Computers Math. Applic. Vol. 20, No. 9/10, pp. 73–82, 1990
Printed in Great Britain.

0097-4943/90 $3.00 + 0.00

SUBSET-EQUATIONAL PROGRAMMING IN INTELLIGENT DECISION SYSTEMS

B. JAYARAMAN
Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14221, U.S.A.

Abstract—Subset-equational programming is a paradigm of programming with subset and equality assertions. The underlying computational model is based on innermost reduction of expressions and restricted associative-commutative (a–c) matching for iteration over set-valued terms, where ∪ is the a–c constructor. Subset assertions incorporate a "collect-all" capability, so that the different subset assertions matching a goal expression and the different a–c matches with each subset assertion are all considered in defining the resulting set of the goal expression. We provide several examples to illustrate the paradigm, and also describe extensions to improve programming convenience: negation by failure, relative sets and quantities. We also discuss the use of subset-equational programming for intelligent decision systems: the rule-based notation is well-suited for expressing domain knowledge and rules; subset assertions are especially appropriate in backchaining systems like MYCIN, which performs an exhaustive depth-first consideration of subproblems before arriving at some decision; and restricted a–c matching is very convenient for querying attributes of objects in such systems, by relieving the concern for the next ordering of attributes.

1. INTRODUCTION

The term "logic programming" is often taken to be synonymous with predicate-logic programming [1], owing to the latter's simple semantics and the efficient implementations available for Prolog [2, 3]. In recent years, other forms of logic programming have been proposed, most notably equational-logic [4], higher-order-logic programming [5], and constraint-logic programming [6]. This paper focuses on a logic programming paradigm based on equations and subset assertions, called *subset-equational programming*, the goal of which is orthogonal to earlier efforts in that it provides a rigorous basis for programming with sets. Existing approaches, such as the "setof" constructs of Prolog systems, are not supported by a simple underlying logic although they are very useful in practice. In subset-equational programming, a program is a collection of two kinds of assertions:

$$f(terms) = expression;$$

$$f(terms) \supseteq expression.$$

The declarative meaning of an equality (resp. subset) assertion is that, for all its ground instances, the function f operating on the argument ground terms is equal to (resp. superset of) the ground set denoted by the expression on the r.h.s. By providing subset assertions with a *collect-all* capability, the meaning of a set-valued function f operating on ground terms is equal to the union of the respective sets defined by the different subset assertions for f. The top-level query is of the form

$$?\, expr$$

where *expr* is a ground expression. The meaning of this query is the ground term t such that $expr = t$ is a logical consequence of the *completion* of the program, i.e. augmenting all subset assertions defining some function with equality assertions that capture the *collect-all* capability of these subset assertions.

Computation with these assertions is a process of "replacing equals by equals". Both equality and subset assertions are oriented left-to-right for rewriting. Because arguments to functions are ground terms, function application requires one-way matching, rather than unification. Actually, the matching operation is associative-commutative (a–c) matching because of the presence of the ∪ constructor. The multiple matching substitutions arising for a–c matching effectively serve to iterate over the elements of sets, thus permitting many useful set operations to be stated non-recursively, with attendant advantages during implementation. We show how a restriction in

the use of ∪ in program assertions supports both clear programming as well as efficient 75implementation. The associated matching algorithm is referred to as *restricted a–c matching*.

In this paper, we present two aspects of subset-equational programming.

(i) Programming concepts and constructs. First, we explain the two key concepts underlying the paradigm: the completion and restricted a–c matching. We then discuss useful declarative extensions to the basic paradigm, in order to make the expression of programs more convenient: *negation by failure*, *relative set constructs* and *quantifiers over sets*. Negation by failure is more tractable in this paradigm than in predicate-logic programming because all arguments to functions are ground. Relative set constructs, which were first introduced by Turner [7], can be systematically translated into this paradigm. The computational model of subset-equational programming can also be readily adapted to support universal and existential quantifiers over sets. With these extensions, several programs can be stated very concisely. Another extension, not discussed in this paper, is *set closures*, which are useful in defining the smallest set satisfying certain constraints, e.g. various transitive-closure operations.

(ii) Use in intelligent decision systems. There are several aspects of subset-equational programming that lend themselves to the design of intelligent decision systems. Like other logic languages, the rule-oriented notation of subset-equational programming is appropriate for expressing domain knowledge. An important facet of decision making is the consideration of a set of alternative solutions. Towards this end, the collect-all capability of subset assertions and the provision of matching are particularly well-suited. These capabilities are especially appropriate in backchaining systems like MYCIN [8], which performs an exhaustive depth-first consideration of subgoals before arriving at some decision. In problems where exhaustive search can be expensive, it is possible to prune potentially unproductive lines of search by simply returning the empty set as the solution for these cases. Restricted a–c matching is also useful in querying attributes of objects; by viewing the attributes as an unordered *set*, the programmer is freed from keeping track of the positions of various attributes in the object.

Subset-logic programming is also applicable to the design forward-chaining production systems. The efficiency of production systems like OPS5 [9] is considerably enhanced because its underlying implementation remembers partial matches across consecutive production applications—a capability not directly provided in our paradigm. Thus, subset-equational programming is appropriate where the number of cases tend to be small. Because the rules of expert systems can usually be partitioned so that the system operates in several phases [8, 10], we expect this restriction may not be too severe in practice. One potential advantage of subset-equational programming compared with production system languages is that it provides greater flexibility in control, because of the provision of function composition, conditionals, recursion, etc.

We note that subset-equational programming does *not* support unification, and is not meant to be an alternative to predicate-logic or constraint-logic programming. A unified language with both capabilities can be designed, but this issue is beyond the scope of this paper. Indeed, we see logic programming ultimately as a combination of programming with relational, equality and subset assertions—this idea has been developed in another paper [11]. In this paper, we explicate the uses of the latter two features. An experimental language called SEL (set-equation language) embodying most of these ideas has been implemented. We have also developed the formal semantics of these ideas [11]. This paper concentrates on programming issues, and only briefly addresses implementation issues. We illustrate the application of subset-equational programming to intelligent decision systems by showing aspects of some typical systems can be encoded in this paradigm.

The rest of this paper is organized as follows: Section 2 explains the completion and restricted a–c matching; extensions to the basic paradigm and examples are presented in Section 3; Section

4 illustrates the use of the paradigm for specifying search and other aspects in programming 76intelligent decision systems; Section 5 is devoted to conclusions and areas of further work.

2. SUBSET-EQUATIONAL PROGRAMMING: CONSTRUCTS AND CONCEPTS

We first specify the syntactic structure of term and expression.

$$term ::= atom \mid variable \mid \phi \mid \{term\} \mid term \cup term \mid constructor(terms)$$

$$terms ::= term \mid term, terms$$

$$expr ::= term \mid \{expr\} \mid expr \cup expr \mid constructor(exprs) \mid function(exprs)$$

$$exprs ::= expr \mid expr, exprs.$$

We will refer to a term as a *set* if it has one of the set constructors, ϕ, { } or ∪, at its outermost level—these are the only set constructors—otherwise, we will refer to the term as an *element*. The constructor ϕ is the empty set, and { } is the singleton-set constructor. The constructor ∪ is stands for set-union, and has its usual properties, such as associativity, commutativity, etc. A *ground term* is a term without any variables in it. Informally, terms correspond to data objects, and we consider only finite terms in this paper.

We next discuss the two key concepts in subset-equational programming: the completion, and restricted a–c matching.

2.1. Completion

We "complete" all subset assertions defining some operation with an equality assertion that captures the collect-all capability of subset assertions. There are two aspects to the completion:

> *(i) The collect-all assumption.* If a set-valued expression e is such that $e \supseteq s_1, \ldots, e \supseteq s_n$, and it is determined that there are no other known subsets for e according to the given program, then the collect-all assumption allows us to infer $e = \cup_{i=1,n} s_i$.
>
> *(ii) The emptiness-as-failure assumption.* This assumption effectively allows us to discard all failing reductions when collecting the different subsets of a set. There are two aspects to the emptiness-as-failure assumption: (a) the value of $\{expr\}$ is ϕ if *expr* reduces to an expression with non-constructors, and (b) applying a (non-constructor) set valued function f_s to terms that don't match any of the l.h.s. of assertions for f_s yields ϕ.

We illustrate both aspects of the completion by a simple example. Note that our lexical convention in this paper is to begin atoms with an uppercase letter and variables with a lowercase letter.

f(Bob) = Mark, m(Bob) = Mary,
f(Ann) = Mark, m(Ann) = Mary,
f(Mark) = Joe, m(Mark) = Jane,
p(x) ⊇ {f(x)}
p(x) ⊇ {m(x)}.

The collect-all assumption effectively supplements p by the assertion p(x) = {f(x)}∪{m(x)}. Thus, for example, p(Bob) = {f(Bob)}∪{m(Bob)} = {Mark}∪{Mary} = {Mark, Mary}.

By the emptiness-as-failure assumption, we have, for example, p(Mary) = {f(Mary)}∪{m-(Mary)} = $\phi \cup \phi = \phi$, because f(Mary) and m(Mary) are irreducible and f and m are not constructors.

2.2. Restricted a–c matching

The a–c matching problem may be stated as follows: given two terms t_1 (possibly non-ground) and t_2 (ground), some constructors of which may be a–c commutative, is there a substitution θ such that $t_1\theta =_{ac} t_2$ (where $=_{ac}$ means "equality modulo the associative and commutative equations")?

This problem was first posed by Plotkin [12] and has since been studied extensively in the literature (see Ref. [9] and references therein). For example, if $t_1 \equiv u \cup v$ and $t_2 \equiv \{\text{Mark, Mary, Joe, Jane}\}$, we will obtain 16 different substitutions for θ such that $t_1\theta =_{ac} t_2$, corresponding to the different ways of splitting the four-element set into two subsets—a typical match would be {u←{Mary, Jane}, v←{Mark, Joe}}. On the other hand, if $t_1 \equiv \{x\} \cup t$ and $t_2 \equiv \{\text{Mark, Mary, Joe, Jane}\}$, we will obtain four different substitutions, corresponding to the different ways of selecting one element from the set and its corresponding remainder—a typical match would be {x←Joe, t←{Mark, Mary,Jane}}.

We will use the notation {x|t} to refer to a non-empty set, one of whose elements is x and the remainder of the set is t. That is, {x|t} is syntactic sugar for {x}∪t. The case when all set patterns are restricted to the form {*term*|*term*}, where *term* does not use ∪ explicitly, is of special interest, because it is amenable to a more efficient implementation. Henceforth in this paper, we disallow *explicit* use of the ∪ constructor in program assertions—we show in Section 3 how set union can be defined using program assertions. Basically, this restriction permits iteration over the elements of a set, rather than iteration over the subsets of a set. While some expressive convenience is sacrificed by this restriction, most practical cases are unaffected. We refer to the associated matching operation as *restricted a–c matching*, which are discussed in greater detail in Ref. [13].

We should note that the equality $=_{ac}$ is based only on the associative and commutative properties, but not the *idempotent* property. Thus, for example, matching {x|t} with {Mark, Mary} cannot yield the substitution {x←Mark, t←{Mark, Mary}}.The reason for disallowing the idempotent property during matching is to avoid a potential infinite loop in recursive definitions where t appears, on the r.h.s. of the rule (see, for example, the perms definition in Section 3). Furthermore, because a singleton set such as {Mark} is represented internally as {Mark}∪ϕ, it can match {x|t} yielding {x←Mark, t←ϕ}, thus the *identity* property is not explicitly required during matching.

The multiple a–c matches arising from the use of patterns such as {x|t} provide a convenient and efficient way of iterating over the elements of a set. Continuing the example from the previous subsection, we may define the set of ancestors of some individual as follows:

$$\begin{aligned} &\text{anc}(x) = \text{allanc}(p(x)),\\ &\text{allanc}(s) \supseteq s,\\ &\text{allanc}(\{x|t\}) \supseteq \text{anc}(x). \end{aligned}$$

For example, to find the ancestors of Bob, we evaluate anc(Bob), which reduces to allanc(p(-Bob)). The evaluation of nested expressions occurs innermost-first; hence the above expression effectively reduces to allanc({Mark, Mary}). Because allanc is defined by two subset assertions, both these assertions are considered in reducing allanc({Mark, Mary}). The result from the first assertion is {Mark, Mary}. When matching allanc({Mark, Mary}) with the l.h.s. of the second assertion defining allanc, both a–c matches are considered, namely, {x←Mark, t←{Mary}} and {x←Mary, t←{Mark}}. The r.h.s. of this assertion is then separately evaluated for each of these matches, and the union of these sets (along with that from the first assertion) is defined as the value for allanc({Mark, Mary}). Thus allanc({Mark, Mary}) effectively reduces to {Mark, Mary, Joe, Jane}. The following points should not noted in the above process:

(i) In general, duplicates must be eliminated when taking the above union, but we showed in Ref. [13] how this check can be deferred for a particular argument when the function *distributes over union* in this argument.

(ii) The case when the argument set to allanc is empty is correctly handled, i.e. allanc(ϕ) = ϕ. We clearly have allanc(ϕ)⊇ϕ, by the first assertion for allanc. We also have allanc(ϕ)⊇ϕ by the second assertion for allanc, because of case (b) of the emptiness-as-failure assumption.

(iii) With reference to the second rule for allanc, because the variable t is not used on the r.h.s., considerable space and time can be saved by not constructing the remainder-set for it. We provide the notation_to refer to the "don't-care variable", as in Prolog, so that the programmer can indicate such cases explicitly.

Before proceeding to extensions of the basic paradigm, we briefly address the confluence of program assertions. Of special interest is the case where the a–c constructor ∪ appears (implicitly) on the l.h.s. of such assertions, as illustrated by the definition of the function below:

$$\text{size}(\phi) = 0,$$
$$\text{size}(\{x|t\}) = 1 + \text{size}(t).$$

Stated as a syntactic condition, we require that:

(i) the l.h.s. of each equality assertion not overlap with any other assertion

and

(ii) when set patterns of the form $\{t_1|t_2\}$ occur in equality assertions, the result should be independent of which one of the potentially many a–c matches is selected.

Other less restrictive conditions are possible, but we shall assume the above conditions, for the sake of specificity. Note that a subset assertion may overlap with other subset assertions. Thus, the definition of the size function is legal; its result is independent of which particular match is selected in the second assertion. However, the definition below for set-to-list conversion is not legal, because the resulting list does in general depend on which particular match is chosen at each recursive call

$$\text{set2list}(\phi) = [\,],$$
$$\text{set2list}(\{x|t\}) = [x|\text{set2list}(t)].$$

3. EXTENSIONS

While the basic paradigm described in the previous section has the capability of any universal language for expressing computation, it does not possess features needed to make practical programming convenient. We therefore propose in this section extensions that are compatible with the basic paradigm described in the previous section.

3.1. Negation as failure

We have seen that $\{e\}$, where e is irreducible expression, is equal to the empty set ϕ, by the emptiness-as-failure assumption. In a similar manner we can augment the primitive if then else, so that in addition to the two usual cases,

$$\text{if true then e1 else e2} = \text{e1},$$
$$\text{if false then e1 else e2} = \text{e2},$$

a negation as failure rule can be incorporated as follows:

$$\text{if } e \text{ then e1 else e2} = \text{e2}, \text{ if } e \text{ is irreducible and not boolean.}$$

Note that the negation as failure rule in the context of subset-equational is more tractable than in the context of predicate-logic, because all expressions that appear as arguments of an if then else must be ground. In the context of predicate-logic, the negation as failure rule, to preserve soundness [14], can be applied only to negative goals that are ground, and hence non-ground negative goals would have to be deferred until they became ground—a provision not found in most practical systems because of its overheads in implementation.

3.2. Relative set abstraction

It is possible to provide the notation of relative sets as a form of syntactic sugar in subset-equational programs. Relative sets were first introduced in functional programming by Turner in his languages KRC and Miranda [7]—actually, these languages supported lists without duplicates rather than sets. Below we present a slightly simplified form of that presented in Ref. [7].

$$\{\textit{term}: \textit{generators}; \textit{condition}\}.$$

The above construct defines a set of *terms* by first generating candidate elements using the *generators*, and eliminating those that do not satisfy the stated *condition*. (The *condition* is actually

optional, and is assumed to be true when omitted.) Each *generator* is of the form *gen-variable* $\in$ *set-expression*, where all the *gen-variables* within a relative set construct must be distinct from one another. Furthermore, the *set-expression* of a *generator* may not use its defining *gen-variable* or any *gen-variable* to its right. However, *term* may use any *gen-variable* from the relative set construct. Because subset-equational programs deal with finite sets, we assume that the *generators* define finite sets.

It is quite straightforward to translate these relative set expressions into subset-equational programs. However, the transformation of subset assertions into the relative set notation is not as direct because set-patterns can have repeated occurrences of variables on the l.h.s., as in the definition of set intersection, and also may refer to remainders of sets, as in the definition of the permutations of the elements of a set (see Section 4 for their definitions). These capabilities of subset-equational programs offer more expressive convenience than the relative set notation.

3.3. Quantifiers over sets

It turns out that there is a simple extension of subset-equational programming to include quantifiers. Observe that a *subset* assertion is basically taking the *union* of the respective *sets* returned by the expression on its r.h.s., for each match on its l.h.s. By replacing the subset assertion by an *ifall* assertion, union by *logical-and*, and sets by *booleans*, we simulate a *universal quantifier*. Similarly, replacing the subset assertion by an *ifone* assertion, union by *logical-or*, and sets by *booleans*, we simulate a *existential quantifier*. We therefore provide the ifall and ifone constructs as extensions to the basic paradigm. For example, the predicate $p(s) = (\forall x \in s)q(x)$ can be defined as

$$p(\{x|\_\})\text{ifall } q(x).$$

Similarly, the predicate $p(s) = (\exists x \in s)q(x)$ can be defined as

$$p(\{x|\_\})\text{ifone } q(x).$$

Operationally, the ifall rule defines the predicate p to be true if for all a–c matches of the head, the r.h.s. of p reduces to true. If there are no matches, the result returned is true. The ifone rule, on the other hand, defines the predicate p to be true if for any one match the r.h.s. of p reduces to true. If there are no matches, the result returned is false. Because of our negation as failure rule, an irreducible expression that is not a boolean is taken to be false.

In general, an operation can be defined by multiple ifall assertions (the reduction of the r.h.s.'s for all matches in all assertions must yield true), or multiple ifone assertions (the reduction of the r.h.s.'s for any one match in any one assertion must yield true). However, a combination of the two assertions may not be used in defining some operation. This restriction is placed to avoid a potential ambiguity: for example, if p were defined by one ifall and one ifone assertion and there were no match for some argument, the result should be true by the ifall assertion, but false by the ifone assertion.

3.4. Set operations

We conclude this section by illustrating the succintness provided by subset assertions, restricted a–c matching, as well as the foregoing extensions in defining familiar operations on sets:

$$\begin{aligned}
&\text{member}(h, \{h|_\}) = \text{true},\\
&\text{crossproduct}(s1, s2) = \{[x|y]\colon x \in s1, y \in s2\},\\
&\text{intersect}(\{x|_\}, \{x|_\}) \supseteq \{x\},\\
&\text{union}(s1, s2) \supseteq s1,\\
&\text{union}(s1, s2) \supseteq s2,\\
&\text{diff}(\{x|t\}, s2) \supseteq \text{if member}(x, s2) \text{ then } \phi \text{ else}\{x\}\\
&\text{subset}(\{x|_\}, s2) \text{ ifall member}(x, s2),\\
&\text{perms}(\phi) = \{[\,]\},\\
&\text{perms}(\{x|t\}) \supseteq \text{distr}(x, \text{perms}(t)),\\
&\text{distr}(x, \{h|t\}) \supseteq \{[x|h]\},\\
&\text{powerset}(\phi) = \{\phi\},
\end{aligned}$$

powerset({x|t}) = distr2(x, powerset(t)),
distr2(x, s) ⊇ s,
distr2(x, {y|_}) ⊇ {{x|y}}.

Note that the first six operations shown above are all stated non-recursively. It is possible to compile these definitions so that no recursive calls occur even during execution. Note that the if then else in the diff function makes use of *negation by failure*; so also does the subset predicate. The perms and powerset examples illustrate recursion in conjunction with a–c matching. In the second assertion for powerset, the result is independent of the particular a–c match chosen. The latter two examples show the expressiveness of subset-equational programs over the relative set constructs.

4. APPLICATIONS IN DECISION-MAKING

We begin with an example illustrating the use of subset-equational programming in specifying search, which is an important aspect of problem solving. We then show how subset assertions and the restricted a–c matching are used in formulating forward-chaining and backward-chaining expert systems.

4.1. Specifying search

The formulation of the N Queens program in subset-equational illustrates how a search may be specified and the multiple solutions gathered into a set. The algorithm below basically places a queen on each successive column, beginning from column 1, as long as each new queen placed is safe with respect to all queens in preceding columns. A solution is found if a queen can thus be placed on all columns.

```
solve(n) = queens(1, φ, n)
queens(col, safeset, n) =
                if eq(col, n + 1) then {safeset}
                else placequeen(col, {1, . . . n}, safeset, n)
placequeen(col, {row|_}, safeset, n) ⊇
                if safe([col|row], safeset)
                then queens(col + 1, {[col|row]|safeset}, n)
                else φ
safe([c1|r1], {[c2|r2]|_})ifall(r1 ≠ r2) and (abs(c1 − c2) ≠ abs(r1 − r2)).
```

The second argument in the call to placequeen, viz. the set {1 . . . n}, enumerates the row positions in each column. If a particular row-column position is not safe, placequeen returns the empty set, thereby pruning this line of search. The function safe specifies the safety condition, and makes use of the universal quantifier ifall—note that ifall would return true in case there is no match on the l.h.s., as would be the case if the second argument were the empty set. We assume the language has the usual complement of arithmetic operations.

4.2. Querying attributes and forward-chaining expert systems

To illustrate the role of subset-equational programming in forward-chaining expert systems, we illustrate how to encode a typical rule from the graduate course advisor (GCA), a multi-phase expert system for advising graduate students in computer science on schedule and course selection [10]. This system has four phases (each with its own rule base): schedule-length determining phase, course-planning phase, course-evaluation phase, and schedule-evaluation phase. For simplicity, suppose that there are three types of elements, Student, Course, and Suggestion, with attributes as shown below:

```
Student    : CareerGoal, LanguageSkills, Recreation,
Course     : Number, Skill, Recreation, Prerequisite,
Suggestion : MightPass, MightEnjoy.
```

If we assume an OPS5-like forward-chaining system, we can encode the *working memory* as a *set* of pairs of the form

Element :: *AttributeSet*,

where :: is a binary infix constructor, and the attributes of each element also as a *set* of pairs of the form

Attribute :: *Value*.

A typical rule in the course-planning phase might be as follows:

> "If a student has recreation x and there is a course with recreation x and number y then the student might enjoy course y."

The above rule might be encoded as follows:

```
course_plan({Student :: {Recreation :: x|_},
             Course :: {Recreation :: x,Number :: y|_}
            |_})
             ⊇{Suggestion :: {MightEnjoy :: y}}.
```

We use the notation $\{x, y|\_\}$ analogous to the list notation of Prolog to refer to a set which has elements x, y and possibly others. This notation is just a short-hand for $\{x|\{y|\_\}\}$. The rule course_plan takes one argument, the working memory, inspects it for the presence of a Student element and Recreation element with certain properties, and, if successful, returns a singleton-set containing a Suggestion element. Several such course_plan rules could be defined using subset assertions, so that several suggestions could be obtained in a single inference step. The module that invokes the course_plan function would be responsible for combining the Suggestion elements into the new working memory—the details of which we don't present here.

The convenience offered by set patterns here is that it effectively relieves the programmer from worrying about the positions of various attributes in an element, and also from the locations of elements in working memory. Obviously, if there were large numbers of rules and working elements, the cost of matching can become prohibitive if performed naïvely. Our point here is simply that the matching and collection of results can be expressed quite easily with set patterns, and the process can be tolerable for partitioned rules bases and partitioned working memories—an assumption that seems to be valid in many practical instances. In general, conditions in production rules could use matching operations other than equality. Such conditions would have to be placed in the body of a subset assertion.

4.3. All solutions and backchaining expert systems

It turns out that sets and the collect-all capability of subset-equational programming are well-suited to backchaining systems such as MYCIN. There are several areas in the MYCIN system where the paradigm is applicable. For example, a *context* can be viewed as a set of *attribute* :: *hypotheses* pairs, where *hypotheses* itself is a set of *value* :: *certainty-factor* pairs. Taken from Ref. [8], a simple rule such as,

> "IF the gram stain of the organism is gramneg, and the morphology of the organism is rod, and the aerobicity of the organism is anaerobic, THEN there is suggestive evidence (0.6) that the identity of the organism is bacteroides."

can be encoded as a subset-equational assertion as follows.

```
identity({Gram :: Gramneg, Morph :: Rod, Air :: Anaerobic|_})
                    ⊇{Bacteroides :: 0.6}.
```

That is, we can associate assertions for each different kind of *clinical parameter*, such as identity. The argument to identity is the *current context* in MYCIN. Note that several such subset assertions can be used to provide different hypotheses on the identity of the organism; all these rules will be invoked to obtain the collective set of hypotheses. In general, premises in MYCIN might make use *special functions* and many require the values of other clinical parameters to be determined in

a goal-directed manner. Such premises would be incorporated in the body of a subset assertion rather than in the head.

We give below a simplified sketch of the overall control cycle of MYCIN. We assume that all functions not shown, e.g. ask_user, lab_data, etc. are primitive for the purpose of this exposition. The function evaluate is responsible for invoking assertions such as the one shown above for identity.

```
monitor(rule_no) = evaluate(rule_no, find_all(clinic_pars(rule_no)))
find_all({cpar|_}) ⊇ findone(cpar)
findone(cpar) = if lab_data(cpar) then ask_user(cpar)
                else monitor_all(rules_for(cpar))
monitor_all({rule_no|_}) ⊇ monitor(rule_no).
```

This program illustrates the use of subset assertions in defining the collect-all behavior of the MYCIN inference system. Finally, we should not that most of MYCIN's special functions, such as same2, notsame2, etc. can be directly expressed in terms of relative sets and the quantifiers.

5. CONCLUSIONS

This paper has presented an informal description of subset-equational programming. This paradigm is not meant as an alternative to predicate-logic [1] or other forms of logic programming [4–6], but instead a complementary approach whose goal is to deal with sets in a logically rigorous way. The two main concepts of subset-equational programming are the completion and restricted a–c matching. The former incorporates a collect-all assumption, for taking the union of all sets from different subset assertions defining an operation, as well as an emptiness-as-failure assumption, for discarding unsuccessful reductions. Restricted a–c matching provides an efficient way of iterating over set elements. We also showed how this paradigm can be extended with useful, yet declarative, features to make programming even more convenient. The three extensions discussed in this paper were negation by failure, relative sets and quantifiers.

Besides providing an introduction to this novel paradigm, this paper also shows its relevance for the design of intelligent decision-making systems. Basically, decision making consists of evaluating sets of alternatives. Although the premises–conclusion format works well for encoding domain knowledge, the collection of multiple results is generally handled in an *ad hoc* manner in most expert systems. It is here that subset assertions play a useful role. Its use is especially appropriate in MYCIN-like systems that exhaustively collect all hypotheses in a depth-first manner. Another observation is that many expert systems do not make use of the full power of unification; in other words, (one-way) matching seems to suffice in many cases. In fact, we have found set matching to be more useful than we had first anticipated, as we have illustrated through our examples.

While finalizing this paper, we became aware of an interesting use of sets in intelligent decision-making based on a set-covering inference model [15]. This model has been used in medical diagnosis, and differs from other expert system models in that it is not rule-based. Its knowledge-base consists of a set of disorders (e.g. diseases), each of which has a set of known manifestations (e.g. symptoms). A manifestation may be caused by more than one disorder, so that an m-to-n "causes" relation exists between these two sets. Given a set of input manifestations, the diagnosis problem is to find the smallest set of disorders whose collective manifestations include the input set. Its inference mechanism is a hypothesize-and-test process, in which each input manifestation is used to refine the set of disorders that are hypothesized as the final diagnosis. This paradigm for expert systems also appears to be well-suited for subset-equational programming.

In order to show its practicality, we have developed a working implementation of subset-equational programming [13, 16]. In this implementation restricted a–c matching and the control strategy are compiled along the lines of the WAM [3]. Important optimizations that were incorporated were last-call optimization, avoiding checks for duplicates through the knowledge of functions that distribute over union in particular argument positions, and avoiding unnecessary copying during restricted a–c matching. This implementation includes all the features mentioned in this paper, except relative sets, and is available free of cost from the author.

There are several further extensions possible to the basic paradigm of subset-equational programming: we are considering the relaxation of our current restrictions on finite sets, first-order terms, and innermost reduction. At a more strategic level, we have been exploring the integration of subset-equational programming and Horn-logic [11]. The use of sets in a logic programming language is suggestive of meta-programming, since set theory is the meta-language in which the semantics of logic programs is expressed [14]. The development of this view would probably lead to ideas closely related to those found in higher-logic programming [5], which provides a more rigorous basis for meta-programming than the meta-logical constructs found in Prolog.

Acknowledgements—I am grateful to Bruce T. Smith for suggesting the relevance of MYCIN-like systems for subset-equational programming, and for making available the OPS5 code for a subset of the GCA. This research was supported by grant DCR-8603609 from the National Science Foundation

REFERENCES

1. R. A. Kowalski, Predicate logic as a programming language. *Proc. IFIP 74*, pp. 556–574. North-Holland, Amsterdam (1974).
2. D. H. D. Warren, F. Pereira and L. M. Pereira, Prolog: the language and its implementation compared with LISP. *SIGPLAN Not.* **12** (8), 109–115 (1977).
3. D. H. D. Warren, An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, Calif. (1983).
4. M. J. O'Donnell, *Equational Logic as a Programming Language*. MIT Press, Cambridge, Mass. (1985).
5. D. Miller and G. Nadathur, Higher-order logic programming. *3rd Int. Conf. Logic Progmng*, London, pp. 448–462, Jul. (1986).
6. J. Jaffar and J.-L. Lassez, Constraint logic programming. *14th ACM POPL*, Munich, pp. 111–119 (1987).
7. D. A. Turner, Miranda: a non-strict functional language with polymorphic types. *Conf. Functional Progmng. Lang. Comp. Arch.*, Nancy, pp.1–6, Sep. (1985).
8. B. G. Buchanan and E. H. Shortliffe, *Rule-based Expert Systems*. Addison-Wesley, Reading, Mass. (1984).
9. L. Brownston, R. Farrell, E. Kant and N. Martin *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, Mass. (1985).
10. M. Valtorta, B. T. Smith and D. W. Loveland, The graduate course advisor: a multi-phase rule-based expert system. *IEEE Wksh Knowledge-based Systems*, Denver, Colo, pp. 53–57 (1984).
11. B. Jayaraman and D. A. Plaisted, Programming with equations, subsets and relations. *N. Am. Conf. Logic Progmng*, Cleveland, Ohio, Oct. (1989).
12. G. Plotkin, *Building-in Equational Theories*, *Machine Intelligence* (Eds D. Michie and B. Meltzer), Vol. 7, pp 73–90. Edinburgh Univ. Press (1972).
13. B. Jayaraman and A. Nair, Subset-logic programming: application and implementation. *5th Int. Logic Progmng Conf.*, Seattle, Wash., pp. 843–858, Aug. (1988).
14. J. W. Lloyd, *Foundations of Logic Programming*, 2nd edn. Springer, Berlin (1987).
15. J. A. Reggia, D. S. Nau and P. Y. Wang, Diagnosis expert systems based on a set covering model. *Developments in Expert Systems* (Ed. M J Coombs), pp. 35–58.
16. A. Nair, Compilation of subset-logic programs. M S. Thesis, University of N. Carolina at Chapel Hill, Dec. (1988).