1. INTRODUCTION

The original vision of logic programming called for using predicate logic as a programming language [35,48]. Prolog only partially realizes this vision, since it has many features with no corresponding feature in first-order predicate logic, and also fails to realize some features of predicate logic. Perhaps the main benefit of the system suggested in this paper, hereafter called Eqlog, is the way it combines the technology of Prolog (its efficient implementation with unification and backtracing) with functional programming (in an efficient first-order rewrite rule implementation) to yield more than just their sum: logical variables can be included in equations, giving the ability to find general solutions to equations over user defined abstract data types (ADTs). In addition, generic (i.e., parameterized) modules become available with a rigorous logical foundation; Eqlog also has a subsort facility that greatly increases its expressive power.

The many advantages that have been claimed for logic programming, including simplicity, clarity, understandability, reusability and maintainability, are all compromised to the degree that the logic underlying the programming language is not a pure logic. Thus, it is highly desirable to extend the logic in such a way as to encompass a greater range of programming language features, provided that one can preserve reasonable efficiency. Our approach is based on many-sorted first-order Horn clause logic with equality, and also draws on some results from the theory of rewrite rules about “narrowing” [32,17] to get a complete implementation of equality; this approach can be seen as arising from the work of Plotkin [40], Robinson & Wos [43], and Slagle [45] on combining equations and resolution. Of course, we use unification [42], and our evaluation algorithm for Eqlog is an extension (by narrowing) of the standard Prolog evaluation mechanism (e.g., [10,8]) and can be seen as a special kind of resolution. It can also be seen as a kind of “universal unification” that solves a wide variety of “logical constraints.”
This paper shows that many useful "impure" features of Prolog can be made "pure" by taking many-sorted first-order Horn clause logic with equality as the formal foundation. Combining many-sorted logic with modules permits a convenient treatment of data abstraction, inspired by our experience with the rewrite rule-based language OBJ [25,24]. In fact, both Prolog and OBJ are "sublanguages" of Eqlog; i.e., both Horn clause programming and modular first-order functional programming are provided. There are, of course, many cases where computational intuition is essentially functional rather than relational, and for such cases it is convenient to use functional programming. In Eqlog, functions and predicates are sharply distinguished, and functional notation, including composition, is available for functions. Also, Eqlog provides both evaluation of ground terms by term rewriting and evaluation of predicates by the usual Prolog method. In addition to its use of many-sorted logic, Eqlog also has a powerful subsort facility; we indicate both a model theoretic semantics and a first-order equational axiomatization of our subsort concept.

However, there is nothing essential about the use of many sorts, and those who do not like strong typing can use unsorted Horn clause logic with equality and still apply most of what this paper offers to ordinary Prolog. In particular, since our approach to generic modules relies on very general results from the logical foundations of specification languages [21] and the theory of rewrite rules, it should apply to other variants of logic programming such as Concurrent Prolog [44] as well as to ordinary unsorted Prolog.

Many authors have attempted to combine logic and functional programming. For example, Komfeld [34] gives several interesting examples (some of which inspired examples in this paper), but gives no theoretical justification for his implementation of equality; in fact, it is not complete (i.e., it can sometimes fail to find the right answer when one does exist). Moreover, the ADT and object oriented facilities are less general than might be desired, since neither modularity nor strong typing are provided, and functions are not carefully distinguished from predicates. In addition to combining logic and functional programming, the Funlog language [47] also provides infinite data structures, lazy evaluation, and nondeterminism; however, no formal logic is given for these features, either model theoretic or proof theoretic, and Funlog's "semantic unification" algorithm is also incomplete. A natural deduction technology is used by Hansson, Haridi, and Tarnlund [28] to implement a superset of Horn clause logic with equality that includes negation and explicit universal quantifiers; the system also handles infinite data structures by lazy evaluation. However, we are not aware of a formal semantic theory for the language. Finally, Bellia, Degano, and Levi [2] describe FPL, a logic programming notation for what is essentially a functional programming language; a rigorous semantics is given, but it does not support logical variables, or solve systems of equations containing them. Also, the model theory is completely different from that of first-order logic.

Completeness is a very important property for the algorithm underlying a logic programming language; without it, a user cannot be sure that what he writes will eventually produce the result that the logic says it should! However, there seems to be a trade-off between generality and efficiency: highly expressive complete languages are not necessarily efficient. Thus, the challenge of designing a good logic programming language is to reach a suitable compromise between efficiency and generality (with completeness).
2. THE UNDERLYING LOGIC

First-order Horn clause logic without equality underlies ordinary Prolog. But there are many other logics, some of which have distinct advantages. Thus, first-order logic with equality supports user definable ADTs, and many-sorted logic gives strong typing. Pure equational logic can also give rise to programming languages. One such language is OBJ [25, 24], which supports user definable ADTs by regarding the defining equations as rewrite rules; Hoffman and O'Donnell [30] describe another such language. Other languages that are systematically based upon some kind of logic are CDS [4] which is based on the lambda calculus, pure Lisp which is based on the logic of recursive functions, and the language of the "Prolog technology" theorem prover of Stickel [46], which is the full first-order predicate calculus.

We now briefly review many-sorted Horn clause logic with equality. Here, one has a set $S$ of sorts, plus signatures $\Pi$ and $\Sigma$ which give the predicate and function symbols, respectively. Each predicate symbol $Q$ has an arity which is a string of sorts that serves to indicate the number and sort of arguments that it can take. Thus, arity $s_1s_2s_3$ indicates that $Q$ takes three arguments, of which the first and third must be of sort $s_1$, and the second of sort $s_2$. Similarly, each function symbol has a rank consisting of a string $w$ of sorts (its arity) and a sort $s$ (its value sort). Equality enters as a distinguished binary predicate symbol $=$ for each sort $s$, which we will write with infix notation, and usually without the subscript. Sentences are Horn clauses in the usual sense, but may involve the distinguished equality predicate; i.e., they are of the form

$$P : - P_1, \ldots, P_n.$$ 

where each $P$ and $P_i$ is a positive atomic formula of the form $Q(t_1, \ldots, t_n)$, and each $t_i$ is a term of sort $s_i$ when $s_1 \cdots s_n = w$ is the arity of $Q$; these terms may include variables, which will of course be "logical variables"; also $P$ and/or any $P_i$ may be an equation, since it can use an equality predicate. $P$ is called the head of the clause, and $P_1, \ldots, P_n$ constitute its tail.

A simple Eqlog program for calculating the population density of countries is

$$density(C) = pop(C)/area(C).$$

In ordinary prolog, this would be given by the clause

$$density(C, D) :- pop(C,P), area(C,A), D \text{ is } P/A$$

using the impure is feature, which is a weak analog of Lisp's eval function. In case, for some reason, one really wants density, pop and area to be predicates rather than functions, one can still write

$$density(C,D) :- pop(C,P), area(C,A), D = P/A$$

in Eqlog. Furthermore, the three tail clauses can be given in any order. Assuming that pop and area are functions, we can add facts to the database with assertions like

$$pop(china) = 800.$$
(in millions!) instead of the more awkward
\[ \text{pop(china,800)}. \]

Similarly, we can compute the temperature in Fahrenheit from that in Centigrade
by the usual formula,
\[ f(C) = (9/5) \cdot C + 32. \]
where \( f \) is a rational (abbreviated \texttt{rat}) valued function and \( C \) is a \texttt{rat}-sorted variable
(assuming the rationals are available; otherwise, one could use floating point
numbers). However, we can still write the query
\[ f(C) = 77. \]
and get the right answer
\[ C = 25. \]
(but unless a suitable output simplifier is provided, one may get a large unreduced
fraction as an answer).

We now indicate how to get the rationals from the integers by using equality. In
fact, one can define equality of rational numbers just as usual in mathematics,
\[ X/Y = Z/W :\} Y \cdot Z = X \cdot W. \]
where / is a rat-valued function symbol denoting division (the denominator must be
nonzero), and \( X, Y, Z, W \) are variables of sort \texttt{int} (integer). The above clause (with a
little syntactic sugar for declarations, as shown in Section 5) will enable an Eqlog
user to define the rationals; by contrast, [34] uses logical variables in a nonobvious
way.

For typical interesting cases, we can build in decision procedures that yield most
general unifiers using techniques like Gaussian elimination (see Section 3), so that
Eqlog can automatically handle most computations that result from this definition of
the rationals. For example, we can define \( \ast \) and \( + \) as usual in mathematics, by
\[ (X/Y) \cdot (Z/W) = (X \cdot Z)/(Y \cdot W). \]
and
\[ (X/Y) + (Z/W) = ((X \cdot W) + (Z \cdot Y))/(Y \cdot W). \]
and then get the expected behavior.

Logical precision requires specifying the intended models. For first-order many-
sorted logic with equality, these have one set for each sort \( s \), together with a
predicate among those sets for each predicate symbol, such that the sorts of its
arguments match its arity: similarly, with a function among those sets corresponding
to each function symbol, such that the argument and values match those of the sorts
in its rank. It is also assumed that equality predicates are always interpreted as
\textit{actual identity} in the models. In addition, there may be a number of sort, function,
and predicate symbols that have a fixed interpretation. Thus, for reasons of
efficiency, it is desirable to build in the integers; in terms of model theory this means
taking a fixed interpretation for the sort \texttt{int} and for all the associated functions and
predicates.

\[ ^3 \text{Compare this with [34], which uses functions like \%times having bizarre definitions that seem to}
\text{involve putting arbitrary Lisp functions inside clauses.} \]
A model $M$ satisfies a clause of the form
\[ P ::= P_1, \ldots, P_n. \]
iff for every assignment $\alpha$ of values in the model $M$ to variables in the clause (such that sort restrictions are satisfied), $\alpha P$ holds in $M$ whenever $\alpha P_i$ holds in $M$ for all $i$. A model $M$ satisfies a set $\mathcal{C}$ of clauses iff it satisfies every clause in $\mathcal{C}$. But for programming, we are not really interested in all models satisfying all the clauses in $\mathcal{C}$; rather, we are interested in the standard model of $\mathcal{C}$, which we now explain.

Given signatures $\Sigma$ and $\Pi$ of function and predicate symbols (respectively) and a set $\mathcal{C}$ of Horn clauses (including equality predicates), the standard model, denoted $T_{\Sigma,\Pi,\varphi}$, has as its elements equivalence classes of ground terms under the equivalence relation
\[ t \equiv t' \iff \mathcal{C} \vdash t = t', \]
where $\vdash$ is the provability relation for many-sorted first-order logic with equality. Let $[t]$ denote the equivalence class of $t$ under this relation. Then function symbols are interpreted in the usual way, and predicate symbols are interpreted by:
\[ P([t_1], \ldots, [t_n]) \text{ is true in } T_{\Sigma,\Pi,\varphi} \iff \mathcal{C} \vdash P(t_1, \ldots, t_n); \]
and is false otherwise. $T_{\Sigma,\Pi,\varphi}$ is like the Herbrand universe, except that it consists of equivalence classes of terms instead of individual terms.

The basic facts about this situation are as follows: $T_{\Sigma,\Pi,\varphi}$ satisfies $\mathcal{C}$; also, if $M$ is any other model satisfying $\mathcal{C}$, then there is a unique $\Sigma,\Pi$-homomorphism $h: T_{\Sigma,\Pi,\varphi} \rightarrow M$ (where a $\Sigma,\Pi$-homomorphism is a many-sorted function preserving the function and predicate symbols in the signatures). A model such that there is a unique $\Sigma,\Pi$-homomorphisms from it to any other, is called an initial $\Sigma,\Pi$-model [18, 26, 27]. Initial $\Sigma,\Pi$-models are unique up to $\Sigma,\Pi$-isomorphism. This approach is, we think, an attractive alternative for the "minimal Herbrand model" approach; it characterizes the construction at a level of abstraction that is independent of representation details, and has many pleasant properties, as shown by

**Theorem 1.** Let $\mathcal{C}$ be a set of Horn clauses with equality, using function and predicate symbols from the signatures $\Sigma$ and $\Pi$, respectively. Then:

1. $T_{\Sigma,\Pi,\varphi}$ satisfies $\mathcal{C}$;
2. if $M$ is any other model satisfying $\mathcal{C}$, there is a unique $\Sigma,\Pi$-homomorphism $h: T_{\Sigma,\Pi,\varphi} \rightarrow M$ (where a $\Sigma,\Pi$-homomorphism is a many-sorted function preserving the function and predicate symbols in the signatures), i.e., $T_{\Sigma,\Pi,\varphi}$ is an initial $\Sigma,\Pi$-model satisfying $\mathcal{C}$;
3. any model initial among those satisfying $\mathcal{C}$ is isomorphic to $T_{\Sigma,\Pi,\varphi}$;
4. two $\Sigma$-terms denote the same element of $T_{\Sigma,\Pi,\varphi}$ iff they can be proved equal using the clauses in $\mathcal{C}$; and finally
5. for $P$ a predicate symbol and $t_1, \ldots, t_n$ terms in variables $Y_1, \ldots, Y_m$, one has
\[ \mathcal{C} \vdash (\exists Y_1, \ldots, Y_m) P(t_1, \ldots, t_n) \]
iff there is a substitution $\sigma$ sending the $Y_i$ to ground terms such that $P([\sigma(t_1)], \ldots, [\sigma(t_n)])$ is true in $T_{\Sigma,\Pi,\varphi}$.

All this is closely related to the so-called "Closed World" assumption for the initial model $T_{\Sigma,\Pi,\varphi}$. In the language of Burstall & Goguen [7], this model has "no junk"
and "no confusion." No junk means that every element of the model can be denoted by a term using the given function symbols. No confusion means that a predicate holds of some elements iff it can be proved to hold using the axioms; in particular, two elements are identified iff they can be proved equal using the given axioms. In fact, these two conditions together are equivalent to initiality. It is worth noting that the full first-order predicate calculus does not admit initial models in the above sense. This appears to be closely connected with the difficulties associated with extending both "circumscription" [38] and Prolog to full first-order logic.

Coercions are treated by Kornfeld [34] in a complex manner involving the use of impure Prolog predicates like var. However, coercions can be handled much more easily using subsorts. Thus, we can just declare

\[
\text{int} < \text{rat}
\]

and then assert

\[
\text{N} / \text{I} = \text{N}.
\]

to get the desired effect. The basic idea is to use a logic having a partial ordering on its set of sorts. Actually, subsorts are traditional in theorem proving and go back at least to Herbrand [29]. The model theoretic semantics of subsorts simply requires that if \( s < s' \) then the set that interprets \( s \) is a subset of the set that interprets \( s' \). The proof theory is a little more complicated; see Section 7.1. Subsorts are easily implemented in a logic programming language, and in fact have long been part of the OBJ language [24]; sorts form an acyclic graph under the subsort relation, so that a form of "multiple inheritance" is provided.

3. SOLVING EQUATIONS OVER BUILT-IN SORTS

Assume that we are given a signature \( \Sigma \) of function symbols and a reachable \(^5\) \( \Sigma \)-model \( A \). Now let \( E \) be a set of \( \Sigma \)-equations over a set \( X \) of variables. Then a ground solution of \( E \) in \( A \) is an assignment \( \alpha \) from the variables in \( X \) to values in \( A \) such that \( \alpha(E) \) is satisfied in \( A \). Now letting \( T_\Sigma(Y) \) denote the \( \Sigma \)-terms with variables from \( Y \), we define a solution of \( E \) in \( A \) to be an assignment \( \sigma \) from \( X \) to terms in \( T_\Sigma(Y) \) such that \( \alpha(\sigma(E)) \) is satisfied in \( A \) for every assignment \( \alpha \) from \( Y \) to \( A \). A complete solution of \( E \) in \( A \) is a set \( L \) of solutions such that every solution of \( E \) in \( A \) is a substitution instance of one in \( L \); i.e., such that for any solution \( \tau \) [from variables \( X \) to \( T_\Sigma(Y) \)] there is a solution \( \sigma \) in \( L \) and a substitution \( \rho \) from the variables in \( Y \) to \( T_\Sigma(Y) \) such that \( \tau = \rho(\sigma) \). (Note that these definitions do not require most general substitutions.)

For example, let \( \mathbb{N} \) be the natural numbers with only the function \( + \), so that \( \Sigma \) contains elements of \( \mathbb{N} \) as constants and \( + \). Let us consider just linear equations, regarding \( 3X \) as an abbreviation for \( (X + X + X) \). Thus, the equations

\[
\begin{align*}
3X + Y + 2Z &= 1 \\
X - 2Y &= 3
\end{align*}
\]

have a ground solution \( \sigma(X) = 7, \sigma(Y) = 2, \sigma(Z) = -11 \), and have a complete solution given by \( \sigma(X) = 3 + 4V, \sigma(Y) = 2V, \sigma(Z) = -4 - 7V \), where \( V \) is a para-

---

4 This really means \( \text{int} \leq \text{rat} \), but we use the simpler notation because it can be done with just one keystroke.

5 This means that every element of \( A \) is denoted by some \( \Sigma \)-ground term, i.e., there is no junk.
ter variable. It is a general theorem that any set of linear equations over the integers has either no solution, or else a complete solution consisting of just one substitution.

One can also get a complete solution over the naturals for any equation of the form

\[ a_1 \cdot a_2 = a_3 \cdot a_4, \]

where \( a_1, a_2, a_3, a_4 \) are either variables or constants, and are not necessarily distinct. For example,

\[ X^2 = 2Y \]

has the complete solution \( \sigma(X) = 2V, \sigma(Y) = 2V^2 \), and

\[ XY = 0 \]

has a complete solution \( \{\sigma, \tau\} \), where \( \sigma(X) = V, \sigma(Y) = 0, \tau(X) = 0, \tau(Y) = V \),

while

\[ X^2 = ZW \]

has an infinite complete solution.

Note that for some theories over built-in sorts like \texttt{int}, things are very inefficient to decide, while others have no complete decision procedure. For example, a user should not expect to get an answer very soon from

\[
\text{fermat} : \leftarrow X \cdot\cdot N + Y \cdot\cdot N = Z \cdot\cdot N, N \geq 3.
\]

since the query will halt iff Fermat's Last Theorem is false. Similarly, there is no decision procedure for arbitrary equations in \( + \) and \( \cdot \), by Matijasevic's negative solution to Hilbert's tenth problem [37,11]. Section 7.3 considers solving equations over user defined sorts and Section 4 describes a sublogic within which equations, over user defined sorts can be made solvable with reasonable efficiency.

Complete solutions do not necessarily exist; also, just because a complete solution exists does not mean that it is recursively enumerable, i.e., that there is an algorithm that will produce all the substitutions in it. Moreover, even if a recursively enumerable complete solution exists, the algorithm can still fail to terminate when faced with a case for which no solution exists. Let us say that we have a \textit{totally complete solution} in case there is an algorithm that will explicitly fail if there is no solution, and otherwise will enumerate a complete solution. Similarly, let us say we have a \textit{r.e. complete solution} in case there is an algorithm that will enumerate a complete solution when there is one, and say we have a \textit{finite solution} if we have a totally complete solution that is always finite. More algorithmically, we will assume that \( \text{SOLN}(E) \) produces substitutions in the solution of \( E \), if any exist, one at a time on request until there are no more.

A further desirable property of a solution \( L \) of \( E \) in \( A \) is that it should be \textit{most general}, in the sense that for any solution substitution \( \sigma \), there is a unique member \( \tau \) of \( L \) and a unique substitution \( \rho \) such that \( \rho \circ \tau = \sigma \). It can be shown that any two most general solutions are essentially the same. Unfortunately, there are cases where totally complete solutions exist, but no most general solution exists [16]. However, the examples given above do have most general solutions; in fact, the (nonground) solutions given are most general.

The most classical case in the present context, of course, is that where the model is the set of terms over some signature \( \Sigma \) and the functions are just those in \( \Sigma \). Then
Robinson's unification algorithm gives a finite solution (it is totally complete and always consists of just one most general unifier). A more complex case is that of integer linear programming, because of the inequality predicates that are involved. Solution algorithms for built-in sorts provide Eqlog with all the power of so-called "constraint languages," and narrowing provides a powerful capability for handling equational and logical constraints over user-defined ADTS.

4. COMPUTING IN HORN CLAUSE LOGIC WITH EQUALITY

This section considers sublogics of Horn clause logic with equality within which equations over user definable ADTs can be solved. We begin with a basic logic and then extend it; most logic programming applications seem to be included. The basic sublogic assumes that all clauses are of two types, either a pure equation, or else a clause whose head is not an equation. Let $E$ denote the set of equations and $P$ the set of Horn clauses whose head clause is not an equation; thus $V = E \cup P$. To unify two positive atomic formulae, say $Q_1(t_1, \ldots, t_n)$ and $Q_2(u_1, \ldots, u_m)$, we must of course have that $Q_1$ is $Q_2$, the arity $w_1$ of $Q_1$ is the arity $w_2$ of $Q_2$ so that $n = m$ and the sort of $t_i$ equals that of $u_i$ and we must also solve the system

$$t_1 = u_1, \ldots, t_n = u_n$$

of simultaneous equations modulo the equations given in $E$; this is called $E$-unification. Under this assumption about the structure of clauses, those in $P$ can have no influence on finding an $E$-unifier.

The computation algorithm of ordinary Prolog has been described clearly but informally by Warren [49]:

To execute a goal, the system searches for the first clause whose head matches or unifies with the goal. The unification process finds the most general common instances of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then activated by executing in turn, from left to right, each of the goals of the body (if any). If at any time the system fails to find a match for a goal, it backtracks, i.e., it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsideres the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

The assumption that the set $E$ of clauses decomposes into disjoint sets $E$ of equations and $P$ of predicate-headed clauses has the desirable effect of isolating the solution of equations into a separate $E$-unification algorithm SOLN, which is then called by the Prolog search algorithm described above. Of course, SOLN must be called in a way that can be backtracked and is fair, in the sense that every substitution gets tried. This gives a semidecision procedure that may not halt; but if SOLN is r.e. complete, then a general proof of completeness of the algorithm can be given along standard lines [40,1]. As shown in Section 7.3, a complete SOLN using narrowing always exists if the equations in $E$ are confluent and terminating.

We now consider some extensions of the basic case $V = E \cup P$. There do not seem to be any essential difficulties with these extensions; rather, the problem is one of verification, since we lack any good way to check completeness in the general case where equality predicates can occur anywhere in clauses. For example, at present the
confluence property can only be established under significant restrictions even for the relatively mild extension where \( δ \) is a set of conditional equations [41]. For the general case of arbitrary Horn clauses with equality predicates, the basic algorithm remains the same, but now SOLN and the Prolog-like algorithm may have to call each other recursively. Several of the examples given in this paper depend upon extensions of this kind.

Here is one relatively simple extension; another is discussed in Section 7.3, after we have defined narrowing. Say that a predicate \( P \) (which may be an equality \( = \)) directly depends on another \( Q \) if there is a clause with \( P \) as the predicate of its head and \( Q \) as a predicate in its tail; let \( \text{depends} \) be the transitive closure of direct dependence. Then we conjecture that our evaluation algorithm works provided that no equality predicate depends on itself. For example, it is reasonable to define \( =_{\text{rat}} \) in terms of \( =_{\text{int}} \) since there is no dependence of the clauses defining \( \text{int} \) on those defining \( \text{rat} \).

Notice that user queries can also be equations, to be solved using the SOLN algorithm. In addition, terms can be evaluated, i.e., reduced to normal form. Reduction to normal form is a completely general and very powerful paradigm for functional computations [25]. To illustrate this, if we declare a function symbol \( ! \) on the natural numbers, write the equations

\[
0! = 1, \\
(succ(N))! = succ(N) \times (N!).
\]

and then write an expression like

\[
(3!) \times (3!).
\]

that contains no variables, Eqlog will apply the given equations as rewrite rules, until it can go no further, in this case, producing the answer 36. We could similarly have defined binomial coefficients, Fibonacci numbers, or more interestingly, non-numerical types like \( \text{set}, \text{bag}, \) and \( \text{queue} \) with their associated functions, as will be shown later. More complex functions, such as sorters and parsers, can also be defined; in fact, any general recursive function can be defined equationally [3, 39].

5. USER DEFINED ABSTRACT DATA TYPES

There is much work on providing user defined ADTs in programming languages (e.g., OBJ [24, 19], Clu [36], Ada\(^6\) [13]), and on the logical basis for this in purely equational logic (e.g., [26, 39]). The essential idea is to allow users to introduce modules that define new sorts and their associated functions; it can also be very helpful to have available subsorts and their associated predicates, as we will see. The intended model-theoretic semantics of a module is the standard or initial model of its associated theory in Horn clause logic with equality.

A purely syntactic notion of module has been given for Mprolog in [14]. Let us now give a complete definition for the data type \( \text{rat} \) in proper Eqlog syntax. Note that Eqlog keywords are underlined, that module names are all

---

\(^6\)Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).
capitals, while variable names begin with a capital and that relation, function, and constant names are all lower case. Also note that Eqlog provides built-in types as modules; e.g., the module INT has sort int with subsort nzint of nonzero integers (modules, especially generic modules, are discussed in more detail in Section 6). “Attributes” can be given for operators; e.g., assoc, comm, and idp indicate that a binary operator is associative, commutative, and idempotent, respectively; and id: e indicates that it has e as its identity. The associative and commutative properties of functions can be built into unification algorithms; these attributes also have important syntactic implications.

Eqlog “mix-fix” notation permits any desired ordering of keywords and arguments for operators; this is declared by giving a syntactic “form” consisting of a string of keywords and underbar characters “_”, followed by a “:”, followed by the arity as a string of sorts, followed by “_”, followed by the value sort of the function; if there are no underbars, then the usual parentheses-with-comma notation must be used. Similar conventions are used for predicates. An expression is considered “well-formed” in this scheme iff it has exactly one parse; the parser can interactively help the user to satisfy this condition. Now we define the rationals:

```
module BASICRAT using INT is
  sorts rat
  subsorts int < rat
  fns
    _/_: int,nzint → rat
    _*: rat,rat → rat (assoc comm id: 0)
    _+: rat,rat → rat (assoc comm id: 1)
  vars N,X,Z: int Y,W: nzint
  axioms
    N / 1 = N.
    X / Y = Z / W :- X * W = Y * Z.
    (X / Y) * (Z / W) = (X * Z) / (Y * W).
    (X / Y) + (Z / W) = (X * W) + (Z * Y) / (Y * W).
endmod BASICRAT
```

Here the keyword using indicates that the sorts, subsorts, predicates, functions, and axioms of the listed modules are imported into the module being defined. We refer to the relationship between modules being defined and being used as the using hierarchy. We now enrich BASICRAT to define division and the subsort of nonzero rationals.

---

7The parser is greatly helped if spaces always separate the keywords declared in the form of a function, and this paper follows that convention throughout, but since parentheses are also delimiters, they do need not to be separated by spaces. These syntactic conventions follow those of OBJ [24, 25].
module BAT using BASICRAT is
  sorts nzrat
  subsorts nzrat < rat
  fns _/_: rat,nzrat → rat
  vars X: int Y,Z,W : nzint
  axioms
    nzrat(X / Y) :- m&t(X).
    (X/Y)/(Z/W)=o( * W)/(Y * Z).
endmod BAT

It is easy to define new types in the same style, e.g.,

module SET-OF-INT using INT is
  sorts set-of-int
  fns
    Ø : set-of-int
    { } : int → set-of-int
  preds
    _ ∈ _ : int, set-of-int
    empty : set-of-int
  vars N: int S : set-of-int
  axioms
    N ∈ S :- {N} ∪ S = S.
    empty(S) :- S = Ø.
endmod SET-OF-INT

Although this definition is algebraically elegant, it is actually not the most efficient or convenient approach; a better definition of a generic set module is given in Section 6.

It is a bit tedious to repeat such a definition every time one wants to define sets or bags of some new sort. One way to avoid this tedium is to use polymorphic sorts; but we prefer parameterization, as explained in the next section.

We have already noted that the sorts and subsorts currently defined form an acyclic graph (thus supporting so-called “multiple inheritance”). This motif is repeated at the module level, with another acyclic graph under the using hierarchy. In fact, the subsort hierarchy and the using hierarchy interact, since subsorts are declared inside of modules: At a given node M of the using hierarchy, the set of currently defined sorts is the union of those declared in M with all those declared in nodes below M in the using hierarchy (i.e., all those related to M by the transitive closure of the using relation); similarly, the subsort relation applicable at M is the
union of the subsort declarations in M with those from all modules below M. Thus, the subsort graph of a lower level module is a subgraph of that of a higher level module. (All this has already been implemented in OBJ and has been found very natural and helpful.)

6. GENERIC MODULES

Reusability is a major goal of modern software engineering. In order to achieve this goal, it is necessary that software be broken into components that are as reusable as possible; parameterization is a technique that can greatly enhance the reusability of components [19]. For example, bag-of and set-of, which have caused considerable controversy in the Prolog Digest, can easily be defined as generic abstract data types, and then automatically implemented using rewrite rules. Generic modules also greatly ameliorate the otherwise odious need for defining abstractions whenever they are used. Without some such facility, strong typing would not be tolerable to use in practice. (Interactive syntax also helps greatly, by allowing menu selection instead of keyword typing.)

Before giving details, we consider how to specify a parameterized module's interface, especially the requirements that an actual parameter should satisfy for the instantiation to make sense, expressed in the form of a theory (that is, a set of axioms) that the actual must satisfy. Such a theory may include sort, subsort, predicate and function declarations, saying what the actual parameter must provide to the parameterized module, as well as axioms saying what properties must be satisfied. For example, a generic sorting module might have the theory of quasi-ordered sets as its requirement theory; this means that an actual must provide a designated sort and a binary relation on it that is transitive and reflexive. In Eqlog, this theory is given as follows:

```plaintext
theory QUOSET is
  sorts elt
  preds _ = < _ : elt,elt
  vars A,B,C : elt
  axioms
    A =< A.
    A =< C : ¬A =< B, B =< C.
endth QUOSET
```

Theories are not intended to be used for computation, but only for declaring the properties of interfaces. The idea is that before an instantiation of a generic can be "certified" it must be shown that the actual parameter does in fact have the properties required by the theory. Because computations do not use the axioms given in theories, there is no reason to restrict the form of the axioms in theories, and in fact, we allow arbitrary first-order axioms. Difficulty only arises when one has to
prove that the axioms hold of some particular module; then, one needs a first-order theorem prover like that of Stickel [46] or Hsiang [31], and, of course, the simpler the axioms the better the chance of actually getting the proof.\textsuperscript{8}

Here is an even simpler theory, the one that is actually used for the generic SET example:

\begin{verbatim}
theory TRIV is
  sorts elt
endth TRIV
\end{verbatim}

This theory requires nothing except that a particular sort be designated. We now give a generic BASICSET module, providing only symmetric difference, \( \cup \), and intersection; later we will define the rest of the set functions from these. After the name of the module comes a left square bracket, indicating that what follows is the formal parameter symbol, ELT in this case, and after the \(:\) comes the theory that it is required to satisfy; the formal parameter part is then closed by a right square bracket.

\begin{verbatim}
module BASICSET[ELT::TRIV] is
  sorts set
  fns
    \emptyset, \Omega : set
    _ _ : elt \to set
    \cup \cup : set,set \to set (assoc comm id: \emptyset)
    \cap \cap : set,set \to set (assoc comm idp id: \Omega)
  vars S,S',S'' : set,
    Elt,Elt' : elt
  axioms
    S \cup S = \emptyset.
    \{ Elt \} \cap \{ Elt' \} = : Elt \neq Elt'.
    S \cap \emptyset = \emptyset.
    S \cap (S' \cup S'') = (S \cap S') \cup (S \cap S'').
endmod BASICSET
\end{verbatim}

This way of defining finite sets follows Hsiang’s [31] approach to the propositional calculus; \( \Omega \) is the “universal” set, i.e., the set of all things of sort elt. The attribute \texttt{id:} should be taken as an abbreviation for the identity equation. In many cases, this definition will execute faster than more conventional axiomatizations\textsuperscript{9}; furthermore, since Hsiang has shown that it is confluent and terminating modulo associativity and

\textsuperscript{8}{Note that certain axioms may require induction for their proof.}\textsuperscript{9}{It is fast for conjunctions, but slow for disjunctions.}
commutativity, and Fages [15] has proven correctness of a unification algorithm modulo these equations, we know that narrowing will work [33]. This observation will be important in some later examples. It should be noted that the BASICSET module provides not only all finite subsets of the set given as actual parameter, but also all cofinite sets (i.e., sets whose complement is finite). The inequality in the axiom

\[
\{ \text{Elt} \} \cap \{ \text{Elt'} \} = \emptyset \iff \text{Elt} \neq \text{Elt'}.
\]

violates the purity of the language only in appearance, since Section 7.4 shows how to reduce the semantics of inequality to that of equality.

To instantiate a generic module, one must provide an actual parameter, A; but more than this is actually needed. Since both modules and theories can involve more than one sort, we need to say just which sorts in the actual correspond to those declared in the requirement theory T of the generic; similarly, we need to say which functions and predicates in an actual A correspond to those required by the theory. Following [19] and ideas developed for use in Clear [5], this correspondence is given by a \textit{view}, which consists of:

1. a function \( \sigma \) from the sorts of the theory T to those of A;
2. a function \( \phi \) from the functions of T to those A; and
3. a function \( \pi \) from the predicates of T to those of A,

such that

1. the subsort relation is preserved, in the sense that \( s < s' \) in T implies \( \sigma(s) < \sigma(s') \) in A;
2. the sorts of functions and predicates are preserved, in the sense that if function \( f \) in T has arity \( s_1 \cdots s_n \) and value sort \( s \) in T then \( \phi(f) \) has arity \( \sigma(s_1) \cdots \sigma(s_n) \) and value sort \( \sigma(s) \) in A; and similarly for predicates; and
3. the translations of the equations and axioms in T to equations and axioms about A are in fact true of the initial model of A.

Note that one sometimes wants to consider views where A is a theory rather than a module, in which case the translated equations and axioms need only follow from the axioms given for A. Thus, when A is a theory, one can use ordinary first-order logic (with equality) to prove the third condition, but when A is a module, more than this may also be needed, e.g., induction. Also note that in practical large-scale programming, one may wish to settle for less than a formal proof of the third condition; e.g., an informal proof might be acceptable. In the language of Goguen and Burstall [21], a view is a “theory morphism” between “data theories”; see Section 7.2 for more about this.

In many cases, it is obvious how to construct a view of A as T; this is formalized by the notion of a \textit{default view} in [19]. In other cases, there is only one appropriate view in the current environment, and of course that is the one to apply. In such cases, it is not necessary to indicate what view is intended; one can just write the name of the actual module. For example, in order to construct SET-OF-INT, we just say

\begin{verbatim}
make SET-OF-INT is SET[INT] endmake
\end{verbatim}
since there is a default view of INT as a TRIV. In other cases, it may be necessary to include a view in the `make` statement. For example,

```
make SORTING-OF-INT-DIV is SORTING[INT-AS-DIV-QUOSET] endmake
```

instantiates a generic `SORTING` module with the quoset of integers ordered by the divisibility relation. When it is not necessary to give the instantiated module a name, we can just write, e.g., `SET[INT]`.

We now enrich the generic `BASICSET` module given earlier (recall that it provided symmetric difference and intersection) to provide union, difference, and cardinality functions, plus some of the usual predicates.

```
module SET[X::TRIV] using NAT, BASICSET[X] is
  fns
  _\cup_ : set, set → set
  _-_- : set, set → set
  _- : set → nat
  preds
  _\in_ : elt, set
  _\subseteq_ : set, set
  empty : set
  _\notin_ : elt, set
  vars X : elt, S, S', S'' : set
  axioms
  SUS' = (S \cap S') \cup S \cup S'.
  S - S' = S \cap (S \cup S').
  empty(S) :- S = \emptyset.
  N \in S :- \{ N \} \cup S = S.
  N \notin S :- \{ N \} \cap S = \emptyset.
  S \subseteq S' :- SUS' = S'.
  \#_\emptyset_ = 0.
  \# (\{ X \} \cup S) = \text{pred}(\# S) :- X \in S.
  \# (\{ X \} \cup S) = \text{succ}(\# S) :- X \notin S.

endmod SET
```

Although \# does not yield the answer \( \infty \) for infinite sets, it does work reasonably. For example, in the case of `SET[INT]`, \# \( \Omega \) is just \# \( \Omega \) again, a reduced term rather than a nonterminating computation. In case \( \Omega \) is a finite set, say \( \{ 1, 3, 5 \} \), one should add an equation explaining this,

\[ \Omega = \{ 1 \} \cup \{ 3 \} \cup \{ 5 \} . \]
and in case \( \Omega \) is an infinite set, one may want to add the equations

\[
\text{succ}(\# \Omega) = \# \Omega. \\
\text{pred}(\# \Omega) = \# \Omega.
\]

in order to get the expected behavior in all cases.

We can also enrich a module without giving the enrichment an explicit name; this can be useful if some constants are being defined for a single query or example. Another feature illustrated by the following module is that when the requirement theory is \textsc{TRIV}, a view can be determined just by giving a sort name (provided that the sort only occurs in one module in the current environment). If the sort name does not occur in any module in the current environment, then it serves to declare a new sort and apply the generic to it; we shall call this a declaration "on the fly." For example,

```plaintext
module using \textsc{SET}[country] is
  fns s,s': set
  vars C : country
  axioms
      C \in s :\text{ pop}(C) \geq 150.
      C \in s' :\text{ area}(C) \geq 3.
endmod
```

We can now pose queries such as

```plaintext
\text{china } \in s.
\text{china } \in s \setminus s'.
\text{india } \in s - s'.
```

and get the expected answers: whether or not china has a population greater than (or equal to) 150 (million people); whether or not china has both a population greater than 150 and an area greater than (or equal to) 3 (million square miles); and whether or not india has a population greater than 150 and an area not greater than 3.

Views also provide an elegant form of declaration at the module level. In ordinary sequential programming, "assertions" can be inserted after a statement to indicate that the program's state is supposed to satisfy some property after the execution of that statement. In Eqlog, a view from a theory to a module serves to indicate that the module (i.e., its sorts, functions, and predicates) satisfies certain axioms. It should be noted that one can also compose generics. For example, one can form \textsc{BAG}[\textsc{SET}[\textsc{INT}]]; this is the so-called "module expression" [19].

Of course, there is nothing special about the details of the features and the syntax described here for Eqlog modules and generics; what is special is the underlying semantic ideas. Unfortunately, there is not room in this paper for a full exposition of this semantics, which is based on ideas from the Clear specification language [6]. The ideas are not really difficult, but they use some comparatively advanced mathematics. Some discussion is given in Section 7.2. The application of these ideas to the equational logic programming language OBJ is described in [19] and to an Ada library system in [20].
We now give a very general parameterized version of the construction that yielded the rationals from the integers. In order to do so, we first give a standard definition from algebra, the theory of rings:

```plaintext
theory RING is
  sorts ring
  fns
    _ + _ : ring,ring → ring (assoc comm id: 1)
    _ : ring → ring
    _ * _ : ring,ring → ring (assoc comm id: 0)
  vars X,Y,Z : ring
  axioms
    X + (−X) = 0.
    X *(Y + Z) = (X * Y) + (X * Z).
endth RING
```

Our general parameterized fraction construction requires that the elements that can appear as denominators be specifically designated by a subsort. In many cases this subsort denotes the nonzero elements; e.g., with the integers, and with polynomials over the integers. But the general construction only requires a set of elements that contains 1 and is closed under multiplication. The following theory expresses this requirement.

```plaintext
theory MULT using RING is
  sorts mult
  subsorts mult < ring
  vars X,Y : ring
  axioms
    mult(1).
    mult(X * Y) :- mult(X), mult(Y).
endth MULT
```

Now we are ready for the general parameterized construction. It is essentially the same as the usual construction for the rationals, except for requiring a multiplicatively closed subsort denoted mult. In addition, we define a subsort invertible of fractions that can be inverted.

```plaintext
module FRACTION[M :: MULT] is
  sorts invertible, fract
  subsorts ring, invertible < fract
```
fns

` `/ : ring,mult -> fract

` * ` : fract,fract -> fract (assoc comm id: 0)

` + ` : fract,fract -> fract (assoc comm id: 1)

` ^{\mathbf{-1}} ` : invertible -> invertible

vars Y,W : molt X,Z : ring

axioms

\[
\begin{align*}
X / 1 &= X. \\
X / Y &= Z / W :\quad X * W = Z * Y. \\
(X / Y) * (Z / W) &= (X * Z) / (Y * W). \\
(X / Y) + (Z / W) &= ((X * W) + (Z * Y)) / (Y * W). \\
\text{invertible}(X / Y) &\iff \text{mult}(X). \\
(X / Y)^{-1} &= (Y / X).
\end{align*}
\]

endmod FRACTION

In order to apply this, one says, e.g.,

```
make RAT is FRACTION[INT] endmake
```

to get the rationals from the integers. Of course, this needs a view of INT as MULT, and we shall assume that the usual one, assigning nzint to mult, has been provided in the environment. This view looks like

```latex
\begin{verbatim}
view NZINT-AS-MULT is INT as MULT by 
  int is ring 
  nzint is mult 
endview
\end{verbatim}
```

so that the explicit form of the application of FRACTION to INT given above is

```
make RAT is FRACTION[NZINT-AS-MULT] endmake
```

(In case any reader is puzzled, it may be worth noting what happens if one takes the multiplicatively closed set to be the whole ring: everything collapses, i.e., everything in the ring of fractions is equal to zero.)

7. LOGICAL FOUNDATIONS

This section discusses in more detail four issues regarding the foundations of Eqlog: subsorts, generics, narrowing, and inequality.

7.1. Subsorts

Many of our examples use subsorts and subsort predicates. We now explain why this is not an impure feature, but rather an expressive shorthand for a specification in
standard Horn clause logic with equality. We also describe some conditions that insure valid use of the equality predicate; these conditions could be enforced syntactically. Although more permissive uses of subsort predicates are possible and certainly worth exploring, the one presented here is already very general.

Whenever a subsort \( s < s' \) is declared, a corresponding unary predicate \( s(\_ ) \) of sort \( s' \) also becomes available; intuitively, this predicate is true of a term iff that term lies in the subsort. Users can give axioms involving the subsort predicate; but these should only assert that certain functions restrict (and constants belong) to the subsort. For example, the subsort \( \text{pos} < \text{int} \) of positive integers can be characterized as containing \( 1 = \text{suc}(0) \) and being closed under the successor function, by the two clauses

\[
\text{pos}(1).
\]

\[
\text{pos}(\text{suc}(X)) :- \text{pos}(X).
\]

Our reconstruction of subsorts within standard Horn clause logic with equality involves giving ordinary signatures \( \Sigma \) and \( \Pi \), and a set \( \mathcal{C} \) of Horn clauses, such that the initial model \( T_{\Sigma,\Pi} \) is isomorphic to the model intended for the subsort declarations and their corresponding predicates. The first step is to regard each subsort as an ordinary sort for each subsort. We then force that in all models, the sort \( s \) is identified with a subset of the sort \( s' \) whenever \( s < s' \) by introducing a new function symbol \( j : s \rightarrow s' \) that is made to play the role of an inclusion by satisfying the axiom

\[
j(X) = j(Y) :- X = Y.
\]

Similarly, we can express the fact that certain functions or constants restrict to a subsort by introducing new function symbols for these functions and constants such that their value sort is the subsort; equations are then given to insure their relationship to the functions and constants in the supersort. For example, from \( \text{pos} < \text{int} \) we would get function declarations

\[
1 : \text{pos}
\]

\[
\text{suc} : \text{pos} \rightarrow \text{pos}.
\]

with equations

\[
j(1) = \text{suc}(0).
\]

\[
j(\text{suc}(X)) = \text{suc}(j(X)).
\]

In order to get the unary predicate \( s(\_ ) \) of sort \( s' \), which is \( \text{pos}(\_ ) \) in the present example, we need give just one clause,

\[
s(j(X)).
\]

This clause makes the predicate true in the initial model iff the element belongs to the subsort. (A more detailed discussion of these issues will appear in [23].)

We now illustrate this technique with two previously given examples. In the definition of the requirement theory \( \text{MULT} \), the clause

\[
\text{mult}(X \cdot Y) :- \text{mult}(X),\text{mult}(Y).
\]

for the subsort \( \text{mult} < \text{ring} \) can be translated, using the new function symbol

\[
\_ \cdot \_ : \text{mult},\text{mult} \rightarrow \text{mult}
\]
to the equation
\[ j(X \cdot Y) = j(X) \cdot j(Y). \]
Similarly, for the subsort invertible \(<\) fract in the FRACTION module, the axiom
\[ \text{invertible}(X / Y) :- \text{mult}(X). \]
corresponds to introducing a new function
\[ \_ / \_ : \text{mult, mult} \rightarrow \text{invertible} \]
and an equation
\[ j(X) / Y = j'(X / Y). \]
where \( j : \text{mult} \rightarrow \text{ring} \) and \( j' : \text{invertible} \rightarrow \text{fract} \) represent the inclusions.

Subsorts also provide a simple and natural way of dealing with functions that are not defined on the entire collection of data items of some sort. For example, head and tail are only defined on nonempty lists. The approach is simply to define a subsort \( \text{relist} < \text{list} \) of nonempty lists; the head and tail are functions with domain \( \text{relist} \) rather than all of list. This is shown in the following:

\begin{verbatim}
module LIST [x :: TRIV] using NAT is
  sorts list, relist
  subsorts x < relist < list
  fns
    _ / _ : list, list \rightarrow list (assoc id : nil)
    _ : list \rightarrow nat
  vars X : x, N, N' : relist, L, L' : list
  axioms
    relist (X).
    relist (N N') :- relist (N).
    relist (N N') :- relist (N').
    nil = 0.
    X = 1.
    (L L') = L + L'.
endmod LIST
\end{verbatim}

### 7.2. Putting Theories Together

The module, theory, view, and instantiation features of Eqlog support generic (i.e., parameterized) programming, a form of programming-in-the-large that permits an unusually high degree of reusability. All these features can be defined for any logical system satisfying some very simple and reasonable axioms that make it an institution
In particular, it has been shown that the logic of Horn clauses with equality is an institution, so the general machinery can be applied directly to this case, giving a semantics for the parameterization features in Eqlog.

This approach relies on category theoretic concepts like colimit, and therefore to explain it in detail here would require substantially greater prerequisites of the reader. However, we can give the flavor of the approach by considering an example in some detail. We will use the example given in Section 6 of applying the generic $\textsc{fraction[M : : multi]}$ to the actual $\textsc{int}$ to get the rationals $\textsc{rat}$. The first thing to notice is that the requirement theory $\textsc{multi}$ can be considered a subtheory of the generic module body $\textsc{fraction}$; this is because $\textsc{fraction}$ can also be considered a theory, since it consists of a set of axioms. However, there is a significant difference in the intended (model-theoretic) interpretations for these two theories: we will allow any interpretation for $\textsc{multi}$, but we will only allow interpretations for $\textsc{fraction}$ such that the sort fract is free over the (arbitrary) sort ring; i.e., the fractions are uniquely determined once the ring elements and denominators are given. This is a kind of relative closed world assumption, and is explained technically by the notion of a “data constraint” in [21]. Let us denote the inclusion of $\textsc{multi}$ into $\textsc{fraction}$ by $I$.

The next thing we need is a view from $\textsc{multi}$ to $\textsc{int}$; it is just the obvious one that maps ring to int and mult to nzint; this view is denoted $\textsc{nzint-as-multi}$ in Section 6, but let us just denote it $V$ here. Thus, we have two views whose source is the theory $\textsc{multi}$. In the language of category theory, what we want is the “pushout” of these two theory morphisms; this is a kind of “least upper bound” theory which contains both the target theories (namely $\textsc{fraction}$ and $\textsc{int}$) but identifies the parts that they are supposed to share, as given by $\textsc{multi}$ (or more accurately, by the two views from $\textsc{multi}$). See Figure 1.

From an implementation point of view, all that needs to be done to get the new module $\textsc{rat}$ is to take the text for $\textsc{fraction}$ and substitute the text for $\textsc{int}$ (or appropriate pointers, in case $\textsc{int}$ is actually built-in, as of course it should be) for the text of $\textsc{multi}$, making exactly the changes indicated by the view $V$; one could regard $V$ as instructions for how to “edit” $\textsc{fraction}$ to get $\textsc{rat}$, but of course, these instructions are given in a very structured form, and are guaranteed (by the additional semantic conditions that must be fulfilled by views) to give a meaningful new module $\textsc{rat}$. Thus, it is also important to implement at least the syntactic checks that are implied by the notion of view; however, the semantic checks involve theorem proving, so that users may well want to put them off, or even omit them entirely. Nevertheless, users should be aware that if the semantic conditions are in fact not satisfied, then the instantiation will have some unexpected properties.

The theory of “institutions” of [21] gives general sufficient conditions for when such pushouts can be calculated, also taking account of any “data constraints” that may occur in the theories involved. It may be worth remarking that the subsystem of

FIGURE 1. Application of $\textsc{fraction}$ to $\textsc{int}$.
Horn clause logic with equality consisting of pure equations plus Horn clauses whose heads are not equations, is also an institution; this restriction on the syntax of clauses was mentioned in Section 4 as a simple sufficient condition for our evaluation algorithm to be complete.

7.3. Unification in an Equational Theory

An equational theory is given by a pair \( \langle \Sigma, T \rangle \) where \( \Sigma \) is an \( S \)-sorted signature of function symbols and \( T \) is a set of \( \Sigma \)-equations. The rules of many-sorted equational deduction [22] define an equivalence relation \( =_{T} \) between \( \Sigma \)-terms with variables, namely that of being provably equal using the equations in \( T \). If \( X \) denotes an \( S \)-sorted set containing an infinite supply of variables of each sort, and if \( T_{\Sigma}(X) \) stands for the \( \Sigma \)-algebra of terms with variables in \( X \), then a substitution is an \( S \)-sorted function \( \alpha: X \rightarrow T_{\Sigma}(X) \); such a function extends to a unique \( \Sigma \)-homomorphism from \( T_{\Sigma}(X) \) to itself that we also denote by \( \alpha \). A substitution \( \alpha \) is said to have domain \( Y = \{ Y_{s} \} \) when \( Y_{s} = \{ x \in X : \alpha_{s}(x) = x \} \); we then write \( Y = \text{dom}(\alpha) \). The set of variables introduced by \( \alpha \) is the \( S \)-sorted \( \text{int}(\alpha) = \bigcup \{ \text{vars}(\alpha(x)) \mid x \in \text{dom}(\alpha)_{s} \} \), where \( \text{vars}(t) \) denotes the set of variables occurring in a term \( t \).

Given an \( S \)-sorted set of variables \( Y \subseteq X \) and substitutions \( \alpha \) and \( \beta \), we write \( \alpha =_{T} \beta \ [Y] \) iff \( \alpha(x) =_{T} \beta(x) \) for each \( x \) in \( Y \). Similarly, we write \( \alpha \leq_{T} \beta \ [Y] \) iff there is a substitution \( \gamma \) such that \( \beta =_{T} \gamma \star \alpha \ [Y] \). A \( T \)-unifier of two terms \( t \) and \( t' \) is a substitution \( \alpha \) such that \( \alpha(t) =_{T} \alpha(t') \). Given terms \( t \) and \( t' \) with \( Y = \text{vars}(t) \cup \text{vars}(t') \), a set \( L \) of \( T \)-unifiers of \( t \) and \( t' \) is called a complete set of \( T \)-unifiers of \( t \) and \( t' \) iff for each \( T \)-unifier \( \gamma \) of \( t \) and \( t' \) there is an \( \alpha \) in \( L \) with \( \alpha \leq_{T} \gamma \ [Y] \). (This was called a most general complete solution in Section 3.) Without loss of generality we may assume, for technical reasons, that \( \text{dom}(\alpha) \subseteq Y \) and \( \text{int}(\alpha) \cap Y = \emptyset \) for each \( \alpha \) in \( L \).

Given an equational theory, a complete \( T \)-unification algorithm \( \text{SOLN} \) is an algorithm such that if started with any two terms \( t \) and \( t' \), \( \text{SOLN} \) generates a complete set of \( T \)-unifiers for \( t \) and \( t' \); \( \text{SOLN} \) is finite iff, in addition, it always terminates with a finite set. Particular unification algorithms for theories \( T \) of frequent use, such as associativity and commutativity, have been given in the literature. For the general case when \( T \) consists of a confluent and terminating set \( \mathcal{R} \) of rewrite rules, a unification algorithm using narrowing has been given by Fay [17] and improved in order to give a termination criterion by Hullot [32]. When \( \mathcal{R} \) is confluent and terminating, then for any two terms one has \( t =_{T} t' \) iff \( \text{can}(t) = \text{can}(t') \), where \( \text{can}(t) \) is the canonical form of the term \( t \), obtained after exhaustive rewriting by applications of the rules \( \mathcal{R} \).

The one step narrowing relation is defined as follows: let \( t \) be a term; by renaming of variables (or some other convention) we can always assume that the variables occurring in \( t \) do not occur in any of the rules. Let \( t_{0} \) be a nonvariable subterm of \( t \) that unifies (in the ordinary sense) with the left-hand side \( t_{1} \) of a rule \( t_{1} = t_{2} \) in \( \mathcal{R} \), with \( \alpha \) the most general unifier. Let \( t' \) be the term obtained by replacing in \( \alpha(t) \) the subterm \( \alpha(t_{0}) = \alpha(t_{1}) \) by \( \alpha(t_{2}) \). Then we say that \( t' \) is a one step narrowing of \( t \), and we write \( t \rightarrow t' \). The narrowing relation is the reflexive and transitive closure of one step narrowing, and contains the rewriting relation as a subset. The following algorithm then provides a complete set of \( T \)-unifiers.
Theorem 2. [17,32]. Let $T = \mathcal{R}$ be a confluent and terminating set of rewrite rules. Given a pair $t,t'$ of terms, introduce a new function symbol $\tau$ and consider all the narrowing chains that begin with $\tau(t,t')$. If such a chain ends with a term of the form $\tau(t_n,t'_n)$ such that $t_n$ and $t'_n$ are unifiable by a substitution $\alpha$, then compose $\alpha$ with the substitutions obtained at the previous narrowing steps in the chain, and add this composition to the set of unifiers already generated. The set so obtained is a complete set of $T$-unifiers for $t$ and $t'$.

This algorithm has been extended to handle the more general situation when the equations in $T$ can be partitioned into a set of rewrite rules $\mathcal{R}$ and a set of equations $\mathcal{E}$ in such a way that $\mathcal{R}$ is terminating and confluent “modulo $\mathcal{E}$”. Many common examples fall into this category. Some important cases were treated by Hullot [32], while a general answer is given by Jouannaud, Kirchner, and Kirchner [33], who generalize Theorem 2 by showing that if there is a finite $\mathcal{E}$-unification algorithm, then narrowing modulo $\mathcal{E}$ still provides a complete $T = \mathcal{R} \cup \mathcal{E}$-unification algorithm. The idea, in this case, is to have part of the $T$-unification work done by a built-in $\mathcal{E}$-unification algorithm, and the rest by $\mathcal{E}$-narrowing. Both Hullot [32] and Jouannaud, Kirchner, and Kirchner [33] give sufficient conditions for termination of their algorithms.

Now a simple example showing how a query involving an equation is evaluated by narrowing; for illustrative purposes, this example does not use the built-in natural number type, but rather provides its own, of sort $\text{nat1}$, with successor function $\text{succ}$; also, notice there is no $\text{nil}$ list here.

\[
\text{module LIST[ELT :: TRIV] is}
\]
\[
\text{sorts elt, list, nat1}
\]
\[
\text{subsorts elt < list}
\]
\[
\text{fns}
\]
\[
0 : \text{nat1}
\]
\[
\text{succ : nat1 } \rightarrow \text{ nat1}
\]
\[
* : \text{list,elt } \rightarrow \text{ list}
\]
\[
\text{length : list } \rightarrow \text{ nat1}
\]
\[
\text{vars Elm : elt, Lst : list}
\]
\[
\text{axioms}
\]
\[
\text{length(Elm) = succ(0).}
\]
\[
\text{length(Lst * Elm) = succ(length(Lst)).}
\]
\[
\text{endmod LIST}
\]

The sort $\text{elt}$ is a parameter, and is empty in the Herbrand universe; however, this causes no problem if a suitable modification of the rules of deduction is used (see

---

10 The reader may find it helpful to construe this symbol as a formal equality symbol.
Figure 2 shows how the query

\[ \text{length}(\text{Lst}') = \text{succ}(\text{succ}(\text{succ}(0))) \]

evaluates to

\[ \text{length}((\text{Elm}'' \ast \text{Elm}') \ast \text{Elm}) = \text{succ}(\text{succ}(\text{succ}(0))) \]

by accumulating the substitutions associated with the narrowings from the root \( \text{length}(\text{Lst}') \) to the expression \( \text{succ}(\text{succ}(\text{succ}(0))) \).

Let us follow this in more detail. First of all, since the term \( \text{succ}(\text{succ}(\text{succ}(0))) \) is a canonical form ground term, it cannot be narrowed any further. Thus the narrowings of \( \tau(\text{length}(\text{Lst}'), \text{succ}(\text{succ}(\text{succ}(0)))) \) are exactly those of the subterm \( \text{length}(\text{Lst}') \). The term \( \text{length}(\text{Lst}') \) unifies with the left-hand side \( \text{length}(\text{Elm}) \) by the substitution

\[ \alpha_1(\text{Lst}') = \text{Elm} \]

giving the narrowing

\[ \text{length}(\text{Lst}') \Rightarrow \text{succ}(0) \]

which is a failure node since \( \text{succ}(0) \) cannot be narrowed any further and does not unify with \( \text{succ}(\text{succ}(\text{succ}(0))) \). The term \( \text{length}(\text{Lst}') \) also unifies with the left-hand side \( \text{length}(\text{Lst} \ast \text{Elm}) \) by the substitution

\[ \beta_1(\text{Lst}') = (\text{Lst}'' \ast \text{Elm}) \]

\[ \beta_1(\text{Lst}) = \text{Lst}'' \]

giving the narrowing

\[ \text{length}(\text{Lst}') \Rightarrow \text{succ}(\text{length}(\text{Lst}'')). \]

In the same way we get a narrowing

\[ \text{succ}(\text{length}(\text{Lst}')) \Rightarrow \text{succ}(\text{succ}(0)) \]

leading to another failure node, and a narrowing

\[ \text{succ}(\text{length}(\text{Lst}')) \Rightarrow \text{succ}(\text{succ}(\text{length}(\text{Lst}''))) \]

with substitution

\[ \beta_2(\text{Lst}') = (\text{Lst}''' \ast \text{Elm}'), \beta_2(\text{Elm}) = \text{Elm}', \beta_2(\text{Lst}) = \text{Lst}''' \]

One more step of narrowing

\[ \text{succ}(\text{succ}(\text{length}(\text{Lst}''))) \Rightarrow \text{succ}(\text{succ}(\text{succ}(0))) \]

\[ \ldots \]

FIGURE 2. Narrowing on the length function.
gives a success node with substitution
\[ \alpha_3(Lst') = Elm'. \]
The corresponding unifier is the substitution
\[ \gamma = \alpha_3 \circ \beta_2 \circ \beta_1 \]
which finally gives the solution
\[ \text{length}((Elm' * Elm') * Elm) = \text{succ}(\text{succ}(\text{succ}(0))). \]

Now that we have discussed narrowing, let us return to the question of sufficient conditions for completeness of our Eqlog evaluation algorithm. Recall that we need to call the function SOLN in order to get the unifications needed by the Prolog-like part of the evaluation algorithm, and that we propose to use narrowing to implement SOLN for user defined modules. However, if there are equations that are conditioned by predicates, then we will need to use Prolog-like evaluation just to see whether or not we can apply such equations during the narrowing process. Thus, in order for SOLN to do narrowing on a term \( t \) by means of the equation
\[ t_1 = t_2 :- P_1, \ldots, P_n. \]
the goals \( o(P_1), \ldots, o(P_n) \) have to be solved (where \( o \) is the substitution unifying \( t_1 \) with a subterm of \( t \)), and this will yield another substitution \( \sigma' \).

Further research is needed to develop methods for establishing the completeness of this strategy for given sets \( \mathcal{C} \) of clauses. However, we conjecture that an abstract proof of completeness can be given for the case where the set of clauses having equations as their heads is confluent and terminating as a set of rewrite rules.

### 7.4. Equality and Inequality

The use of negation for arbitrary predicates gives rise to difficulties. However, perhaps surprisingly, it is not so difficult to treat the negation of equality. For example, the \textsc{Basicset} module of Section 6 contains the axiom
\[ \{ \text{Elt} \} \cap \{ \text{Elt'} \} = \emptyset :- \text{Elt} \not= \text{Elt'}. \]
which appears to lie outside the realm of Horn clause logic with equality. However, this is only an appearance, because the semantics of inequality can be reduced to that of equality. The equational part of any Eqlog module should define a \textit{computable} abstract data type. This is implicit in our requirement that the equations form a confluent and terminating set of rewrite rules (perhaps modulo some decidable equations such as associativity, commutativity, etc.) since it has been shown that any computable data type can be presented that way. Equality and inequality of ground terms is then built in, since one can just compute the canonical forms of the terms in question and see whether or not they are equal. Moreover, as shown in [39], a data type is computable if and only if its equality is finitely axiomatizable by equations. This means that we can always axiomatize equality for each sort \( s \) as a function \( \equiv : s \times s \rightarrow \text{bool} \), by means of a finite set of equations. \texttt{bool} is a new sort having two constants, \texttt{true} and \texttt{false}, such that for any two ground terms \( t, t' \) we have
\[ t = t' \text{ (in the data type) iff } (t \equiv t') = \texttt{true} \text{ (in the equational equality enrichment)} \]
and similarly,
\[ t \not= t' \text{ (in the data type) iff } (t \equiv t') = \texttt{false} \text{ (in the equational equality enrichment)}. \]
In this way, inequality is reduced to equality.
Given an inequality \( t \neq t' \), the Eqlog system will then:

1. compute it by rewriting if both \( t \) and \( t' \) are ground terms; and
2. otherwise, requiring the existence of an equationally defined equality, \( \equiv \), for the sort in question, translate the inequality into the equation

\[ (t = t') = \text{false}. \]

and then solve this equation by using narrowing.

8. THE MISSIONARIES AND CANNIBALS PROBLEM

To illustrate the power of Eqlog, we show how to use some standard generics (which of course would be taken from the Eqlog library), plus subsorts, functions, and predicates, for a general Missionaries and Cannibals problem (hereafter, MAC); once the parameters are instantiated, Eqlog solves MAC by \( \& \)-narrowing, for \( \& \) a set of equations including associativity and commutativity for the set functions.

We begin with a theory \( \text{MACTH} \) of the preconditions for MAC: there are two disjoint sets of persons, \( m0 \) of missionaries and \( c0 \) of cannibals. Later we instantiate \( \text{MACTH} \) to the usual case of three missionaries and three cannibals. \( \text{MACTH} \) uses a generic \( \text{SET} \) module to get set difference, union, and cardinality. By convention, a module with a “principal” sort has the same name as that sort (unless explicitly indicated otherwise); e.g., the sort of \( \text{PSET} \) is \( \text{pset} \).

```
theory MACTH[PERSON::TRIV] using SET, PSET = SET[PERSON] is
  fns
  m0 : pset
  c0 : pset
  axioms
  m0 \cap c0 = \emptyset.
endth MACTH
```

The MAC module also uses a generic \( \text{LIST} \) module that provides the empty list \( \text{nil} \), the length function \( \# \), and concatenation \( * \). The new sort \( \text{trip} \) is introduced “on the fly” (see Section 6) in the submodule \( \text{TRIPLIST} \).

We now briefly discuss the intuition behind this specification. A solution is a list of trips having certain “good” properties, where a trip is a boat containing a set of persons; odd numbered trips go from the left bank to the right, and even trips go from the right to the left. Missionaries and cannibals are persons. The predicate \( \text{boatok} \) indicates that a boat has an \( \text{ok} \) number of persons; the predicate \( \text{good} \) is true if a list of trips never allows there to be more cannibals than missionaries on a bank; the predicate \( \text{solve} \) indicates that a trip list is a solution to the problem. The functions \( \text{lb} \) and \( \text{rb} \) give the sets of persons on the left and right banks, respectively, and the functions \( \text{mset} \) and \( \text{cset} \) extract the subsets of missionaries and cannibals (respectively) from a set of persons.
module MAC[T :: MACTH] using NAT, TRIPLIST = LIST[trip] is

preds
  boatok : trip
  solve,good : triplist

fns
  boat : pset → trip
  lb,rb : triplist → pset
  mset,cset : pset → pset

vars PS : pset, L : triplist, P : person, T : trip

axioms
  boatok(boat(PS)) :- # PS = 1.
  boatok(boat(PS)) :- # PS = 2.
  lb(nil) = m0∪c0.
  mset(PS) = PS∩m0.
  cset(PS) = PS∩c0.
  rb(nil) = ∅.
  lb(L * boat(PS)) = lb(L) - PS :- even # L.
  rb(L * boat(PS)) = rb(L)∪PS :- even # L.
  rb(L * boat(PS)) = rb(L) - PS :- odd # L.
  lb(L * boat(PS)) = lb(L)∪PS :- odd # L.
  good(L * T) :- # cset(lb(L * T)) =< # mset(lb(L * T)).
                 # cset(rb(L * T)) =< # mset(rb(L * T)) good(L), boatok(T).
  good(nil).
  solve(L) :- good(L), lb(L) = ∅.

endmod MAC

Now the constants to instantiate MAC to the usual case.

module EX1 using SET[ID] is

axioms
  m0 = { taylor, helen, william }.
  c0 = { umugu, nzware, amoc }.

endmodule EX1

The notation \{a, b, c\} is shorthand for \{ a \}∪\{ b \}∪\{ c \}. We can now instantiate MAC and ask Eqlog to solve the resulting problem with

make MAC[EX1] endmake

solve (L).

using the default view of EX1 as MACTH, and not bothering to give the resulting module a name.
Of course, what we have written will produce a crude depth first search by $\delta$-narrowing. However, the logic programming slogan "Program = Logic + Control" can be applied to add some control information to the above. (We have not discussed control for Eqlog in this paper, but the problems are clearly of the same kind as in ordinary Prolog, and will yield to the same kinds of solution.)

9. FUTURE RESEARCH AND CONCLUSIONS

There is some interesting recent work that can be viewed as providing ways to reduce all logic programming to equational logic. For example, Dershowitz [12] encodes predicates as boolean valued functions and turns queries into equations of the form

$$t(x_1, \ldots, x_n) = \text{true}.$$ 

that are solved by a method essentially equivalent to narrowing. Of course Eqlog can do that too, but we adopt a different philosophy. Rather than trying to encode all function symbols as predicates or all predicates as function symbols, we prefer to provide queries for predicates, solutions for equations, and evaluation for terms, as different options within a unified framework, in the conviction that both programmers's intuition and efficiency considerations will, for each particular problem, make some combinations of options preferable and more natural than others.

The work of Hsiang [31] is also of great potential interest, since it provides an entirely algebraic paradigm for theorem proving by means of rewrite rules (also encoding predicates as boolean valued functions, and doing unification modulo the boolean equations). However, his present approach does not provide an explicit way to get the substitution that answers a query; rather, a refutation is proved by completing associated rewrite rules until $\text{true} \rightarrow \text{false}$ is generated. Another approach for dealing with the full power of first-order predicate calculus is presented by Stickel [46]. It would be very interesting to explore how these three approaches might be combined with that presented in this paper.

Although most of the language features that we have presented for Eqlog have been implemented in Prolog, OBJ, or Clear, the mechanism that joins the predicate logic and equational logic features, namely narrowing, has not been implemented in a programming language context. This is something that we very much wish to do in the near future. In fact, once unification and term rewriting are available, it is very little extra effort to provide narrowing, since it is just a combination of these two. Most of the work would actually lie in implementing solution algorithms for the built-in types, including good incremental implementations of built-in unification algorithms for frequently used theories. This will enable us to explore the important open question of the efficiency of narrowing. Clearly, narrowing can be very inefficient for some cases, and very efficient for others. Therefore it would be very useful to have simple and general sufficient conditions (on the form of ADT definitions) that would guarantee the efficiency of narrowing.

Further research is needed to develop methods for establishing the completeness of our Eqlog evaluation algorithm for given sets $\mathcal{E}$ of clauses. Several conjectures regarding sufficient conditions have been given in the paper; in particular, we
conjecture that an abstract proof of completeness can be given for the case where the set of clauses having equations as their heads is confluent and terminating as a set of conditional rewrite rules.

The many advantages that have been claimed for logic programming, including simplicity, clarity, understandability, reusability, and maintainability, are all seriously compromised to the extent that the logic used in the programming language is not pure. Thus, it is highly desirable to extend the logic in such a way as to encompass a greater range of programming language features. This paper has shown how many-sorted logic supports strong typing and, with a little extra effort, subsorts. We also show how some results from the theory of institutions support very powerful features for programming with generic modules, including instantiation with views. Moreover, we have shown how Hsiang's [31] axiomatization of Boolean algebra provides a convenient implementation of finite (and cofinite) set theory that supports narrowing. Finally, we have provided a number of examples showing the power of the various language features, including a very simple program for the Missionaries and Cannibals problem. Probably, there are too many new ideas for one paper.

It is worth emphasizing certain fundamental points. Eqlog combines pure predicate logic programming with first-order functional programming; i.e., predicates are predicates, functions are functions, and the programmer does not have to encode either one in terms of the other. However, Eqlog is based on the logic of Horn clauses (with equality), rather than the full first-order predicate calculus (with equality). This means that any set of clauses has a unique (up to isomorphism) standard (i.e., initial) model in which there is "no junk" (i.e., all its elements are namable) and "no confusion" (i.e., a predicate holds iff it is provable from the given clauses, and in particular, two terms are equal iff that is provable). This amounts to the Closed World assumption upon which Clark's [9] negation as failure rests. Eqlog also supports arbitrary user definable generic modules, including abstract data types. Finally, perhaps the main point of this paper is that narrowing is the proper join of predicate logic and functional programming.

We extend our most sincere thanks to Jean-Pierre Jouannaud and Fernando Pereira for their extensive comments on this paper, and for their help and encouragement while it was being imagined and then constructed.

REFERENCES


published in *Houston Journal of Mathematics*.


