element, then the value can be considered unique, and hence known. For example, this gives the equations $true \vee unknown = true$ and $[1, 2] + [4, 5] = [5, 7]$. It is also possible to have a better approximation, if

the whole expression is used to compute its value. For example $x - x$ can result in 0, regardless of the value of x. The same holds for $x \vee \neg x$ which can always be replaced by $true$. This view of the semantics of unknown permits to use all algebraic laws to evaluate expressions. However, different strength of algebraic manipulations may result in different outcomes of the same program. For example, in the case of interval arithmetic, the distributive law in combination with a modular definition of $*$ and $+$ may decrease the exactness of approximations. If a compiler uses such algebraic manipulations or simplifications, then this should only be done, if a better approximation results.

Such an improvement can also exhibit hidden "conflicting facts", for example, let the rules be $\{A := 2, A := (B - B) * A\}$ Without algebraic simplifications, the Mini-RBL program started with $A = B = unknown$ terminates with $A = 2, B = unknown$. After algebraic simplification, the execution is halted with "conflicting facts".

In the interval case an operator corresponding to $known?$ would be $Exact?(e, x)$, which is $true$, if the length of the interval for x is not greater than e. It can be used in a similar way as the $known?$ -operator. Suppose, we have a function IL, that computes the length of an interval. This function is not monotonic, since $length[0,3] = 3$, $length[1,2] = 1$, we have $[0,3] \leq [1,2]$, but not $3 \leq 1$ in the information ordering $\leq$. Thus the functions $known?$ and $exact?$ are truth-preserving and can be used in conditions, if they are not under a negation. The function IL, and the extraction of upper and lower boundary of the intervals is not monotonic and should be illegal in conditions.

7 The Usage of Defaults

If expressions like $\neg(known?(x))$ are used to implement defaults, then the order of rule execution plays a role. For example, consider the two rules

IF $NOT(known?(v_i))$ THEN $v_i := 1$

IF $NOT(known?(v_i))$ THEN $v_i := 2$

If we use them without precaution, then the result may be $v_i = 1$, or $v_i = 2$, depending on the sequence of execution, and there also is no possibilitiy to detect this error dynamically, since once one rule has fired, both rules become inapplicable. As we have seen in the last paragraph, the construction $\neg(known?(x))$ is not truth-preserving, and thus does not satisfy the conditions of theorem 5.2 .

To remedy this we permit such default-rules, split the program into a "normal part" and a "default part". The normal execution of the program consists of firing the rules in the normal part, and to fire the rules in the default part only in the case, where the normal part of the program has terminated. One must ensure, however, that the defaults are not conflicting, since otherwise, one would reintroduce a dependence of the result on the order of execution. The example above is such a set of conflicting defaults. A correct interpretation would be to keep the variable values fixed, collect all results of the default rules, and add the lub of all the outcomes, with a certain bad chance of getting as answer "conflicting facts".

Now there are two alternatives, either the program could go again into the usual cycle of firing normal rules until the program terminates, or the program stops after firing the default rules. There are some reasons (see below) to prefer the second one.

Another, more subtle method would be to also permit the application of default rules if the program status ensures that the default conditions are true, and cannot be changed to *false* by other rules. In a practical implementation, this can be done by stopping the program interpretation, inspecting the current program status, and then executing all the defaults. In order to have a clean loop breaking method such that the results do not depend on the time of the break, it must be ensured that the application of defaults behaves as if the limit was already reached. This method of using defaults ensures that programs have a unique result.

If the compiler makes use of algebraic laws to simplify expressions, and the program uses defaults, then the outcome of the whole program may change, since for example, the first execution of all non-default rules may produce *unknown* for a certain variable, but after algebraic simplification, the value becomes known. In the first case, a default may be applicable, wheras in the second case the default rules are no longer applicable. In order to give an example, consider the following rules:

A := B*C
IF $not(known?(A))$ THEN $A := 1$

A critical input is $B = 0, C = unknown$. The usual implementation of $*$ is that A remains $unknown$. The program terminates, and then the default is fired and gives $A = 1$. If the algebraic law $0 * x = 0$ is applied, then A is assigned 0, and the default rule is not applicable. For sensible programming, this means that defaults should only be used as a last resort, for example for preparing a readable form of the output, but not as a normal programming device.

8 Comparison with other Approaches

In this paragraph we want to comment on the possibilities to implement the Mini-RBL method in other paradigms. The issue is adequacy of such an embedding.

(1) *Constraints* (cf. [Win92]) can express dependencies between variables and perform the same computations for a subset of the Mini-RBL programs. Constraints are not directed, whereas rules are directed in the sense, that a single rule has a fixed input and as output a new information on the variable in the assignment. Thus Mini-RBL programs cannot be used in a backward direction. The advantage of Mini-RBL programs is that also cyclic dependencies can be encoded and lead to a stable output or to an approximation of the output, whereas constraint systems require a direct method for finding the solution, or if constraint propagation is used, some kind of oscillation may occur, if cyclic dependencies are given.

(2) *Logic programming* . ([CM91],[Llo84],[SS86]) In its pure form, the execution of a logic program amounts to computing new facts, thus approximating the minimal model in a fixed-point computation. Global variables as in Mini-RBL can be encoded as predicates and corresponding facts. The manipulation must be done using Assert, Retract (or setval, getval), and a possible computation of least upper bounds would be a method to modify asserted facts. This manipulation methods are not a possible in a pure subset of Prolog, hence it is not clear, how the existence of a minimal model can be guaranteed.

(3) *Functional programming.* ([BW88],[Dav92]) Pure functional programming as a basis for Mini-RBL has several problems. Non-mutable variables are fine, however, even modifying global variables from unknown to a value is not possible in pure functional languages. If the functional language permits some non-pure techniques in its I-O-processing (cf. [SA92]), then the emulation becomes more adequate. However, the possible cyclicity of variable dependencies in Mini-RBL cannot be directly translated into the functional paradigm, and must be checked in order to prevent infinite loops. This translation is studied in detail in [Moo93]. The polymorphic type-discipline of functional programming is easily adaptable to extensions of Mini-RBL, as long as the information-order is compatible with the type structure.

(4) *Rules as implications.* If production rules are seen as two-valued logical implications, then the corresponding rule based programs have the properties of being independent of the sequence of execution. In order to simulate our programming paradigm using implicational rules, the logic must be at least a three-valued one. In this case, there must be some non-determinism (see the examples in Section 7), and the results of this paper are applicable.

9 Conclusion

We have shown that some restrictions on the usage of production systems permits to have a clean and nice semantics of PSG-programs. The consequences are that the result of a program is independent of the sequence of the execution of rules. A further result is that parallelization of rule execution, even in a distributed environment, leads to the same results as sequential execution.

A clear distinction can be made between legal and illegal usage of several operators, like the $known?$, $proved?$, and $exact?$- operator. Furthermore, the usage of $not(known?(x))$ for defaults can be permitted if the interpretation of default rules differs from the interpretation of other rules.

Acknowledgements

I would like to thank Bernhard Pohl, Frank Puppe, Sven-Eric Panitz and Marko Schütz for reading a draft of this paper.

References

[BFKM85] L. Brownston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison-Wesley, Reading, 1985.

[BS84] B.G. Buchanan and E.H. Shortliffe. *The MYCIN experiment of the Stanford heuristic programming project.* Addison-Wesley, Reading, 1984.

[BW88] Richard Bird and Phil Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[CM91] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1991.

[Dav92] A.J.T. Davie. *An introduction to functional programming systems using Haskell.* Cambridge University Press, Cambridge, 1992.

[GK91] R. Glas and A. Knoche. Semantische Fundierung der asynchron prozeduralen Sprache DONUTS im AEM. Forschungsbericht 1991/3, Technische Universität Berlin, Fachbereich Informatik, Germany, 1991. in German.

[Gla90] R. Glas. Ein abstraktes Environment Modell (AEM) zur Beschreibung parallelen und asynchronen Verhaltens . Forschungsbericht 1990/3, Technische Universität Berlin, Fachbereich Informatik, Germany, 1990. in German.

[Häh93] R. Hähnle. *Automated Deduction in Multiple-Valued Logic.* Oxford University Press, Oxford, 1993.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

[Moo93] Marcus Moos. Übersetzung von Pro.M.D.-Wissensbasen in funktionale Sprachen. Master's thesis, FB Informatik, J.W.Goethe-Universität Frankfurt, 1993.

[PT88] B Pohl and Chr. Trendelenburg. Pro.M.D.- A diagnostic expert system shell for clinical chemistry test result interpretation. In *Meth. Inform. Med.*, volume 27,3, pages 111–117, 1988.

[SS86] L. Sterling and E. Shapiro. *The Art of PROLOG.* MIT-Press, Cambridge Mass., 1986.

[Sto81] J.E. Stoy. *Denotational semantics, the Scott-Strachey approach to programming language theory.* MIT-Press, Cambridge Ma., 1981.

[TP90] Chr. Trendelenburg and B. Pohl. *Pro.M.D. Medizinische Diagnostik mit Expertensystemen.* Thieme-Verlag, Stuttgart, Germany, 1990. in German.

[Urq86] A. Urquhart. Many-valued logic. In *Handbook of philosophical logic*, volume III, pages 71–116, 1986.

[Win92] P.H. Winston. *Artificial Intelligence.* Addison-Wesley, Reading, third edition, 1992.

This article was processed using the LaTeX macro package with LLNCS style