THE REWRITE RULE MACHINE, 1988

by

Joseph Goguen, Sany Leinwand, José Meseguer, and Timothy Winkler

Technical Monograph PRG-76

August 1989

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

The Rewrite Rule Machine, 1988[*]

Joseph Goguen[†], Sany Leinwand[‡], José Meseguer[‡], and Timothy Winkler[†]

Summary

This monograph consists of two papers which jointly summarize research in the Rewrite Rule Machine (RRM) Project as of about the end of 1988. Research in this period focussed on two topics: the design of very high level multi-paradigm programming languages; and an architecture for executing such languages using graph rewriting. The first paper, "Software for the Rewrite Rule Machine," gives an overview of RRM implementation techniques for functional, relational ("logic"), and object oriented languages, as well as for their combinations. This paper is nearly the same as one that appeared on pages 628-637 of the *Proceedings of the International Conference on Fifth Generation Computer Systems*, held in Tokyo in November 1988. The languages are unusual because their designs are based directly on logic, and nothing has been allowed to compromise their basis in logic. The second paper, "Cell, Tile and Ensemble Architecture of the Rewrite Rule Machine," describes the quite unconventional hierarchical architecture of a custom VLSI chip, called a *rewrite ensemble*, which processes data directly in memory, in SIMD mode. A complete RRM consists of many independent rewrite ensembles connected over a network. This paper is a very substantial modification of one appearing on pages 869-878 of the same Proceedings.

[*]Supported by Office of Naval Research Contracts N00014-85-C-0417 and N00014-86-C-0450, NSF Grant CCR-8707155, and a gift from the System Development Foundation.

[†]University of Oxford and SRI International.

[‡]SRI International.

Contents

Software for the Rewrite Rule Machine

Cell, Tile and Ensemble Architecture of the Rewrite Rule Machine

Software for the Rewrite Rule Machine

Joseph A. Goguen[*] and José Meseguer[†]

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

[*] University of Oxford and SRI International.

[†] SRI International.

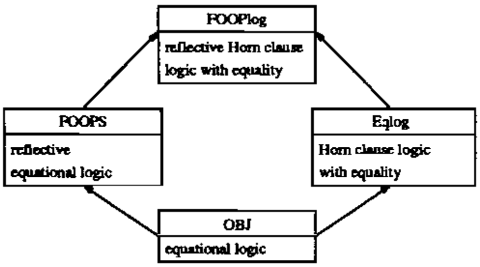
FOOPlog
reflective Horn clause logic with equality
FOOPS
reflective equational logic
Eqlog
Horn clause logic with equality
OBJ
equational logic

Figure 1: Overview of Languages

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.

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 ([12] 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 [12], which combines functional and object oriented programming;
3. Eqlog [10, 11], which combines functional and "logic" (i.e., Horn clause relational) programming; and
4. FOOPlog [12], 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 `retract`; the first is really an assignment statement (with a 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 hy 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

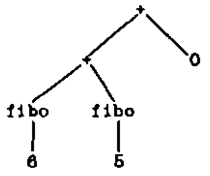

Figure 2: The Tree of a Term

rewrites on its own data, 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 archtitecture [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

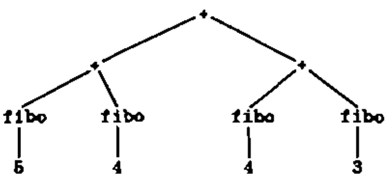

Figure 3: Result of Some Concurrent Rewriting

`((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 [8] 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 [4]), 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. **Parameterised 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 [8],
    - *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.

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, 5, 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

```
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

token	lptr	rptr	flags

Figure 5: The Logical Structure of a Cell

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 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).

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

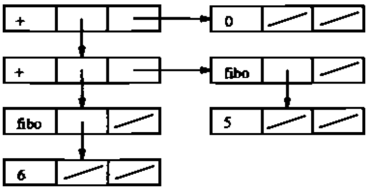

Figure 6: The Cell Representation of a Term

```
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

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

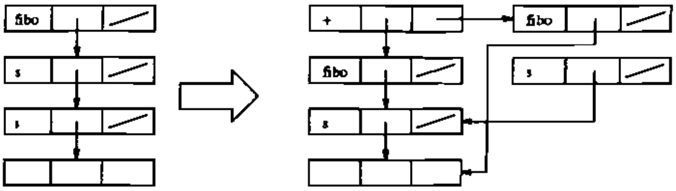

Figure 8: Rewriting a Cell Representation

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.

4.1 FOOPS

FOOPS [12] 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, [12] 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 [6].

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 ([12] 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 [12].

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 $a\colon t$ with $a\colon$ 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

$O(a_1\colon (t_1), ..., a_n\colon (t_n))$

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

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

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

`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

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

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]Although this paper only discusses *basic methods* whose axioms have the form stated, axioms for so-called *derived methods* may involve other methods in their righthand sides [12].

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.

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 [11]. 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 [11].

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]; 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

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

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

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

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'

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

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

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

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

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

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

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

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

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

$\mathit{let} \ I \ \mathit{be} \ t \ \mathit{in} \ X(x_1, ..., x_n) = X(\mathit{let} \ I \ \mathit{be} \ t \ \mathit{in} \ x_1, ..., \mathit{let} \ I \ \mathit{be} \ t \ \mathit{in} \ x_n).$

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

$\mathit{elim}(\mathit{nil}) = \mathit{nil}.$

$\mathit{elim}(\mathit{fail}) = \mathit{fail}.$

$\{L \wedge \ \mathit{elim}(L' \ \wedge \ (I \equiv t) \ \wedge \ L'')\} =$

$\quad \mathit{if} \ (I == t) \ \mathit{then} \ \{L \ \wedge \ \mathit{elim}(L' \ \wedge \ L'')\} \ \mathit{else}$

$\quad \{(I \equiv t) \ \wedge \ (\mathit{let} \ I \ \mathit{be} \ t \ \mathit{in} \ L) \ \wedge \ \mathit{elim}(\mathit{let} \ I \ \mathit{be} \ t \ \mathit{in} \ (L' \ \wedge \ L''))\} \ \mathit{fi}.$

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.

[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.

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, 11], if a predicate symbol P is defined by Horn clauses of the form

   $P(\vec{t_1}(\vec{x_1})) \Leftarrow B_1(\vec{x_1},\vec{y_1}), \ldots, P(\vec{t_1}(\vec{x_n})) \Leftarrow B_n(\vec{x_n},\vec{y_n})$

   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]

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

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

($\dagger$) *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)

[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.

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 [12], which combines all three major emerging programming paradigms, the functional, object oriented, and relationel. 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 [12, 13].

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 HélèneKirchner 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, 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. 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.

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

[6] 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.

[7] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P. Subramanyan and Graham Birtwhistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989. Also, Technical Report SRI-CSL-88-4R2, SRI International, Computer Science Lab, August 1988.

[8] 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*. to appear 1989. Preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.

[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. 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.

[12] Joseph Goguen and José Meseguer. Unifying functional, 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.

[13] Joseph Goguen and José Meseguer. Logical programming for situation semantics. In Mark Gawron, David Israel, and José Meseguer, editors, *Semantics of Natural and Computer Languages*. MIT Press, 1989. To appear.

[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, editors, 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 1988*, pages 869–878. Institute for New Generation Computer Technology (ICOT), 1988.

[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, Volume 2: Rewriting Techniques*, pages 245–273. Academic Press, 1989. 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, November 1988. Technical Note SRI-CSL-TN88-3.

Cell, Tile and Ensemble Architecture of the Rewrite Rule Machine

Joseph A. Goguen[*], Sany Leinwand[†], and Timothy Winkler[*]

Abstract: The Rewrite Rule Machine (RRM) architecture is a massively parallel *multi-grain hierarchy,* in which five different levels of organization can be distinguished. The lowest is the **cell**, which stores an individual data token. Next, a **tile** provides common communication resources for a small number of cells. Third, an **ensemble** consists of many cells, which can represent complex data structures, to which rewrite rules are applied under the direction of a common controller; each ensemble is implemented by a single custom VLSI chip. Fourth, a **cluster** organizes many ensembles to cooperate in solving larger problems, and finally, an entire RRM is a **network** of clusters. Ensembles support fine-grained SIMD parallelism, while clusters support coarse-grained MIMD parallelism. This **multi-grain parallelism** allows the RRM to exploit the fact that many typical complex computations are *locally* homogeneous but are not *globally* homogeneous. This paper describes the RRM architecture at the ensemble level, and presents some recent simulation results which suggest that one RRM ensemble has roughly 50 times the power of a Sun-3/60. We consider this very encouraging.

1 Introduction

The goals of the Rewrite Rule Machine (RRM) project are to achieve:

- *efficiency*, through multi-grain massively parallel execution;
- *generality*, supporting both symbolic and numerical computation, as well as both homogeneous and inhomogeneous computation;
- *programmability*, by supporting a wide variety of languages, from the traditional imperative to the modern declarative, including the object-oriented, logic (i.e., relational), and functional paradigms, as well as their combinations; and
- *semantic flexibility and precision*, by compiling all languages into a common model of computation called **concurrent rewriting**.

Thus, the RRM project aims to develop architectural concepts and a model of computation to bridge the gap between high level programming and massively parallel execution. This paper focuses on RRM architecture at the ensemble level, which

[*]University of Oxford and SRI International.
[†]SRI International.

implements fine-grain SIMD parallelism. Certain topics from our previous papers [5, 6, 18, 15] are only briefly summarized in this paper, but are not necessary for understanding the rest of it.

Acknowledgements

We thank Prof. Hitoshi Aida for his very valuable contributions to this paper and to the RRM project in general, and in particular, for his work on the RRM simulator, for his compiler from OBJ to the RRM, and for his help with the simulations reported in the final section. We also thank Dr. José Meseguer for his many important contributions to the work described here, ranging from programming language semantics to project management. Finally, we thank Prof. Ugo Montanari for his assistance with RRM compiler technology and for many useful discussions about parallelism.

1.1 Multi-grain Architecture and Locally Homogeneous Computation

Current generation massively parallel architectures typically fall into one of the following two classes:

- **Coarse-grain** architectures whose computations are organized into sizable tasks that only rarely need to exchange data;
- **Fine-grain** architectures whose computations are organized into small tasks that frequently need to exchange data.

Each option imposes certain limitations, either on the amount of parallelism that it can effectively exploit, or else on the types of problems for which it is suitable. For example, only computations that are very *homogeneous*, in the sense that many instances of one instruction can be applied simultaneously at many different places in the data, can make efficient use of a SIMD machine. But experience shows that many large, complex computations have many different parts with little or no overlap among their instructions. That is, large complex computations tend to be *globally inhomogeneous*. We say that computations that are locally homogeneous but globally inhomogeneous have **multi-grain parallelism**, and note that current generation architectures, languages, and models of computation are poorly suited for such computations. In particular, von Neumann machines, languages and models of computation are inadequate, because they are inherently sequential. Similarly, coarse-grain architectures can only exploit part of the parallelism available in multi-grain parallel computations. In contrast, the RRM has a hierarchical **multi-grain architecture** that exploits VLSI to pack many simple cells onto a single chip organized for fine-grain SIMD parallel computation, and then organizes many such chips for coarse-grain MIMD parallel computation. In this way, the RRM can realize the high performance potential of fine-grain parallelism for highly homogeneous computations, on the much larger class of computations that are only locally homogeneous.

1.2 Model of Computation

The model of computation plays two important roles in the RRM project:

1. It provides an abstract description of what the RRM is supposed to do, and thus a correctness criterion for its architectural design.

2. It serves as a common target language for compilers from a variety of source languages; this allows the second stage of compilation, which produces RRM ensemble controller code, to be shared among all languages.

This model of computation, called **concurrent rewriting**, adds the concepts of partitioned rewriting (see Section 1.2.3 below) and extended variables (see [10]) to the usual term and graph rewriting models, such as Dactl [4], Rediflow [14], Id [1], and Alice [12]; see [13] for an overview of graph reduction models of computation.

1.2.1 Data Representation

Although terms, such as $f(x) + g(x)$, can be represented as trees, it is preferable to represent them as graphs, for the following reasons:

- Storage can be reduced by sharing common data, such as x in $f(x) + g(x)$. Such sharing also reduces the amount of computation required, since a shared subcomputation can be performed just once.

- Replacement is much simpler for graphs, because maintaining a tree representation requires copying (possibly large) structures when variables occur more than once in the righthand side of a matched rule.

Moreover, graph structures are unavoidable for object-oriented programming, because of multiple access to objects [10]. The nodes of these graphs are labelled by (tokens that represent) operator and constant symbols; constant symbols are considered to be operator symbols with no arguments. In general, sort (i.e., type) restrictions may be attached to operator symbols.

It is easy and natural to represent familiar data structures as such labelled graphs, and although it may seem surprising at first, it is also easy and natural to represent familiar algorithms as sets of rewrite rules. Much of this follows from well-known results about implementing abstract data types using term rewriting, as embodied, for example, in the OBJ3 language [11].

1.2.2 Concurrent Rewriting

For simplicity, the following discussion is largely confined to functional computation, where there is no persistent store data. See [10] for a discussion of implementation techniques for object-oriented computation.

An RRM computation starts with a graph and a set of rewrite rules. These rules are applied until the graph is **reduced**, in the sense that no rule is applicable[1]. Each rule has **a lefthand** and a **righthand side**, constructed from operator and

[1]This restriction can be relaxed to implement so-called *perpetual* processes.

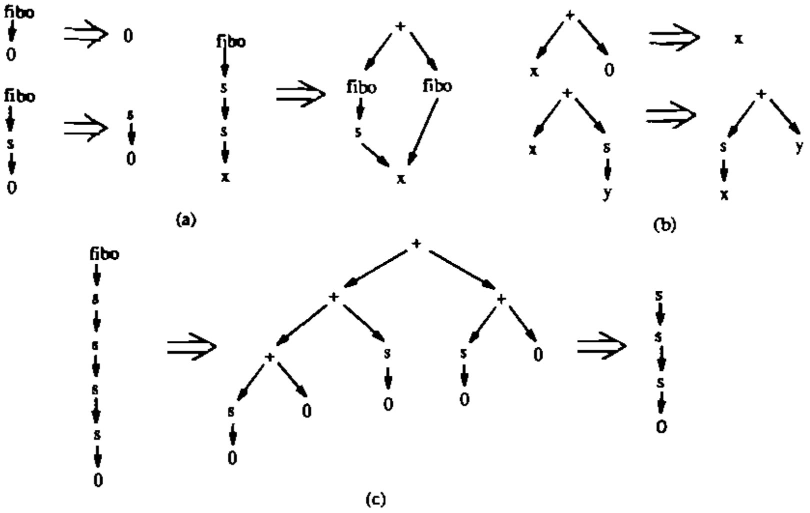

Figure 9: Rewrite Rules for Fibonacci and Addition

variable symbols. Variables can be instantiated with any graph (of the appropriate sort), and a set of instantiations for variables is called a **substitution**. Rewriting has two phases, called **matching** and **replacement**. The matching phase finds a substructure of the data graph called the **redex**, such that some substitution yields the redex when applied to the lefthand side. Then the redex is replaced by the corresponding substitution instance of the righthand side.

Concurrent rewriting allows applying multiple rules at once, at multiple places in the data, and is well adapted to massively parallel computation because no explicit constructs are required to achieve or to describe parallelism. This greatly eases programming.

Let us consider a simple example, the Fibonacci function, as defined by the equations

```
fibo(0) = 0 .
fibo(s(0)) = s(0) .
fibo(s(s(x)) = fibo(s(x)) + fibo(x) .
```

where the natural numbers are represented using only the constant 0 and the successor function s, i.e., using Peano arithmetic, so that, for example, 3 is represented by `s(s(s(0)))`. Figure 9(a) shows these rules in graphical form; notice the sharing of x in the righthand side of the third rule. Similarly, Figure 9(b) shows the rules for the + function. These rules can be applied to any graph containing `fibo`, +, s, and 0 symbols. For example, the three graphs in Figure 9(c) show an initial data graph, then the result of applying the rules in Figure 9(a) to it, and finally the result of also applying the rulee in 9(b), giving a graph with only 0 and s symbols, i.e., an integer.

A set of rules is **confluent** (sometimes called Church-Rosser) if the order of applying its rules is immaterial to the result. The fact that large confluent subsets of rules are quite common in practice permits some significant simplifications of the RRM architecture, for the following reasons:

1. The rules within a confluent set can be scheduled in any order, and any one rule can be applied at several different places in parallel.

2. Given a confluent set of rules, it is not necessary that each match of a given rule to a given substructure is replaced when it is first recognized, provided that the rule will be tried again later. This means that local data access can fail without compromising correctness.

The first property removes the sequential control straitjacket inherent in the von Neumann model of computation, while the second allows both the high performance of fast local connections, and the flexibility of remote connections.

1.2.3 Variants of the Model

It is useful to distinguish the following variants of the basic rewriting model of computation.

1. **Concurrent rewriting** allows the application of multiple rules at multiple places in the data at once. This could be implemented in a MIMD architecture where multiple controllers direct rewriting at multiple places. Although this is potentially the fastest strategy, it would be unrealistically expensive to implement in full generality.

2. **Parallel rewriting** allows the application of a single rule at multiple places at once. In traditional architectures this corresponds to SIMD execution, with one controller broadcasting instructions to many processors. Homogeneous computations are typically handled well by this strategy, which is implemented at the ensemble level of RRM architecture.

3. **Sequential rewriting** applies rules one at a time, each at a single place in the data. This corresponds to the traditional von Neumann model.

4. **Partitioned parallel rewriting** partitions a data graph into domains, within each of which parallel rewriting is performed, typically with different rules for different domains. Partitioning is dynamic, reflecting the evolution of the data.

Thus, parallel rewriting models fine-grain parallelism in a SIMD style of computation, whereas partitioned parallel rewriting, being locally SIMD but globally MIMD, models multi-grain parallelism, thus extending the efficiency of parallel rewriting to computations that are only locally homogeneous.

An **evaluation strategy** can be given as an annotation on an operator to impose specific restrictions on the order of rewriting its argument subgraphs. These annotations can be used to improve performance, and also to support the explicit programming of concurrency that is needed for systems programming [5].

1.3 Support for Programming

Programming is often the major obstacle to the effective use of a massively parallel machine. There are two common approaches to trying to overcome this obstacle:

- Reuse old programs, often written in old sequential imperative languages.
- Write new programs in new parallel imperative languages.

The first approach would have considerable merit if it were possible for a compiler to extract sufficient parallelism from "dusty decks." The current state of the art can extract moderate parallelism for homogeneous computations, often making use of some reprogramming by hand. The second approach can generally offer much greater parallelism, but often results in programs that are difficult to debug, modify, and port to other machines, because of the notorious difficulty of understanding parallel programs, and often also because of machine dependency in the programming language and/or the program.

These problems should be alleviated by our well-defined model of computation and by the techniques that we are developing to compile any language onto the RRM [10]. While we believe that the RRM can effectively support programs written in imperative languages, we also believe that appropriate declarative programming languages can exploit the RRM with greater efficiency, and at the same time offer significant advantages in programming ease and maintenance. From the hardware point of view, declarative languages do not prescribe specific orders of execution, and thus provide maximal opportunity for compilers and runtime systems to exploit parallelism. From the software point of view, the languages that we are developing for the RRM have features to support all phases of program development, from specification and design to maintenance. Moreover, they have simple syntax and semantics, and thus are relatively easy to learn and to compile. These multi-paradigm declarative languages, which extend our functional language OBJ [2, 3] to the object-oriented [9] and logic programming paradigms [7, 8], are described in [10], along with techniques for implementing them on the RRM. These same techniques also suitable for compiling more traditional languages. In particular, imperative programming can be seen as a degenerate form of object-oriented programming.

1.4 Architectural Levels

The RRM architecture can be described at the following levels of granularity:

1. A **cell** stores one node of the data structure, and can also perform simple operations. Cells are kept as simple as possible, so that as many as possible can be fit onto a single chip.
2. A **tile** provides shared communication resources for a small number of cells.
3. An **ensemble** is a single VLSI chip containing many tiles, with support for their communication needs, a common controller, and local storage for the rules that are applicable to its current data. Many cells together represent complex data structures, which are manipulated according to instructions broadcast by their controller.

4. A **cluster** interconnects many ensembles so that they can cooperate in larger computations.

5. A **network** consists of several clusters, giving a complete RRM.

Implementing an ensemble as a single VLSI chip allows fast inter-cell communication, and minimizes the delay of off-chip signal propagation, so that a high clock rate can be used. Physically adjacent tiles within an ensemble communicate over short, high speed wires. This supports rapid rewriting on data that involves only direct connections. Remote connections are handled by (automatically) relocating a remote cell to a adjacent tile when it is needed. Ensembles have a regular inter-cell communication grid, to make good use of silicon floorspace.

Although this paper focuses on the first three levels, the other two are briefly discussed below, to help the reader understand the complete RRM architecture.

1.4.1 Cluster Architecture

A cluster contains many ensembles, a large backup memory, and a connection to a conventional computer that stores the complete set of rewrite rules. Communication protocols, rule distribution strategies, and load distribution are important issues at this architectural level. Clusters are relatively independent entities that need to communicate relatively rarely. Thus they admit coarse-grain parallelism and support multi-grain parallelism.

1.4.2 Network Architecture

A complete RRM is a network of relatively few clusters, used for solving multiple or very large problems. A general purpose interconnection switch is appropriate at this level. Tokens must have long global names, because the 8 to 10 bits used inside a cluster are not sufficient to identify all the operator symbols that may occur in a larger program. Therefore translation is required when two clusters interact. A conventional compnter provides a user interface to the RRM as a whole.

2 The Cell

Cell structure is determined by the fact that cells provide physical representations for abstract nodes in a graph, as well as the basic processing power for rewriting. Previous work [16] has suggested nsing flags stored in cells to implement the matching of lefthand sides against data. Matching consists of finding cells which are the roots of subtrees that match the pattern of a lefthand side. The occurrence of a given pattern or subpattern at a particular cell is represented by setting a corresponding flag in that cell. The simplest subpatterns are tokens which are labelled with corresponding flags. Complex patterns are matched by progressively identifying larger and larger substructures, and marking them with further flags. A flag representing a larger subpattern is set in a cell when flags corresponding to its immediate subpatterns occur in its child cell(s), and it contains the required token. This process can be primarily bottom-up (from the leaves of the pattern to its root), or a combination of top-down and bottom-up, and requires that cells have access to a comparator for tokens and flags. Following a successful match, a new structure is built, which in general requires allocating new cells and storing pointers in them. When completed, this new structure replaces the original redex by overwriting the root.

These considerations suggest that cells should have the following:

- A **token** to represent the node's operator symbol.
- **Pointers** to other structures that serve as arguments to the operator symbol represented by the token.
- **Flags** to summarize the local state of computation at a cell, e.g., that a certain condition has been checked, or that a certain substructure is reduced.
- **Temporary registers** to store pointers to newly created structures.

Partitioning computation into local tasks keeps the number of distinct operator symbols small enough so that only (say) 8 to 10 bits are needed to represent tokens. Pointers can also be fairly small (say 10 bits), because the number of cells in a single ensemble is limited by available silicon area. Two pointers suffice, because operator symbols taking more than two arguments can be represented using several binary operator symbols. We have found that three temporary registers are sufficient for rapid replacement.

2.1 Cell Control

The RRM design requires that each cell:

- Obey instructions from the controller, for example, to move data between registers (token, pointer, or temporary) and to compare tokens with broadcast data.
- Attempt to connect to another cell whose address is in a register, and if successful, exchange data with that cell.
- Enter an inactive state whenever a comparison or connection attempt fails. Inactive cells do not "listen" to the controller's instructions, but can be brought back to attention by a special activation instruction.
- When inactive or free, use its resources for maintenance processes, such as garbage collection and data relocation.

2.2 Numerical Computation

Efficient numerical computation on the RRM needs capabilities at the cell level beyond those required for rewriting. Although numerical operations could be implemented from "basic principles" by representing natural numbers using only successor and zero (i.e., using Peano arithmetic), this would be much too slow. Instead, we can let cells perform simple operations on small (8 to 10 bit) numbers, including signed addition, negation, shift, and bit operations. The incremental cost over the already required equality comparison on tokens is small. Then RRM compilers can implement a complete set of arithmetic operations from these built-in cell operations. A redundant representation of arbitrary precision numbers as trees of small integers [17] permits highly parallel arithmetic operations, and very effectively exploits RRM capabilities.

3 The Ensemble

An ensemble consists of many cells, a shared controller that broadcasts instructions to them, and local storage for the rules that are applicable to its current data, all on a single VLSI chip. Cells should be interconnected with a regular mesh of fixed degree (e.g., rectangular or hexagonal), to effectively use silicon floorspace.

Data sharing is difficult for massively parallel architectures because it requires expensive coordination. The two major approaches to this problem are message passing and data passing. **Message passing** uses explicit access requests, which typically involve large overhead, and thus are most suitable for coarse-grain machines in which data transfer is sufficiently rare. **Static data passing**, as in pipelined processors and systolic arrays, moves data over prearranged paths, such that data arrives at each processor just when it is needed. Static data passing allows high performance, but at great cost in applicability and programmability.

The RRM avoids the overhead of message passing and the inflexibility of data passing by using what might be called **dynamic data representation**. In this approach, adjacent tiles are directly connected by short high-speed wires, so that matching can be very efficient when all cells are physically adjacent. When replacement requires new cells, adjacent free cells can usually be found, but if all adjacent tiles are full, then a resource allocation failure occurs. Pointers to remote cells can also arise when a newly built structure uses a substructure obtained by matching a variable, or when a remote cell is relocated without relocating its descendents.

3.1 Tile Structure

To make better use of resources, the silicon area of an ensemble is divided into a regular array of **tiles**, each of which provides communication, equality comparison, and basic arithmetic and logic operations for a small number of cells; 8 is the number currently under evaluation. Direct connections are only provided between adjacent tiles, by implementing each edge in the tessellation mesh with one communication port, supporting either duplex or half-duplex data transmission[2], shared by all cell pairs located in those two tiles. If there are multiple requests for a port, only one request succeeds and the others fail. If the mesh degree is sufficiently high (at least 4), the probability that two cells in the same tile will request communication to the same adjacent tile is low. The fact that all cells in an ensemble execute the same instruction stream also helps to minimize competition for bandwidth.

At the model of computation level, failures due to non-local connections or to low inter-tile bandwidth appear as occasional non-determinism, in the sense that

[2]A **duplex** wire can simultaneously transmit data in both directions, while data a **half-duplex** wire can only transmit data in one direction at a time.

the sequence of events is unpredictable and unreproducible. But non-determinism is natural to the concurrent rewriting model of computation, and can be made to yield determinate results for confluent rule sets, simply by repeating ensemble controller code instruction cycles until the data graph is reduced. Indeed, this unpredictability is a bonus, because it reduces the danger of deadlock. For example, if two similar data structures are located in the same part of an ensemble, then matching them may overload the available communication resources, and it is possible that neither will succeed in matching the broadcast pattern. Non-deterministic execution reduces the probability of staying forever in such a deadlocked situation. The rare remaining deadlock situations are eliminated by having the communication ports select the winning request at random whenever connection demands exceed a port's capabilities.

3.2 Ensemble Operation

A set of rewrite rules is compiled into simple ensemble controller instructions and loaded into the controller. These instructions implement the matching and replacement phases of rewriting, as well as rule sequencing. A program will first activate all cells that contain data, and then broadcast tests. A cell that does not satisfy a test is deactivated and will not execute further instructions until the next global activation; a cell that experiences a communication failure is also deactivated.

Each controller instruction is interpreted in the local context of each cell; in this respect, ensembles depart from classical SIMD design. In particular, the results of cell operations are in part determined by the contents of their local storage, including pointers to other cells. In contrast, the central controller in traditional SIMD architectures needs to know the *physical location* of each data connection. Also, cells in an RRM ensemble can temporarily activate other cells to request data.

3.2.1 Replacement

Following a successful match, the replacement phase builds a new structure to replace the redex. Since communication or allocation failures can prevent this new structure from being completed, it must be possible to abort replacement at any step without corrupting the original data. The following measures are taken for this purpose:

- The replacement is always performed by constructing the righthand side using only newly allocated cells. Pointers into the redex may arise from instantiated variables.

- The redex is replaced in a single atomic `commit` instruction that is guaranteed to succeed, but may take some time to complete. When the `commit` is finished, the newly constructed righthand side replaces the redex. This only requires changing pointers (and possibly the token) in the root cell.

- If constructing the righthand side does not succeed, due to cell allocation or communication failure, then the existing partial new structure is deallocated and the redex is left unchanged.

3.2.2 Controller Instruction Cycles and Termination

The model of computation does not in general prescribe an order for applying rules. Because one rewrite can create a structure to which another rule applies, rules must be attempted several times, to see if intervening replacements have created new redexes. Controller code should therefore be grouped into related **cycles** that are broadcast repeatedly until there are no matches. The choice of rules to place in the same cycle, their order in that cycle, and the order of broadcasting cycles, are important for efficiency. In other cases, the controller should check that a redex exists before starting a long instruction cycle for replacement.

A closely related issue is termination. The model of computation requires applying rules until there are no more redexes. The fact that communication failure can cause match failure means that instruction cycles may have to be repeated. But in our Fibonacci example, it suffices to check that there are no instances of certain operator symbols. The fact that ensembles generally execute only a subset of rules on a subset of data may sometimes require either downloading further rules or else moving some data to another ensemble; however, this is an inter-ensemble issue, and thus outside the scope of the present paper.

Checking if there are any instances of certain flags or tokens requires feedback from the cells in an ensemble to the controller. This can be implemented with a simple binary tree network that ORs signals from all cells; but obtaining results from this network will of course take more time than a single instruction.

3.3 Autonomous Processes

Dynamic data representation may introduce remote connections that cannot be used for direct data transfer. We propose to handle such situations with an **autonomous hardware process**. The basis for this is the observation that a currently inactive cell (i.e., one that has not satisfied some test in the current instruction cycle, or else is free) ignores the controller's instructions, and is therefore available to help with other useful tasks.

Because of remote connections, any request for data transfer may fail, and then start an autonomous process that will eventually relocate the connection target to a cell that is physically connected to the source by making a copy of it. This may be done by creating a message cell that "moves" toward the target cell, by having each such cell allocate another cell closer to its target, and then copy its state into that cell, and finally deallocate itself. Notice that when the target is reached, a reverse process is performed, until the original requesting cell is reached. When a cell is relocated, its connections to previously adjacent cells can become remote. The effect of autonomous data relocation is that cells move around the grid, attempting to eliminate remote connections that are needed for further computation. Figure 12 in Section 4.2 illustrates this process.

Garbage collection can also be handled by autonomous processes at the cell level.

4 Simulation

Several ensemble simulators have been written, at different levels of abstraction:

- A **concurrent rewriting simulator** takes rewrite rules as its primitive instructions. This is useful for determining the amount of parallelism potentially available in a program. Some results at this level are reported in [18].
- A **graph simulator** uses abstract graphs as data, treating all connections between cells as if they were local, so that communication never fails. This is useful for debugging ensemble controller code and the compilers that produce it.
- A **geometrical simulator** takes account of the geometry of the array of cells in an ensemble, and the communication limitations that are involved. This is useful for exploring high level design choices, to avoid errors before descending to more detailed levels.

Both graph and geometrical simulators execute ensemble controller instructions, but they do so in different ways.

4.1 Graph Simulation

We continue the example of Figure 9 by giving hand-coded controller instructions for the Fibonacci function. There are two cycles: the first, shown in Figure 10, reduces each Fibonacci symbol to a combination of plus and successor symbols; the second, shown in Figure 11, eliminates plus symbols (i.e., performs addition in Peano arithmetic).

The registers of a cell are called `token`, `left`, `right`, `temp`, `ltemp`, and `rtemp`. The RRM ensemble simulator uses a Lisp-like syntax, and include the following:

1. `(init)` activates all cells.
2. Tests, including the following:
   - `(test-token t)` checks that a cell's token is "t".
   - `(test-flag x)` checks that a cell has the "x" flag set.
   - `(test-tree a :lflag b :rflag c)` checks that a cell has the "a" flag set, and that the cells indicated by its left and right pointers have their "b" and "c" flags set, respectively; both the `:lflag b` and `:rflag c` arguments are optional.
3. Instructions that are performed by all active cells, including:
   - `(set-flag x)` sets the flag "x" (in each active cell).
   - `(set-flag x :pointer left)` sets the "x" flag of the cell pointed at by each active cell's left pointer.
   - `(unset-flag x)` unsets the flag "x".
   - `(get x y)` places "x", which may be either a token or a register name, in the "y" register of each active cell.
   - `(get x y z)`, for each active cell, places the contents of register "y" of the cell pointed at by "x" into register "z".

```
(loop (init) (test-token 'fibo)
             (set-flag 'a)
      (init) (test-token 's)
             (set-flag 'b)
      (init) (test-token '0)
             (set-flag 'c)
      (init) (test-tree 'a :lflag 'c)
             (get '0 token)
             (commit 0)
      (init) (test-tree 'a :lflag 'b)
             (set-flag 'd :pointer left)
      (init) (test-tree 'd :lflag 'c)
             (set-flag 'e)
      (init) (test-tree 'a :lflag 'e)
             (get left left ltemp)
             (get 's token)
             (commit 1)
      (init) (test-tree 'd :lflag 'b)
             (get left left temp)
             (set-flag 'f)
      (init) (test-tree 'a :lflag 'f)
             (alloc ltemp 'fibo)
             (get left left temp)
             (alloc rtemp 'fibo)
             (put temp ltemp left)
             (get left temp temp)
             (put temp rtemp left)
             (get '+ token)
             (commit 2)
)
```

Figure 10: Ensemble Controller Code for the Fibonacci Function

```
(loop (init) (test-token '+)
             (set-flag 'a)
      (init) (test-token 's)
             (set-flag 'b)
      (init) (test-token '0)
             (set-flag 'c)
      (init) (test-tree 'a :rflag 'c)
             (get left right rtemp)
             (get left left ltemp)
             (set-flag 'd)
      (init) (test-flag 'd)
             (commit 0 :pointer left)
      (init) (test-tree 'a :rflag 'b)
             (alloc ltemp 's)
             (get right left rtemp)
             (put left ltemp left)
             (commit 2)
)
```

Figure 11: Ensemble Controller Code for Addition

- `(put x y z)` places the content of register "x" in register "z" of the cell pointed at by "y".
- `(alloc x t)`, for each active cell, allocates a new cell witb token "t" and puts a pointer to it in register "x".
- `(commit c)` replaces the redex by the newly built structure, and transfers `ltemp` to `left` and `rtemp` to `right`. The numerical argument is used to count acknowledgements of reference count increments required to finish building the instantiation of the RHS of the rule; at least c must be received before the commit will finish and delete the old LHS structure. The instruction `(commit 0 :pointer left)` first overwrites the active node with the contents (`token`, `left`, and `right`) of the cell pointed at by its `left` register and then does a commit.

4.2 Geometrical Simulation

At this level of simulation, all cell-to-cell communication requests are checked to see if the source and target cells are in adjacent tiles. If so, then the instruction is immediately performed (barring failures); otherwise, an autonomous process is started to move the target cell into a directly connected tile.

Simulation results can be shown graphically with grid "snapshots," and Figure 12 shows selected pictures for our ongoing Fibonacci example, assuming a 4 by 4 rectangular grid of tiles, i.e., that each tile connects to four neighbors, over half-duplex wires. These snapshots show the token of each allocated cell, with its arguments indicated by arrows that correspond to its left and right pointers.

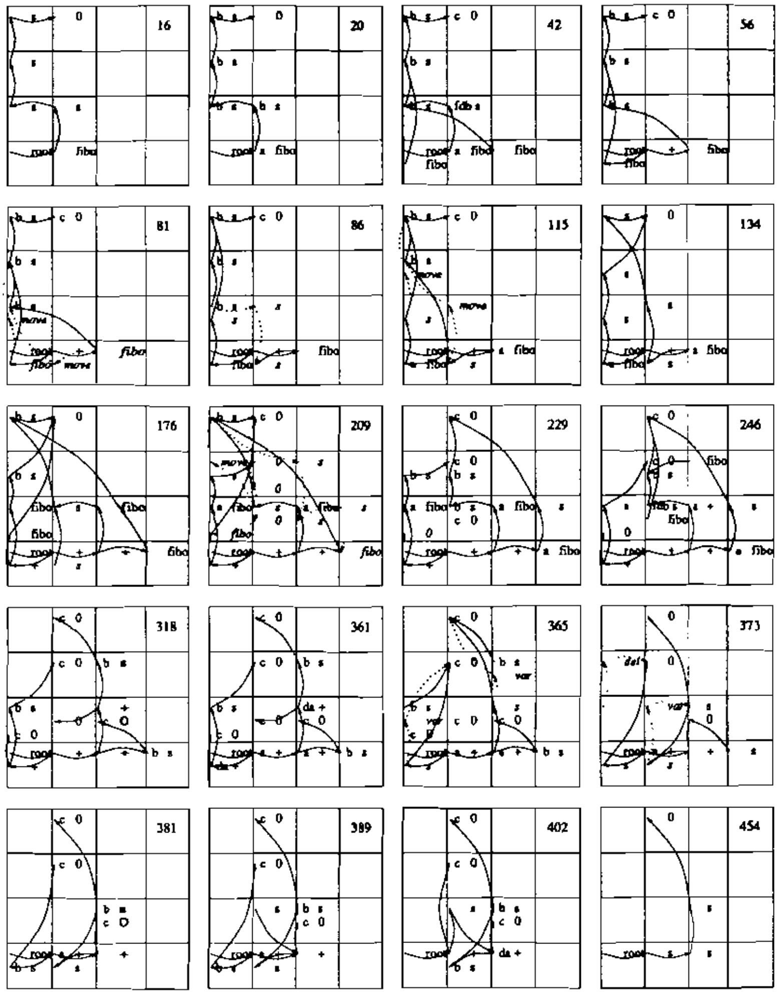

Figure 12: Geometrical Simulation Snapshots

The snapshot in Figure 12 at clock time 16 shows the initial structure. Tbe cell with token "root" points to the root of the data graph, which has the token fibo with an argument a sequence of s operators applied to 0. Although each of the 16 tiles can accommodate several cells, cells are normally evenly distributed.

First, the instruction cycle of Figure 10 is applied. At time 20 the interesting tokens are being identified by having the appropriate flags set. Since an instance of fibo(s(s(x))) is found, the ensemble proceeds to build a new instance of the corresponding righthand side. The snapshot for time 42 shows the original fibo cell with two adjacent cells allocated that also contain fibo tokens, in one case pointing to the original argument with one s removed, and in the other case with two removed. The redex is replaced by the new structure in a single commit instruction, with the result shown in Figure 12 at time 56.

Since this structure contains remote pointers, the matching process fails, and requests for relocating remote cells are issued. Figure 12 at time 81 indicates these requests (actually their messengers) by dotted arrows. Both fibo cells request their arguments. The details of the algorithm for relocating cells are too complex for this paper, but the result is shown at time 86, where one of tbe moved arguments has been located "above" the fibo operator, while the second one is still moving closer. Completing these relocations causes new pointers to be remote, which in turn causes further relocation requests. These can be seen at time 115, attempting to bring the next level of fibo argument cells closer.

By time 134, there is enough adjacent structure for further rewriting. As with the first rewrite, the redexes are identified and new structures are allocated for replacing them. Notice that two redexes are rewritten in parallel. The graph after the replacement is shown at time 176. Tbe initial fibo token has now spawned four new fibo tokens.

As before, the appropriate cells are activated, and new righthand structures are built. But since there are four redexes, there is a high probability that some requests will fail due to resource limitations. For example, the snapshot for time 209 shows three separate relocation requests independently finding their targets. A bit later, enough cells are adjacent so that even more rules can match. At time 229, one fibo(0) has heen replaced by 0, and all other fibo arguments are in adjacent locations. Another rewrite step is shown at time 246, when another two instances of fibo are rewritten. Finally, in the snapshot for time 318, all instances of fibo have been eliminated, and the graph is reduced with respect to this rule set. The resulting data structure represents the graph shown in the middle of Figure 9(c).

Next, the instruction cycle for addition (see Figure 11) is broadcast. By time 361, all + and s cells have been flagged, and by time 365, two instances of + are heing rewritten. Things proceed much as in the first cycle for a while. Tbe snapshot at time 373 shows some of the messages used during commit. At time 381, two + tokens have heen eliminated, and the remaining two have their arguments in adjacent locations. Their rewriting is shown at times 389 and 402. By time 454, the data structure is fully reduced, and the result represents 3, as expected.

A number of other programs have also been run on the geometrical simulator, including additional versions of Fibonacci, matrix transposition, bubble sort, and a highly parallel tree sort. These simulations have validated our algorithms for match and replacement, and have also shown that our techniques for resource management do not require unacceptable overhead.

5 Summary of Novel Features

This section summarizes some of the unusual architectural concepts that have been developed by the RRM project.

1. **Combine processing, storage, and communication** at the cell and tile levels, to avoid the memory access bottleneck of von Neumann architectures and the memory latency problem of dataflow architectures.

2. **Locally interpreted code.** The instructions broadcast to cells are interpreted locally, by each cell in its own context. This differs significantly from traditional SIMD design, in which the central controller uses the *physical location* of data in its instructions. Local instruction interpretation allows much greater flexibility, which greatly simplifies compilation, and also improves efficiency, because requests for scarce resources are resolved in the specific context of each cell, rather than by the controller.

3. **Fast local connections.** Connect physically adjacent tiles in an ensemble by short, high speed wires. This allows very fast operations when all the cells involved are adjacent.

4. **Non-critical scheduling.** Since the model of computation requires that rules are executed until the data is fully reduced, the rules must be executed repeatedly; but for many large subsets of rules, the order is arbitrary. This permits a rewrite to be abandoned whenever convenient, provided its redex is left in a consistent state. This in turn allows the RRM to make use of its extremely fast local connections.

5. **Autonomous processes.** Any cell that is either inactive or free during a particular instruction can use its resources for other worthwhile tasks, without hindering normal rewriting. Such tasks may include garbage collection and cell relocation.

6. **Dynamic data relocation.** A cell's request for access to data in another cell will only succeed if the other cell is adjacent. This can be accommodated by (autonomously) relocating remote cells to adjacent locations on an as-needed basis. Data sharing is thus achieved by dynamically allocating resources to cells. The result is a self-organizing array of cells that either interpret the instruction stream, or else reorganize themselves to make better use of available resources.

7. **Design for average load.** Communication is a critical resource in any massively parallel machine. In most cases, a rewrite can be abandoned if it will be tried again later, so communication overload can be handled by some communication requests failing, and then deactivating their originators. This allows

communication resources to be designed for average load, rather than for the worst case.

8. **Multi-grain execution.** The RRM extends the potential efficiency of fine-grain execution to computations that are only locally homogeneous, by dynamically partitioning data into homogeneous domains, each of which is efficiently processed inside one ensemble.

6 Performance Estimates

In order to determine whether we are wasting our time on this ambitious and unusual project, it is important to run some benchmark programs and compare their performance with standard von Neumann processors. It is not easy to compare machines with radically different architectures and models of computation. An ideal comparison with the von Neumann paradigm would use equal clock speed, equal silicon floorspace, equal design effort, equal compiler technology, and equal power in supporting hardware (e.g., caching and memory). We chose a SUN-3/60 for comparison, just because it was readily available. Its clock rate is 20MHz, and we can approximate its 68020 floorspace simply by not considering simulations that involve excessively large graphs, e.g., over a thousand cells. It is clear that this does not give equality in total silicon area, in design effort, or in compiler technology. We have put perhaps 2 or 3 man-years into the RRM project, as compared with thousands for the von Neumann tradition represented by the SUN-3/60, and we have really very little idea of what techniques could be used to optimize RRM performance, whereas many such techniques are already used in the 68020 and its associated hardware and software. Furthermore, a Sun-3/60 has many auxiliary chips for coprocessors, onboard memory, etc. Thus, our estimates are conservative.

Our benchmark programs computed the Fibonacci numbers with Peano arithmetic and with machine arithmetic (called Peano and arithmetic fibo below), and bubblesorted lists of numbers. Handcoded programs were run on an RRM simulator which yields quite accurate timing results, using a 12-by-12 grid of tiles of 8 cells. We tried to achieve approximate parity of compiler power by comparing our RRM Fibonacci code with compiled Common Lisp (KCL) code for the Sun-3/60, and our bubblesort program with a good sorting program in the well established low level language C; this is unfair to the RRM because we are still in an early stage of ensemble controller code technology. (However, the handcoded Fibonacci code is very similar to that produced by our OBJ to RRM compiler.) We then approximated the timing data by simple functions, as shown in the following table, where n represents the argument for a Fibonacci function, or the length of a list for bubble sort:

program	time	sizes
RRM Peano fibo	$.732 \times 1.53^n$	0–10
Sun Peano fibo	2918.41×1.748^n	0–19
RRM arith fibo	$5.1 \times n$	0–14
Sun arith fibo	4.69×1.638^n	0–20
RRM bubblesort	$4.7 \times n$	0–10,20,...,130
Sun bubblesort	$2.497 \times n^2$	0–10,20,...,170

Times are in micro-seconds. The ensemble has a maximum capacity for representing terms, so for each problem there is a largest instance that can be run. Furthermore, problem instances that are close to the maximum size usually exhibit degraded performance because communication costs increase tremendously. Our data indicates that the RRM Fibonacci programs have exponential speedup, while the RRM bubblesort is linear and the SUN bubblesort quadratic, giving a linear speedup.

The speedup factor for the largest common case of the Peano Fibonacci computation run on the RRM simulator is 60, while for the arithmetic Fibonacci it is about 34. For sequences of length 120 in bubblesort example we see a speedup of about 64. This supports a conservative claim that a single RRM ensemble is roughly 50 times faster than a SUN-3/60, which is equivalent to about 150 MIPS, assuming that a SUN-3/60 is rated at 3 MIPS.

Of course, the RRM project is still at a relatively early stage, and we have not explored an especially wide range of problems, nor have we tried especially hard to get the best programs, or to test them over a wide range of data. On the other hand, the results seem relatively consistent and our simulator is quite accurate for this kind of timing data, so there is no reason to suppose that more work will have any effect other than to increase our performance estimates at this level. Given these conditions, it seems fair to say that the RRM ensemble's observed factor of roughly 50 over the SUN-3/60 is quite impressive. Moreover, we believe that this performance will scale approximately linearly for larger problems in which parallelism is actually available, and that, especially for inhomogeneous problems, this is much better than can be expected from more conventional current generation parallel machines. Of course, this extrapolation is on less solid ground than the above estimates, because our simulations have not yet been extended to the cluster level.

References

[1] Arvind, Risbiyur Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 337–369. Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[2] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, 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. Parameterised programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60. IEEE Computer Society Press, March 1987.

[4] J.R.W. Glauert, K. Hammond, J.R. Kennaway, G.A. Papadopoulos, and M.R. Sleep. DACTL: Some introductory papers. Technical Report SYS-C88-08, School of Information Systems, University of East Anglia, 1988.

[5] 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.

[6] Joseph Goguen, Claude Kirchner, José Meseguer, and Timothy Winkler. OBJ as a language for concurrent programming. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 195–198. International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.

[7] 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.

[8] 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.

[9] Joseph Goguen and José Meseguer. Unifying functional, 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.

[10] Joseph Goguen and José Meseguer. Software for the rewrite rule machine. In *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637. Institute for New Generation Computer Technology (ICOT), 1988.

[11] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988.

[12] Peter G. Harrison and Michael Reeve. The parallel graph reduction machine, ALICE. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 181–202. Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[13] Robert Keller and Joseph Fasel, editors. *Proceedings, Graph Reduction Workshop.* Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[14] Robert Keller, Jon Slater, and Kevin Likes. Overview of Rediflow II development. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, pages 203–214. Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[15] Sany Leinwand and Joseph Goguen. Architectural options for the rewrite rule machine. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 63–70. International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.

[16] 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.

[17] Timothy Winkler. Numerical computation on the RRM. Technical report, SRI International, Computer Science Lab, November 1988. Technical Note SRI-CSL-TN88-3.

[18] Timothy Winkler, Sany Leinwand, and Joseph Goguen. Simulation of concurrent term rewriting. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 199–208. International Supercomputing Institute, Inc. (St. Petersburg FL), 1987.