PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

SOFTWARE FOR THE REWRITE RULE MACHINE*

Joseph A. Goguen† and José Meseguer
SRI International, Menlo Park CA 94025

ABSTRACT

The Rewrite Rule Machine (RRM) has an innovative massively parallel architecture that combines fine-grain SIMD computation with (two levels of) coarse-grain MIMD computation. This paper describes techniques for compiling and running functional, object oriented, relational (i.e., "logic"), and multi-paradigm languages on the RRM. The languages that we use for illustration have the advantage that they are rigorously based upon logical systems, but the implementation techniques are more general, and even apply to imperative languages. The most novel of these techniques is a restricted form of *second order rewriting*, which involves variables that can match against function symbols. Rules involving such variables have enormous expressive power, and can also be implemented very efficiently on the RRM; indeed, they are implemented essentially the same way as ordinary rules. A second innovative technique involves representing *objects* (with local state) in graphs, by restricting the ways that rules can act on them. The RRM languages also embody many useful modern features, including abstract data types, flexible generic modules, powerful module interconnection, multiple inheritance, and "wide spectrum" integration of specification, documentation and coding.

1 Introduction

Beginning with a plea for powerful, simple languages that are rigorously based upon pure logics, this introduction discusses *multi-grain concurrency* and our model of computation, *concurrent term rewriting*. These concepts motivate the Rewrite Rule Machine (RRM) architecture and languages. The subsequent body of the paper provides details about the languages and their RRM implementation, showing that they can be given very efficient implementations in part because of their high level abstract character and clean design. The references deliberately emphasize related works by the RRM group. See [16] for details of RRM architecture.

1.1 Programmability and Logical Languages

Programmability is a central issue for massively parallel machines, because such machines lose their value if they are too difficult to program. We suggest that *declarative languages* are the key to combining hardware efficiency with programming ease. Programs in such languages tend to describe problems, rather than solutions. From the hardware viewpoint, declarative languages do not prescribe specific orders of execution, and thus give maximum opportunity for concurrency. From the software viewpoint, declarative languages avoid the need to explicitly program concurrency, which in general is difficult. Moreover, a modern declarative language can provide facilities that can greatly augment programmer productivity, including the *wide spectrum* integration of specification, rapid prototyping, validation, testing, documentation, and coding, as well as generic modules, multiple inheritance, and program transformation, all of which support reusability, as well as strong typing and multiple inheritance, which support exception handling.

Figure 1: Overview of Languages

Since all this can be given a solid logical foundation, correctness problems can be properly addressed, with both programs and proofs in the same formal system. Moreover, programs written in declarative languages do not need to be rewritten if the underlying hardware is slightly changed — e.g., if more processors are added — since they are already independent of any assumptions about the underlying hardware.

The most promising approach to declarative programming may be through *logical programming languages*, which (roughly speaking) are languages whose statements are sentences in some logical system, and whose computation is deduction in that system ([13] gives a more precise definition). This paper describes RRM implementation techniques for four wide spectrum logical programming languages:

1. OBJ [2,3], which is purely functional;
2. FOOPS [13], which combines functional and object oriented programming;
3. Eqlog [10,12], which combines functional and "logic" (i.e., Horn clause relational) programming; and
4. FOOPlog [13], which combines all three major emerging programming paradigms.

Figure 1 shows the relationships between these languages and their logics.

It is widely recognized that pure Horn clause logic is not an adequate basis for practical programming, and Prolog, for example, has added imperative features like `is`, `assert` and

*Supported by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450.

†Address from September 1988 onward: Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road,

quite misleading name), while the second and third update Prolog's database. However, this does not mean that logical programming languages are a bad idea, but only that some logic more powerful than Horn clause logic is needed in order to make pure logical programming practical. Subsequent sections of this paper discuss languages that are based on reflective (order sorted) equational logic, and on (order sorted) Horn clause logic with equality; these languages are FOOPS and Eqlog, respectively. Prior to this, we discuss OBJ, which is based on (order sorted) equational logic. Order sorted logic provides a rigorous foundation for multiple inheritance. All these languages have **initial model semantics**, which formalizes the idea that one wants to program over a "standard model" or "closed world" in which questions have determinate answers.

1.2 Multi-Grain Concurrency

Concurrent execution may be roughly classified as either fine-grain or coarse-grain. Fine-grain SIMD concurrency (broadcasting a Single Instruction stream to Multiple Data sites) achieves efficient performance at the cost of generality, flexibility, and programmability. Coarse-grain MIMD (Multiple Instruction streams at Multiple Data sites) execution is more broadly applicable, but cannot achieve maximum concurrency because of high communication costs. It is an important research problem to escape this fateful dichotomy.

Experience shows that many computations are *locally homogeneous*, in the sense that many instances of one instruction can be applied simultaneously at many different data sites. For example, sorting, searching, matrix inversion, the fast Fourier transform, and arbitrary precision arithmetic, all have this character. For such computations, SIMD architecture is advantageous at the VLSI level.

On the other hand, *complex problems* tend to have many different subproblems with little or no overlap among their instructions — that is, complex problems tend to involve *globally inhomogeneous* computation. SIMD computation can be very inefficient for such problems. We say that computations that are locally SIMD but globally MIMD exhibit *multi-grain concurrency*. Architecturally, this suggests many processors, each running its own SIMD program, independently of what is running on other processors. Such an architecture is naturally realized by a network of VLSI chips, each a SIMD processor.

Thus, progress in VLSI and communication has created a technological opportunity, answering a real need for large, complex computations. Unfortunately, there are serious conceptual and linguistic obstructions to exploiting this opportunity. In fact, no well known programming language or computational model is adequate for multi-grain concurrent computation. In particular, the von Neumann languages and model of computation are inadequate, because they are inherently sequential. However, concurrent term rewriting seems ideally suited for multi-grain concurrency.

1.3 Models of Computation

A *model of computation* defines the major interface between the hardware and the software aspects of a computing system. This interface specifies what the hardware team must implement, and what the software team can rely upon, and thus plays a basic role in "Fifth Generation" projects. The only justification for continued interest in the von Neumann model of computation is that it connects current generation (efficient) von Neumann machines with current generation (ugly but very widely used) von Neumann languages. This model is characterized by enormously long streams of fetch/compute/write cycles, and is inherently sequential.

By contrast, in *concurrent term rewriting*, data has a graph structure, and programs are sets of rewrite rules. A *rewrite rule* consists of two templates, one describing substructures to be modified, and the other describing what they should be replaced by. In principle, all possible rewrites can be executed simultaneously, at all possible data sites (see Section 2 for more detail); however, in practice, we will implement some form of multi-grain concurrency. This model of computation supports the functional, object oriented and relational paradigms, as well as their combinations, and can effectively exploit any inherent program concurrency. For example, in object oriented programming, data accesses are sequentialized only when required for correct behavior; otherwise concurrent execution is allowed.

The RRM and its model of computation also support programs in conventional imperative languages, but this seems less desirable, because these languages have many inherently sequential features that restrict opportunities for concurrency; also, their tendency to encourage the undisciplined use of global variables and obscure side effects makes their programs harder to write, read, debug and modify. Conventional concurrent programming languages fare better, but their programs remain difficult to write, read, debug, and (especially) to modify and port to new machines. However, we should not forget that an enormous amount of software has already been written in conventional languages.

1.4 RRM Architecture

The RRM is a massively concurrent machine that realizes concurrent term rewriting in silicon, using revolutionary architecture but conventional electronic technology. The design avoids the so-called von Neumann bottleneck by using a custom VLSI chip that processes data where it is stored. The cells in a given ensemble share a single controller, so that execution is SIMD for each chip. Local communication predominates, since rewrites require only local connectivity. The following sketches our current prototype RRM design:

1. a **cell** holds one data node and its structural links, and also provides basic processing power;

2. a **Rewrite Ensemble** (RE) is a regular array of cells on a single VLSI chip, with wiring for local data exchange; one RE might hold about a thousand cells plus a shared controller and some interface circuitry;

3. a **board** might contain about a hundred REs, some backup memory, and an interface microcomputer;

4. a complete **RRM** prototype might have about ten boards, with a general connection network and a conventional minicomputer for storing rules, balancing load, and remote communication.

A single RE yields very fast fine-grain SIMD rewriting, but RRM execution is coarse-grain at the board level, since each RE independently executes its own rewrites on its own data,

Figure 2: The Tree of a Term

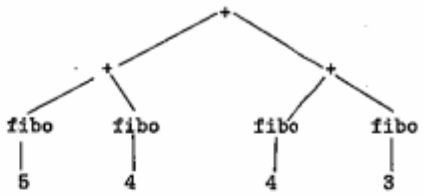

Figure 3: Result of Some Concurrent Rewriting

occasionally communicating with other ensembles. This realization of multi-grain concurrency yields high performance without sacrificing programmability. Our simulations of the RRM architecture at various levels of detail have been extremely encouraging. Our companion paper on RRM architecture [16] gives much more detail.

The RRM is intended as a general purpose computational engine, and its flexibility is one of its strong points. But, like any machine, it is more suitable for some applications than for others. Certainly, symbolic computations are very suitable, especially when there is much parallelism. Hardware simulation falls well within this class; natural language processing, "intelligent" databases, theorem proving, and expert systems are other examples. Originally, we thought the RRM would not be especially impressive for numerical computation, but recent research has shown that certain redundant representation data structures for numbers can very efficiently exploit RRM capabilities, for example, with arbitrary precision arithmetic [22].

2 Concurrent Term Rewriting

In the concurrent term rewriting model of computation, **data** are terms, constructed from a given set of operation and constant symbols, and **programs** are sets of equations that are interpreted as left-to-right rewrite rules. The left- and right-hand sides of an equation are both terms constructed from **variables** as well as operation symbols and constants. A variable can be instantiated with any term of appropriate sort, and a set of instantiations for variables is called a **substitution**.

Term rewriting (or **reduction**) has two phases: first, **matching**, which finds a substitution, called a **match**, that yields a subterm of the given term when applied to the lefthand side of the rule; and second **replacing** that subterm by the corresponding substitution instance of the righthand side of the rule. For example, matching the lefthand side of the conditional rewrite rule (from a program for Fibonacci numbers)

```
(*) fibo(N) = fibo(N-1) + fibo(N-2) if 2 <= N
```

to `fibo(6)` in the term `(fibo(6) + fibo(5)) + 0` (which is represented by the tree shown in Figure 2) succeeds with the variable `N` instantiated to the constant 6. Since the condition `2 <= 6` is satisfied, the subterm where the match occurs (called the **redex**, which is `fibo(6)` in this case) is replaced by the corresponding substitution instance of the righthand side. In this case, the original term is rewritten to

```
((fibo(6-1) + fibo(6-2)) + fibo(5)) + 0 .
```

Rewriting at only one location at a time is called **sequential term rewriting**. If the rewrite rule (*) had been applied to `fibo(5)` instead, one step of sequential rewriting would have yielded

```
(fibo(6) + (fibo(5-1) + fibo(5-2))) + 0
```

but the rule (*) could also have been applied simultaneously to both `fibo(6)` and `fibo(5)`, yielding

```
((fibo(6-1) + fibo(6-2)) + (fibo(5-1) +
   fibo(5-2))) + 0
```

in just one step. This is called **parallel term rewriting**, where just one rule is applied several places at once, and it is what a single Rewrite Ensemble realizes. More generally, true **concurrent term rewriting** allows the application of several different rules at several different sites at once; this is what a multi-ensemble RRM realizes. For example, first applying the rule (*), and then concurrently applying both the rule `N + 0 -> N` and a rule for subtracting 1, transforms the original tree into the tree shown in Figure 3, in two steps of concurrent rewriting.

It is worth remarking that with concurrent term rewriting, the number of steps required to compute `fib(n)` with the rule (*) is linear in `n`, whereas it is exponential for sequential rewriting. This simple example illustrates that concurrency is inherent in concurrent term rewriting, and that no explicit concurrency constructs are required at the language level to achieve it or to describe it. However, sometimes a clever choice of data structure or of rewrite rules is needed to achieve optimal performance.

[7] and [5] show that a certain second order equational logic is a natural extension of standard first order equational logic. It is exciting that there is a corresponding natural extension of term rewriting, called **extended rewriting**, that can be realized on the RRM just as easily and efficiently as standard rewriting. This is the most significant new idea in this paper, since extended rewriting has important applications to implementing object oriented programming (Section 4.2) and unification (Section 5.2). The idea is simply to allow variables that can match operation symbols. For example,

```
m(X(a(A),b(B)),D) = X(a(A + D),b(B + D))
```

is similar to a rule in Section 4.2, with `X` matching operation symbols of appropriate arity, with `A,B,D` matching subtrees as usual, and with `a,b` unary operation symbols. Note that this is not the most general form of second order rewriting, since (as far as we know) only special cases can be implemented very efficiently in VLSI.

Two additional topics deserve mention. The first is **sharing**, which permits a common substructure of two or more given structures to be shared between them, rather than requiring that it be duplicated. This leads to dag's (directed acyclic graphs) rather than just trees. The second topic is **evaluation strategies**, which are annotations that impose restrictions on concurrent execution; these can be used to improve the performance of parallel computations. Under relatively mild assumptions, such strategies do not change

the semantics of functional computations. However, for applications where concurrency is essential, evaluation strategies can be used for concurency control. For example, some further extensions to this concept support systems programming [9].

3 Functional Programming

This section gives a brief overview of the OBJ functional programming language, and then indicates how it is implemented on the RRM.

3.1 OBJ

OBJ [14,2,3] is a declarative functional programming language with semantics based upon equational logic. It is well known that initial algebra semantics is correctly implemented by term rewriting under certain simple assumptions (this was first proved in [6]), and [9] shows that concurrent term rewriting is also correct under the same assumptions. OBJ has no explicit constructs for creating or synchronizing parallel processes. Rather, the parallelism of an OBJ program is *inherent* in the program itself. OBJ was also designed to directly embody various modern software engineering techniques, rather than provide them indirectly in an associated environment having separate conventions and notations. These features include:

1. **User-definable abstract data types**, not limited to constructors, as in most functional languages.
2. **Parameterized programming**, to support software reuse and wide spectrum integration of design, documentation, rapid prototyping, and specification, with
   - powerful "tunable" *generic modules* that go far beyond Ada's generics or mere functional composition, and are powerful enough to give the power of higher order programming without its difficulties in understandability and verification [5],
   - *theories*, which describe semantic as well as syntactic properties of modules and module interfaces,
   - *views*, which assert semantic properties of modules, and
   - *module expressions*, which support programming-in-the-large, by describing how to build complex subsystems from previously defined modules, and then actually build them when evaluated.
3. **Subsorts**, which support multiple inheritance, exception handling, partial functions, and operation overloading in an elegant way.
4. **Pattern matching modulo equations**, including the associative, commutative, and identity laws, which greatly increases the power of matching, and hence the expressiveness of the language.
5. **Module hierarchies**, whereby old modules may be imported into new modules.
6. **Evaluation strategies**, which avoid enslavement to any fixed evaluation strategy, such as eager or lazy, and thus allow greater efficiency in both time and space.

```
obj FIBO is protecting NAT
 op fibo : Nat -> Nat .
 var N : Nat .
 cq fibo(N) = N if N < 2 .
 cq fibo(N) = fibo(N - 1) +
     fibo(N - 2) if 2 <= N .
endo
```

Figure 4: Fibonacci Code in OBJ

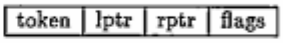

Figure 5: The Logical Structure of a Cell

7. **Very simple denotational semantics**, given by the initial algebra of the equations in a program.

OBJ has been rather extensively studied from both theoretical and practical viewpoints [14,2,8,3], and there are now several implementations besides OBJ3 at SRI International, including one from the Washington State University, three in Great Britain, one in Italy, and one in Japan. The British project at UMIST (University of Manchester Institute of Science and Technology) was supported by Alvey, and involved a rather extensive set of experiments, which clearly demonstrated the value of OBJ for practical software engineering applications; a version of UMIST-OBJ is now available as a commercial product in Britain, and another is being developed by Hewlett-Packard in Bristol, England.

3.2 A Simple Example

We use the simple program for Fibonacci numbers given in Figure 4 to illustrate some basic features of OBJ. The most basic OBJ entity is the **object**, a module encapsulating executable code. The keywords `obj ... endo` delimit the text of an object. Immediately after the initial keyword `obj` comes the object name, in this case `FIBO`; then comes a declaration indicating that the built-in object `NAT` is imported. This is followed by declarations for the new sorts of data (in this case there are none) and the new operations (in this case, `fibo`), with information about the sorts of arguments and results (here, both are `Nat`). Finally, a variable of sort `Nat` is declared, and two equations are given; the keyword `cq` indicates that these are conditional equations (unconditional equations use the keyword `eq`). `<` is the "less than" predicate, and `<=` is the "less than or equal" predicate; these are imported from `NAT` along with the addition and subtraction operations.

3.3 Implementation on the RRM

Before describing how to implement OBJ on the RRM, we need more information about the RRM design. The RRM has been designed *hierarchically*, that is, as a series of models, each more concrete than the one above. The highest levels are actually semantic rather than architectural; for OBJ, these models are equational logic and term rewriting, the former providing a denotational semantics, and the latter an operational semantics. We now discuss the most abstract architectural model for the RRM, the **cell machine**, consisting of an arbitrary number of cells, each with three major

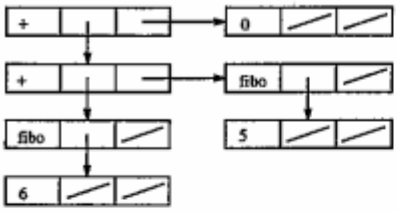

Figure 6: The Cell Representation of a Term

```
obj FIBO is sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  op fibo : Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N).
  eq fibo(0) = 0 .
  eq fibo(s(0)) = s(0) .
  eq fibo(s(s(N))) = fibo(s(N)) + fibo(N) .
endo
```

Figure 7: Peano Fibonacci Code in OBJ

registers and an arbitrary number of "flags," which can be "set" or "unset" (i.e., "up" or "down"). The **token** register stores the "content" of a cell, while its **left** and **right pointer** registers each give the location of another cell (or else are empty)[1]. The flags are used to store local status information during matching and rewriting. Figure 5 shows the *logical* structure of a cell; of course, the *physical* structure is more complex, but our subsequent diagrams actually simplify further and omit the flags. This model assumes that each cell can communicate directly with any other cell; [16] explains how the actual RRM realizes the same logical power using only local connectivity.

It is evident how to represent a binary tree (or dag) in such a cell machine; for example, Figure 6 shows the tree of Figure 2. We now consider how to implement rewriting with SIMD streams of *microinstructions* that are broadcast simultaneously to all cells from the central controller. The following are some typical microinstructions: set a certain flag if the token has a certain value; fetch a token (or pointer) from another cell whose location is known; and set the token to a certain value if a certain flag is set. In this model, every instruction is interpreted and (if applicable) executed in each cell using only information that is *local* to that cell.

A given rewrite rule is implemented by first identifying instances of its lefthand side in a **matching phase**, and then **replacing** each matched pattern by the corresponding righthand side. Although arithmetic for the natural numbers is provided by the RRM hardware, the following discussion will use a basic Peano representation, with constructors the constant `0` and the unary successor operation `s`, as shown in Figure 7. Then the rewrite rule

`fibo(s(s(N))) = fibo(s(N)) + fibo(N)`

from Figure 7 can be implemented by first identifying each cell that contains the token `fibo`, and then checking that the cell indicated by its left pointer contains a successor that points to another successor (in practice, this check could be done bottom-up).

Once the instances of the pattern `fibo(s(s(x)))` are identified, then replacement can begin; for example, we may replace the token `fibo` at the root of the pattern by `+`, replace its left pointer by a pointer to its `s(x)` cell, and set its right pointer to the `x` cell. See Figure 8. Notice that there is now one less pointer to the first `s` cell, so that it should be collected as garbage if there are no other pointers to it. Also notice that a dag structure has been created from what might previously have been just a tree structure. The following *copy rule* expresses an important restriction on modifying cells during term rewriting:

> If there is more than one active pointer to a cell, then it cannot be modified, and must instead be copied, unless it is the root of the redex.

Many questions might occur to the reader who has followed this discussion closely. In general, these fall into one of the following classes:

1. **Architectural questions**, such as "How to realize arbitrary logical connections between cells that are only locally connected physically?" or "How are the microinstructions implemented?" Such questions are discussed in [16].

2. **Model of computation questions**, such as "What happens if two instances of the same rule want to modify the same cell?" or "What happens if more than one ensemble must cooperate on a rewrite?" Such questions are answered in [9].

3. **Detailed programming questions**, such as "How to compute `fibo` with optimal efficiency on the RRM?" Some such questions are answered in [20], while others must be deferred to a future paper.

4 Object Oriented Programming

The recent history of programming languages can be seen as an attempt to obtain the advantages of imperative programming without its disadvantages, while adding new features to encourage better programming style and better support for programming-in-the-large, program maintenance, etc. A major problem with traditional imperative programming style is its obsessive and obscure use of globally shared structures, particularly global variables; this not only makes programs difficult to understand and maintain, but is also a particular disadvantage for distributed computing, since global variables cannot reflect and exploit distributed memory. In our view, the essence of object oriented programming is not inheritance (multiple or otherwise), nor is it message passing (which is after all just a metaphor for procedure calling), but rather it is the organization of memory into *local persistent objects*, as opposed to a single global store. Such a programming style makes programs easier to understand and to modify, as well as more appropriate for distributed computing. It is significant that object oriented programming arose in a language designed for simulation, so that its concepts are motivated by the physical world, with its natural intuitions of hierarchical and distributed structure.

[1] An n-ary source level operation symbols is translated into $n-2$ binary operations for $n > 2$, so that binary cells are sufficient.

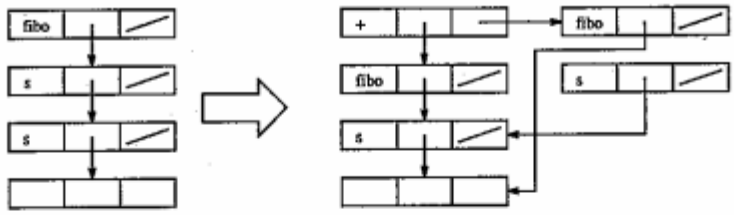

Figure 8: Rewriting a Cell Representation

4.1 FOOPS

FOOPS [13] was designed to be a simple, yet expressive and efficient general purpose object oriented language that embodies the various modern software engineering techniques developed for OBJ. We chose to combine object oriented programming with OBJ-style functional programming rather than with imperative programming, because we wanted to restrict features that change memory to *methods* that only update local properties of objects. By contrast, Common Loops builds on Lisp, which has many imperative features with global side effects, such as `setq`, `rplaca`, and their ilk, that encourage an undisciplined programming style.

In FOOPS, objects, abstract data types, methods and attributes are all defined in a declarative functional style. This gives FOOPS a simple syntax and semantics, and makes it comparatively easy to read, write and learn. FOOPS is also relatively easy to reason about, since it is based on a formal logical system; indeed, [13] gives what seems to be the first ever rigorous semantics for object oriented programming. Moreover, we have designed a graphical programming interface with which the user can directly manipulate icons that represent objects, using a mouse; this leads to a programming style that is almost "physical" in its intuitive impact [4].

OBJ is a proper sublanguage of FOOPS, used to define the abstract data types that provide values and the functions that manipulate these values. In addition, FOOPS allows declarations for **classes**, **attributes** and **methods**; for added clarity, classes and methods are written in italics, and keywords are underlined. Each object of a given class has a unique name, and also has values for certain attributes; these values are usually from abstract data types, but may also be from other classes ([13] gives details of FOOPS' powerful object-valued attribute facility). FOOPS distinguishes between ok axioms and error axioms, which respectively describe normal and exceptional behavior; the basis for this distinction in order sorted algebra is given in [13].

We illustrate FOOPS with the following simple module for bank accounts. Objects in this example are bank accounts with two attributes. The first, `bal` gives the balance of an account as a `Money` value, assuming that a representation for money (with a positive or negative sign) has already been given in the module `MONEY`, and that the sort `Money` has a subsort `Pmoney` for positive amounts of money. The second attribute `hist` is a history of the transactions performed on the account since its creation, represented as a list of money amounts. This list data type is imported into the `ACCT` module by applying the generic `LIST` module to the data sort `Money` and renaming its sort `List` to `Hist`. Two methods can modify accounts, *credit* and *debit*, with the effect of incresing or decreasing the balance, and of appending the corresponding amount (with appropriate sign) to the history list. There are also error axioms to handle overdraw exceptions.

```
omod ACCT is class Acct
 protecting LIST[Money]*(sort List to Hist)
 attrs bal : Acct -> Money
       hist : Acct -> Hist
 error overdraw : Money -> Money
 methods credit, debit : Acct PMoney -> Acct
 ok-axioms
  bal(new(A)) = 0 .
  hist(new(A)) = nil .
  bal(credit(A,M)) = bal(A) + M .
  hist(credit(A,M)) = app(hist(A),M) .
  bal(debit(A,M)) = bal(A) - M if bal(A) <= M .
  hist(debit(A,M)) = app(hist(A),-M)
     if bal(A) <= M .
 err-axioms
  bal(debit(A,M)) = overdraw(M) if bal(A) < M .
  hist(debit(A,M))= app(hist(A),overdraw(M))
      if bal(A) < M .
endo ACCT
```

The first two axioms can actually be omitted by invoking the FOOPS "principal constant" convention, which says that the initial value of an attribute is the "principal" constant of its abstract data type, if there is one.

4.2 Implementation on the RRM

We now discuss the implementation of FOOPS objects, attributes and methods by (extended) rewrite rules, using the above bank account example. An object, such as `Johnson-Acct`, is internally represented as a term

```
Johnson-Acct(bal:(500),
    hist:(200 -100 300 -100 200))
```

with top operation symbol the name of the object, and with as many subterms as attributes. For an attribute a the corresponding subterm is of the form $a: t$ with $a:$ a unary operation symbol having t as its argument. In general, an object O in a class with attributes $a_1, ..., a_n$ has the form

$$O(a_1: (t_1), ..., a_n: (t_n))$$

and the value $a_i(O)$ of the attribute a_i for an object O is obtained by applying the rewrite rule

$$a_i(O(a_1: (x_1), ..., a_n: (x_n))) = x_i.$$

For example, this gives `bal(Johnson-Acct) = 500` for an account in the state described above.

Method application is only slightly more complex. The axioms for a FOOPS method declare the effects on each of the object's attributes[2]. For example, the axiom

[2]Although this paper only discusses *basic methods* whose axioms have

`bal(`*credit*`(A,M)) = bal(A) + M`

declares that the new balance is the old balance plus the amount being credited. In general, a method m with axioms of the form

$a_i(m(O, \vec{y})) = t_i(a_1(O), ..., a_n(O), \vec{y})$

translates into a single rewrite rule of the form

$m(X(a_1\colon (x_1), ..., a_n\colon (x_n)), \vec{y}) =$

$X(a_1\colon (t_1(x_1, ..., x_n, \vec{y})), ..., a_n\colon (t_n(x_1, ..., x_n, \vec{y})))$.

This rule is second order, involving a variable X that ranges over the operation symbols that correspond to the names of the objects in the given class. For the *credit* method, the corresponding rewrite rule is

```
credit(X(bal:(B),hist:(L)),M) = X(bal:(B + M),
    hist:(app(L,M))).
```

It is fortunate that the same style of broadcasting microinstructions to RRM cells that is used for ordinary first order rewriting will also implement this restricted form of second order rewriting. Thus, it is straightforward to implement FOOPS on the RRM. The following points summarize the differences between implementing objects and implementing values:

1. **Objects persist**, and can only be destroyed by application of a `delete` command.
2. **Objects are locked for method application**, to ensure object integrity. This is realized by allowing only one match attempt to succeed when several instances of a method refer to the same object. (There is no problem when instances of different methods refer to the same object, because the RRM executes in SIMD mode locally.)
3. **Copying of objects is forbidden**, to ensure object uniqueness.

It is remarkable that these restrictions actually *simplify* ordinary term rewriting; for example, the third condition says that we don't need to enforce the "copy rule" of Section 3.3 for objects. To enhance efficiency, each object may be kept in a fixed location, with a global address that includes the ensemble and the specific cell where the (root of the) object resides; such an address can also be used as the internal name of the object. Also, all objects of the same class should be kept together in one or more ensembles which store the rules for the methods and attributes of the corresponding class. For the purposes of implementation on the RRM, imperative programming can be considered a degenerate case of object oriented programming.

5 Relational Programming

It is widely recognized that the relational paradigm is especially suitable for problems that involve search and/or deduction; typical application areas are natural language processing and expert systems. Since pure Horn clause logic is not powerful enough to support truly practical programming, the RRM project has chosen to investigate more powerful logics, rather than to graft extralogical features into Horn clause syntax. The results of our explorations include designs for the languages Eqlog and FOOPlog and some initial ideas on how to implement them, as discussed below.

the form stated, axioms for so-called *derived methods* may involve other methods in their righthand sides [13].

5.1 Eqlog

Eqlog combines the functional and relational programming paradigms, and also provides the same parameterization and wide spectrum capabilities as OBJ and FOOPS. Like these languages, Eqlog is based on a rigorous order sorted logic that provides multiple inheritance and a precise initial model semantics. Like FOOPS, Eqlog is a proper extension of OBJ. However, instead of adding classes, methods, and so on, Eqlog adds only one basic thing to the syntax of OBJ, namely *predicates*. To achieve semantic consistency, equality is now regarded as a rather special predicate that is always interpreted in models as *actual identity*. The logic for this is quite well known; it is Horn clause logic *with equality*, and there are rules of deduction with completeness and initiality theorems [12]. In this regard, the contribution of our original (1984) Eqlog paper has been rather widely misinterpreted: the main point was not so much the suggestion to use narrowing in the operational semantics of Eqlog, but rather the suggestion to use the initial model semantics of Horn clause logic with equality as a criterion for the *correctness* of *any* proposed implementation, and to use initiality *also* for the semantics of built in types (i.e., for what is now called Constraint Logic Programming), as further developed in [12]. From this viewpoint, the narrowing algorithm merely provides an existence proof that certain classes of programs can be implemented. The problem of finding an *efficient* implementation for some sufficiently rich subclass of Eqlog programs remains the subject of much current research. However, the initial model semantics of Horn clause logic with equality remains the right criterion for correctness of proposed algorithms. For practical purposes, one might choose to implement Eqlog with the restiction that only syntactic equality between terms involving constructors is allowed in Horn clauses and in queries involving predicates, but with arbitrary user definable equations for defining functions and doing functional computation; such an implementation could also provide powerful built in types, making it a modular Constraint Logic Programming Language. Of course, the abstract data types defined by constructors can be seen as another built in type.

The operational semantics of Eqlog divides naturally into two algorithms, one for solving systems of equations, and the other for searching. The first algorithm generalizes standard, syntactic unification, the extreme case being *universal* or *semantic unification*, while the second differs little from the usual Prolog-style implementation of search for SLD-resolution, except that it exploits the opportunities for concurrency which the RRM provides. These algorithms are discussed in Sections 5.2 and 5.3 below, respectively.

5.2 Unification

Unification and term rewriting are closely related; in particular, the matching phase of rewriting is a special case of unification. What may be more surprising is that unification can be naturally implemented by rewriting, so as to exploit parallelism in a natural way. As in the Martelli-Montanari unification algorithm [18], we represent both unification problems and their solutions as sets of equations, and we give rules that transform the former into the latter[3];

[3]Herbrand's original work on unification can also be seen as an algorithm of kind.

in fact, the solutions are reduced forms under the given rules. This subsection illustrates the approach with a very simple algorithm without the occur check. Some other unification algorithms are briefly discussed at the end of the subsection.

We can consider an equation between terms t and t' to be another term $t \equiv t'$, with the binary infix operation $\equiv$ assumed commutative[4]. Next, we can group several equations together into a *system* of equations, represented by a term of the form

$\{t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n\}$.

Then solving a system of equations corresponds to evaluating a term of the form

$solve\{t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n\}$

using the rewrite rule

$solve\{L\} = \{elim(L)\}$

where *elim* is an auxiliary operator for variable elimination whose meaning is defined below. Our rewrite rules for unification use associative pattern matching on lists of equations to ease the exposition, and sometimes leave the sorts of the variables implicit, for example, L above ranges over lists of equations, and the variables I, I' below will range over identifiers, of sort `Id`.

We first give rules for "decomposing" equations, using the power of second order rewriting. Assume that each operator name has a fixed arity (zero for constants) and that arities are bounded by a small number (although these assumptions are realistic, they are used here only to simplify the exposition). Then the *decomposition rules* are

$X(x_1, \ldots, x_n) \equiv X(y_1, \ldots, y_n) = (x_1 \equiv y_1) \wedge \ldots \wedge (x_n \equiv y_n)$.

$X(\vec{x}) \equiv Y(\vec{y}) = fail$ *if* $X \neq Y$.

where the variables X and Y are second order and match operation symbols, and where *fail* is a constant obeying the rule

$\{L \wedge fail \wedge L'\} = \{fail\}$.

Before explaining the rules for variable elimination, we briefly discuss the operation of replacing a variable by a term. We regard replacement as a ternary operation *let* I *be* t *in* t' with I an identifier and t and t' terms. Then the *replacement rules* are

let I *be* t *in* $I' = if\ (I == I')\ then\ t\ else\ I'\ fi$.

let I *be* t *in* $X(x_1, \ldots, x_n) =$
$X(let\ I\ be\ t\ in\ x_1, \ldots, let\ I\ be\ t\ in\ x_n)$.

where $==$ denotes syntactic identity. Replacement extends to equations (by applying the replacement to both sides) and to lists of equations in an obvious way. Now the variable *elimination rules* are

$elim(nil) = nil$.

$elim(fail) = fail$.

$\{L \wedge elim(L' \wedge (I \equiv t) \wedge L'')\} =$
$if\ (I == t)\ then\ \{L \wedge elim(L' \wedge L'')\}\ else$
$\{(I \equiv t) \wedge (let\ I\ be\ t\ in\ L) \wedge$
$elim(let\ I\ be\ t\ in\ (L' \wedge L''))\}\ fi$.

Finally, to avoid trivial equations in the solved form, we add the equation

$\{L \wedge (I \equiv I) \wedge L'\} = \{L \wedge L'\}$.

Other unification algorithms can be implemented on the RRM with similar techniques. In particular, the occur check (which Eqlog needs) can easily be added to the above unification algorithm[5]. Eqlog also needs order-sorted unification, which can actually be significantly *more efficient* than unsorted unification, due to earlier failure detection. A quasi-linear order-sorted Martelli-Montanari style unification algorithm is given in [19]. Eqlog also needs unification modulo equations. The narrowing algorithm [15] shows that this is possible, but is known to be inefficient. However, the RRM's parallelism can be effectively exploited for this problem, since narrowing combines rewriting and unification. The work of Martelli *et al.* [17] treating narrowing as a transformation of a set of equations seems suggestive in this regard.

5.3 Search

Searching for solutions to an Eqlog query should exploit RRM concurrency to explore many parts of the search space in parallel. Solving a given query Q for a particular Eqlog program can be conceptualized functionally rather than nondeterministically. To find the first n solutions of the query Q we reduce the term *show* Q *upto* n to a set of substitutions, represented as a disjunction $\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n$. One approach to implementing Eqlog in the RRM may be based up on ideas similar to those of J.A. Robinson [21] and K. Berkling [1]. However, our context is broader since it includes Horn clause logic with equality, and our functional basis is equational logic rather than lambda calculus. The key observations are:

1. A sentence of the form $\forall \vec{x} \forall \vec{y} (A(\vec{x}) \Leftarrow B(\vec{x}, \vec{y}))$ is logically equivalent to one of the form $\forall \vec{x} (A(\vec{x}) \Leftarrow \exists \vec{y} B(\vec{x}, \vec{y}))$.
2. In the initial (or Herbrand) model I_C defined by a set C of Horn clauses (possibly involving equality) [10,12], if a predicate symbol P is defined by Horn clauses of the form
   $P(\vec{t_1}(\vec{x_1})) \Leftarrow B_1(\vec{x_1}, \vec{y_1}), \ldots, P(\vec{t_n}(\vec{x_n})) \Leftarrow B_n(\vec{x_n}, \vec{y_n})$
   where the B_i's are conjunctions of positive atoms, then
   $I_C \models P(\vec{x}) \Leftrightarrow ((\vec{x} \equiv \vec{t_1}(\vec{x_1}) \wedge \exists \vec{y_1} B_1(\vec{x_1}, \vec{y_1})) \vee \ldots \vee (\vec{x} \equiv \vec{t_n}(\vec{x_n}) \wedge \exists \vec{y_n} B_n(\vec{x_n}, \vec{y_n})))$
   where $\vec{x} \equiv \vec{t}(\vec{z})$ is compact notation for a conjunction of equations equating the first variable with the first term, the second with the second, etc.
3. Since the $\Leftrightarrow$ symbol in the last formula can be interpreted as equality of terms, we can view such a formula as a rewrite rule for SLD resolution in Horn clause logic with equality.

For example, a program to compute the transitive closure TC of a binary relation R having clauses

$TC(x, y) \Leftarrow R(x, y)$
$TC(x, y) \Leftarrow TC(x, z) \wedge TC(z, y)$

is transformed into a functional program involving the rewrite rule[6]

[4] Commutative operations can be implemented on the RRM without any special difficulty.

[5] It might even be possible to perform such a check "by need" so that, say, when exploring a search tree the cost is only incurred on successful paths.

[6] This rule acts in a sense like a *closure*; its implementation should create new instances of the existentially quantified variables for each rewrite. It can thus be implemented as a (special kind of) *object*, in the sense of Section 4.1.

$expand(TC(x,y)) = R(x,y) \vee \exists z(TC(x,z) \wedge TC(z,y))$

where the meaning of the function *expand* is clarified below. Rewrite rules like the above for TC produce complex *formulas* built up from systems of equations and atomic formulas like $R(x,y)$, by repeated application of $\vee$, $\wedge$ and $\exists$. The original expression *show* Q *upto* n is reduced to a disjunction of n solutions, i.e., to n systems of equations in solved form, each solving the query Q. Solutions are extracted one by one using the rule

(†) *show* $S \vee F$ *upto* $n = S \vee$ *show* F *upto* $p(n)$

where p is the predecessor function, S ranges over systems of equations, and F over formulas. In general, an expression corresponding to a logical formula is reduced by certain auxiliary rules (in addition to those already discussed for unification), including:

1. $F \wedge (F' \vee F'') = (F \wedge F') \vee (F \wedge F'')$
   (distributivity)

2. $\exists x\{E \wedge (x \equiv t) \wedge E'\} = \{E \wedge E'\}$
   (existential quantifier elimination)

3. $\{E\} \wedge \{E'\} = solve\{E \wedge E'\}$
   (solving the conjunction of two systems of equations)

The parallel model of computation underlying the RRM supports search with "or" parallelism so that the logical completeness of Eqlog is not sacrificed. However, the exponential explosion of the search tree must be controlled, even when ample parallel resources are available. For this purpose, the *expand* function expands an atomic formula $P(t_1,...,t_n)$ into a disjunction of formulas, one for each clause having P in its head, as in the TC example above. This permits a lazy breadth first strategy, exploring deeper levels of the search tree only when no more solutions are available at higher levels; then, the rule (†) above does not match, and further expansion is initiated by rules such as:

1. *show* $F \vee F'$ *upto* $n =$
   show $expand(F) \vee expand(F')$ *upto* n.

2. *show* $F \wedge F'$ *upto* $n =$
   show $expand(F) \wedge expand(F')$ *upto* n.

3. *show* $\exists x\ F$ *upto* $n =$ *show* $\exists x\ expand(F)$ *upto* n.

4. *show* $P(t_1,...,t_n)$ *upto* $n =$
   show $expand(P(t_1,...,t_n))$ *upto* n.

5. $expand(F \vee F') = expand(F) \vee expand(F')$.

6. $expand(F \wedge F') = expand(F) \wedge expand(F')$.

7. $expand(\exists x\ F) = \exists x\ expand(F)$.

This seems a reasonable and simple way to explore the search tree, but many other strategies are possible. The RRM supports very flexible and general evaluation strategies [9] that can be applied to this problem. Also, creating an object with two attributes, one the solutions already found, and the other for the remaining search tree, and with methods for requesting additional solutions would support very natural user interactions.

5.4 FOOPlog

There is not space here for more than a few remarks about FOOPlog [13], which combines all three major emerging programming paradigms, the functional, object oriented, and relational. It appears that techniques similar to those described above will support the efficient implementation of FOOPlog on the RRM. Moreover, we believe that FOOPlog is an especially suitable language for knowledge processing, and in particular, for natural language processing [13,11].

Acknowledgements

Mr. Timothy Winkler deserves special thanks for extensive discussions and many very valuable suggestions on the ideas presented in this paper, especially on the implementation of Eqlog, and also for help with the figures. We also thank the other members of the Rewrite Rule Machine Project, Dr. Sany Leinwand, Prof. Hitoshi Aida and Prof. Ugo Montanari, with whom we have had extensive discussions of these ideas, and the other members of the OBJ team, Dr. Kokichi Futatsugi and Prof. Jean-Pierre Jouannaud, as well as Drs. Claude and Hélène Kirchner and Mr. Aristide Megrelis.

References

[1] Klaus Berkling. *Epsilon-Reduction: Another View of Unification.* Technical Report, Syracuse University, 1986.

[2] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, Association for Computing Machinery, 1985.

[3] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60, IEEE Computer Society Press, March 1987.

[4] Joseph Goguen. Graphical programming by generic example. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 209–216, International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.

[5] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Proceedings, Univeristy of Texas Year of Programming, Institute on Declarative Programming*, Addison-Wesley, 1988. To appear; preliminary version as SRI Technical Report SRI-CSL-88-1, January 1988.

[6] Joseph Goguen. How to prove algebraic inductive hypotheses without induction: with applications to the correctness of data type representations. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 356–373, Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 87.

[7] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P. Subramanyan, editor, *Proceedings of Second Banff Hardware Verification Workshop*, Springer-Verlag, Computer Science Series, to appear. Preliminary version is Technical Report SRI-CSL-88-4, SRI International, Computer Science Lab, April, 1988.

[8] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[9] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In Robert Keller and Joseph Fasel, editors, *Proceedings, Graph Reduction Workshop*, pages 53–93, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[10] Joseph Goguen and José Meseguer. Eqlog: equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363, Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[11] Joseph Goguen and José Meseguer. Logical Programming for Situation Semantics. In Mark Gawron, David Israel, José Meseguer, and Stanley Peters, editors, *Semantics of Natural and Computer Languages*, MIT Press, 1988. To appear.

[12] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 250; also, Technical Report CSLI-87-91, Center for the Study of Language and Information, Stanford University, March 1987.

[13] Joseph Goguen and José Meseguer. Unifying object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477, MIT Press, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986; also, Technical Report CSLI-87-93, Center for the Study of Language and Information, Stanford University, March 1987.

[14] Joseph Goguen and Joseph Tardo. An introduction to OBJ: a language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189, IEEE Press, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, Eds., Addison-Wesley, 1985, pages 391-420.

[15] Jean-Marie Hullot. Canonical forms and unification. In *Proceedings, 5th Conference on Automated Deduction*, pages 318–334, Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 87.

[16] Sany Leinwand, Joseph Goguen, and Timothy Winkler. Cell and ensemble architecture of the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems*, ICOT, 1988. To appear.

[17] Alberto Martelli, C. Moiso, and G.F. Rossi. Lazy unification algorithms for canonical rewrite sytems. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*, Academic Press, to appear 1988. Preliminary version in *Proceedings*, Colloquium on the Resolution of Equations in Algebraic Structures, held in Lakeway TX, May 1987.

[18] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[19] José Meseguer, Joseph Goguen, and Gert Smolka. *Order-Sorted Unification*. Technical Report CSLI-87-86, Center for the Study of Language and Information, Stanford University, March 1987. To be submitted for publication.

[20] Ugo Montanari and Joseph Goguen. *An Abstract Machine for Fast Parallel Matching of Linear Patterns*. Technical Report SRI-CSL-87-3, Computer Science Lab, SRI International, May 1987.

[21] J. Alan Robinson. *A 'Fifth Generation' Programming System Based on a Highly Parallel Reduction Machine*. Technical Report, School of Computer and Information Science, Syracuse University, 1984.

[22] Timothy Winkler. *Numerical Computation on the RRM*. Technical Report, SRI International, Computer Science Lab, 1988. To appear, Technical Memorandum Series.