Efficient Integration of Declarative Paradigms into Symbolic Computation Systems

Georgios Grivas[1]

Department of Computer Science
ETH Zürich
grivas@inf.ethz.ch

Abstract. This paper describes the efficient integration of the functional, logic and constraint paradigms into symbolic computation systems. Moreover, it proposes the constraint logic paradigm for the programming language of symbolic computation systems. First, it describes the integration of a separate constraint logic inference engine with the functional language of the symbolic computation system *AlgBench*. The procedural, functional, and APL-like programming styles are inherited from Mathematica and are integrated on top of its term-rewriting mechanism. The proposed language could serve as a stand alone programming language. Second, it shows the efficient compilation of the resulting language to an intermediate code. The compiler generates code for a uniform abstract machine (extended WAM) which is achieved in an object-oriented way.

1 Introduction

The new generation of symbolic computation systems, such as Mathematica™ [21] or AXIOM™ [13] provides beyond the usual mathematical and calculational capabilities a full programming language. The user of such systems has the possibility to write programs that might call the conventional symbolic commands, which appear as built-in compiled (and hence quite efficient) functions. The experimental symbolic computation core system *AlgBench* [14] provides a functional language with the syntax and operational model (*term-rewriting*) of Mathematica. It is designed and implemented in an object-oriented way (in C++).

From the programmer's point of view, *AlgBench* has inherited from Mathematica a multi-paradigm language. Functional, procedural and APL-like programming are integrated on the top of the rewrite rule paradigm. Given a problem, the user can take one style related to its requirements or combine them in a seamless interface.

The need for more logic and proving capabilities in systems doing mathematics (especially in Mathematica) is highlighted in [3]. Besides pattern matching, we have also introduced in [7] unification as a primitive operation in *AlgBench*'s term rewriting environment.

Another paradigm whose need for mathematical programs has been shown early in [17] is that of constraint-based programming. Although, compared to other constraint languages, the existing symbolic computation systems contain sophisticated constraint solvers, they do not make use of

[1]Research supported by the Swiss National Science Foundation

them as a means of programming[2]. The recently introduced language RISC-CLP(Real) [10] can be viewed as a constraint logic user interface for accessing some constraint solvers of the symbolic computation system SAC-2 [5]. While the language is intended for scientists and engineers, it does not allow the full access to SAC-2.

Constraint logic languages [12, 11], replace their primitive operation (unification) by the more general operation of *constraint solving*. From a theoretical point of view, unification is a kind of constraint solving. Constraint logic programs are usually more efficient than logic programs. In addition, programs augmented with constraints achieve often *narrowing* [6] behavior such as the computation of inverse of functions, for example. For these reasons, we have done without the general narrowing mechanism and have introduced two simple inference engines: a rewriting engine and a constraint logic engine.

In this paper we deal only with the programming language part of *AlgBench*. Our main contributions are twofold: First, the enrichment of *AlgBench*'s language with constraint logic features which in our knowledge are not supported either from commercial nor from experimental research symbolic computation systems. Second, the compilation of term-rewriting and of the constraint logic part of the system. Almost all symbolic computation systems are based on imperative languages or on LISP. Most of them are symbolic interpreters. The computer algebra system AXIOM provides a compiler, but it is not based on term-rewriting. Compared to other functional-logic languages [9], our language possesses more sophisticated pattern and algebraic manipulation capabilities but it does not yet provide full support of constraint functional logic programming.

The paper is organized as follows: In the next section we briefly give an informal overview of our language with some examples emphasizing the new logic, functional-logic and constraint capabilities of the system. In Section 3 we explain the need of a compiler for a declarative language of symbolic computation systems. Section 4 deals with the compilation of the two inference mechanisms. It describes our abstract machine (extended WAM) and shows how a uniform implementation is achieved in an object-oriented way. Subsection 4.4 is devoted to the compilation of arithmetic constraints. Section 5 concludes the paper and gives our future aims and directions.

2 Integrating declarative paradigms in *AlgBench*

In contrast to other modern symbolic computation languages such as AXIOM or functional languages such as Miranda™ [19] or Standard ML [15], *AlgBench* provides a more sophisticated algebraic pattern matcher. It offers type-constrained matching, conditional rewrite rules, named patterns, associative, commutative and identity matching, etc. Higher order functions and pure functions are supported along with term-rewriting to give full functional programming support. The user is free from some restrictions of various narrowing languages [9], functional languages [15, 19] or equational languages [16] such as left linearity, strong sequentiality or non-ambiguity restrictions and has the possibility to choose between eager or lazy evaluation.

As we have already mentioned in the introduction, *AlgBench*'s language inherited from Mathematica the functional, procedural and APL-like programming styles. The combination of the integrated constraint logic paradigm with these styles into a coherent system is not trivial, since we do not support narrowing. The operational semantics of the functional part of the language is term-rewriting and of the logic part is SLD resolution. The two parts are at the moment only loosely cou-

[2] with the exception that the conditions of Mathematica's rewrite rules could be seen as a simple form of constraints

pled with non-declarative constructs. In order to provide a full integration we plan to adopt in *AlgBench* the residuation control strategy (as e.g. in LIFE [6]).

A “bridge” between the two inference engines is the Prolog non-declarative operator **is**/2. Any functional symbolic expression may occur on its right hand side. That means we can make use of the arbitrary precision numbers and the algebraic manipulation capabilities of the system in its logic part. On the other hand this is a way for a function call within a predicate.

Function calls in place of goals in predicates are in principle allowed if the user create them within a **Cond**/1 construct. Actually, this is the same procedure as the one that tests if a condition of a rewrite rule is satisfied. If this expression is neither a relation nor a boolean function, it fails. Assume, for example, the conditional rewrite rules **abs**, the Fibonacci function **fib0**/2 as a logic rule and the logic rules **fib1**/1 and **fib2**/1 which contain function calls of **abs**:

```
abs[x_] :=  x  /;  x >= 0
abs[x_] := -x  /;  x < 0

fib0[0, 0] :-.
fib0[1, 1] :-.
fib0[n_, r_] :-
   n1_ is n_-1, n2_ is n_-2,
   fib0[n1_, r1_], fib0[n2_, r2_],
   r_ is r1_ + r2_.

fib1[y_, r_] :-
   n_ is abs[y_],
   fib0[n_, r_].

fib2[y_, r_] :-
   Cond[abs[y_] == y_],
   fib0[n_, r_].
```

A capitalized variable of Prolog, e.g. X, is here of the form x_ or X_. To the logic queries **Query[fib1[16, r_]]** and **Query[fib2[16, r_]]**, we get the binding list **{r, 987}**. Of course **fib0**, **fib1** and **fib2** is not an efficient way to program the Fibonacci function. Here we just want to demonstrate the two kinds of a function call in the predicates **fib1** and **fib2**, respectively.

Logic queries within functions are allowed as well. In case of success the binding list (i.e. a list of pairs of bounded logic variables) is passed and can be further manipulated. In case of failure the symbol **Null** is passed which is generally used to indicate the absence of an expression or a result. Now consider the logic rules **del**/3 (deleting an item from a list) and **perm**/2 (generating permutations of a given list) and the functional rule **power_list**/1 (applying the built-in function **Power** to the permutations of a list **l**), whereas braces denote lists:

```
del[{hx_|tx_}, hx_, tx_] :- .
del[{hx_|tx_}, y_, {hx_|tz_}] :-
   Cond[hx_ =!= y_], del[tx_, y_, tz_].

perm[{}, {}]:-.
perm[{hx_|tx_}, y_] :-
      perm[tx_, z_],
      del[y_, hx_, z_].

power_list[l_List] :=
      Apply[Power, Query[perm[l_, x_], All]]]
```

The argument of the `power_list` function requires type-constraint matching. The type-constraint patterns are of the form *_h* or *x_h* where *h* is a function symbol and *x* is the variable name of the matched argument. In the functional query `power_list[{a,b}]` the type constrained matching with the list object `{a,b}` succeeds and we get `{`x^x`, {`a^b`, `b^a`}}`. The built-in function `Power` is applied by the function `power_list` to the binding list `{{{x, {a, b}}}, {{x, {b, a}}}}`, which is returned from the logic query `Query[perm[l_, x_], All]`.

2.1 Named Patterns

The logic part of *AlgBench* adopts from its functional part the powerful and general *named patterns*, which are not provided in logic languages. A named pattern has the form *x*:*pattern* and assigns the pattern *pattern* the name x. The logic or functional variables *x*_ are a special case of such a named pattern *x*`:_` where the pattern is a *don't care* _[3]. For example assume the fact `p[x:_^_, x_] :-.` We name the pattern `_^_` as *x* and we can refer to it as a whole. To the query `Query[p[a,a]]` we get `fail` but to the query `Query[p[a^b, r_]]` we get the binding list `{r_, a^b}`. Note that `a` and `b` are neither constants nor variables. They are symbols and are treated separately (see Section 4.1.1).

2.2 Integration of linear equation constraints

Constraint logic languages are for combinatorial problems both more expressive and more efficient than other programming paradigms. Hence, providing this paradigm for a symbolic computation system is of great significance.

If ground constraint terms are encountered, the functionality is the same as in the mentioned `Cond` construct. In case of constraints with unbound variables the solver appends the constraint in the *constraint store* (i.e. the memory of the solver which contains the unresolved constraints) and checks their consistency. If the constraint store contains an unsatisfiable set of constraints, backtracking is performed.

Assume the following constraint logic program of the factorial function:

```
fac[0,1] :-.
fac[n_, r_] :-
        n1_ + 1 =:= n_,
        r1 n_ =:= r_,
        fac[n1_, r1_].
```

In *AlgBench* we may use a space instead of an asterisk (*) to denote multiplication. To the query `Query[fac[5, x_]]` we get the binding list `{x, 120}` although the variables r1_ and r_ are unbound during the execution of the condition `r1_ n_ =:= r_`. If we exchange the last two goals of the above program, we take a more efficient version. In this way the previous query does not force to solve a set of equations because every constraint has only one unbound variable. Additionally, if we set the query `Query[fac[x_, 120]]` we get the binding list `{x, 5}`. That means we have narrowing behavior.

[3]Actually *x*_ is an abbreviation of *x*`:_`.

2.3 Non-linear constraints

Non-linear constraints are solved once they become linear during their execution because of their variable bindings. On that ground we evaluate the non-linear part and then we check if the expression turns into a linear form. The expression before the evaluation may contain any functional subexpressions and therefore it may be reduced to an arbitrary expression. For example the constraint `f[x_] + g[y_] =:= 10` could be evaluated to `3 x_ + 5 y_ =:= 10` if the functions `f` and `g` are defined appropriately. Constraint logic languages as might be expected do not support this kind of evaluation.

Note also that the *AlgBench* language provides much algebraic manipulation and numerical computing capabilities which are not in the scope of this paper.

3 The Need for Compilation

One part of our approach is to compile *AlgBench*'s functional language and consequently its term-rewriting mechanism. In this section we discuss the reasons for this compilations. But first, we explain why *AlgBench*'s operational model is term-rewriting.

Mathematica's way of thinking with pattern matching and a database of rules arise from its designers' goal: a system must fit naturally the way a scientist works [21]. Looking up tables of mathematical functions in handbooks is an important (and tedious) aspect of this work. A rule-based system can implement the implicit pattern matching that goes on during this activity. A symbolic computation system should provide a database of the necessary handbooks of mathematical functions and a fast search procedure for a formula in the database. If this system uses term-rewriting, it additionally imitates the method a scientist uses and at the same time offers a natural and easy way for the specification of mathematical formulas.

On the other hand, there are also relatively efficient symbolic computation systems that do not support this declarative kind of programming (e.g. Maple™ [4]). This gives the indication that a procedural language is more efficient. The main reason for this is the relatively bad efficiency of pattern-matching algorithms used in existing declarative symbolic computation interpreters, since the pattern matcher is at the core of a rule-based system. Programming based on pattern matching has one fundamental characteristic compared to imperative programming: the programmer writes his programs in a higher level of abstraction, where the large `case` statements are replaced by elegant declarative rule-based programs. It is a general belief that programs written in languages strongly supporting these styles (e.g. Prolog or most modern pattern-based functional languages) are relatively slow compared to procedural languages (e.g. C). For this reason we claim that the effort to increase the efficiency of these declarative styles to the advantage of higher program abstraction is very important.

The interpreted implementation of *AlgBench* attempts each rule separately not using the common information contained in the patterns. Every time a pattern is matched against an expression the pattern is reread and reinterpreted. The problem with interpreting the pattern directly is that there is no place to store the extra information required to perform code sharing or other improvements. Compiling the pattern-matching operation and the rewrite rules into an intermediate language that is itself interpreted or compiled to native machine code makes this much easier.

4 Compiling *AlgBench* programs

As in any implementation of term-rewriting and SLD-resolution, efficiency is in *AlgBench* a crucial issue. The first efficient implementations of term-rewriting systems presented a compiler that translates the rewrite rules into a high level language (Lisp, Prolog, or Pascal). The generated code was then compiled with the corresponding compiler of the high-level language and the system did not turn out to be faster than modern compilers of functional languages. Some efficient rewriting implementations have been done in the compilation of narrowing languages such as ALF [9]. They compile the rewrite rules into an extension of *"Warren's Abstract Machine"* (WAM) which is the well-known and the de facto technology to compile logic languages [20, 1]. However, the most narrowing languages need a complicated compile-time analysis.

The *AlgBench* abstract machine is an extension of the WAM as well. A major reason for our choice is that the WAM contains all necessary elements that support the term-rewriting and SLD-resolution models and can be easily extended. In this way we get a uniform machine for both mechanisms. In WAM implementations, rules with the same topmost function symbol and the same arity are transformed in one procedure where the common parts of the code are merged, so that the match for the same arguments is performed only once. Each of the remaining terms is then examined until the first successful match is found, or all rules have been tried[4]. Another reason for our choice for compilation to an intermediate code is the consequence to retain *AlgBench*'s portability (*AlgBench* runs on Sun SPARCstations, Cray Y-MP, Convex C220, Alliant FX/80 and Sequent S81b, among others).

4.1 Overall WAM extensions and modifications

In this section, we give the main extensions and simplifications compared to the WAM as described in [1]. Note that we use the WAM terminology of [1] throughout this paper.

The abstract machine is implemented as a class. Although this is related with some inefficiencies, it is for functional logic programs of significant worth. Various instances of this class are active in case, for example, of a logic program that evaluates a functional expression which in turn evaluates a predicate. Moreover, a subclass of the WAM class gives comprehensive capabilities for debugging the programs generated by the compiler.

Like WAM, our abstract machine consists of a memory and a number of registers and instructions. There are many ways to realize the memory of the abstract machine. The most efficient way would be a linear array. The disadvantages of this method are the fixed-size array and the inefficient interaction with the rest of *AlgBench* system. Another way is to realize the memory as a linked list. The problem of this method is the inefficient direct memory accesses. The memory of our abstract machine is realized as a *demand paging* [18] system. Pages are fixed-size blocks of contiguous memory. Every access on a page is managed through a page table. A new page is allocated only when a memory access on a non existent page is attempted. In this way, the memory grows dynamically up to a maximum dependent on the page table size and no pages are allocated after the initialization of the abstract machine.

Unlike [1], where the compiled code is stored in memory and the X-registers are treated separately from the heap, we store the code separately in a double-linked list and we use the memory to store the X-registers. This notion is meaningful because the compiled programs have a longer life-

[4] For a pattern with associative and/or commutative function symbols this could also result in considerable savings.

time than the memory of the abstract machine which is deallocated after a solution search. In addition, it enables an efficient delete of instructions during *peephole optimization*.

All instructions that have as parameters both temporary and permanent variables, are implemented separately for each variable type. In this way, we do not need to check the type in the instruction implementation and the additional type information in the arguments of the instructions is not necessary.

4.1.1 List representation

AlgBench's specific expressions such as lists or symbols are stored in a constant cell (CON). Hence, a constant cell contains a pointer to a functional expression. A list in functional part of *AlgBench* is represented as an array of pointers to their elements while in WAM and in the most functional languages it is represented as *head-tail* (*car-cdr*) pairs. This requires two conversions: from WAM format to functional *AlgBench* format and vice versa. The conversions are necessary because a unique list representation would restrict either the efficiency of the functional engine or that of the logic engine.

4.1.2 Complete indexing

A great optimization in the execution time of any rule-based language can be achieved by speeding up the rule search. To reduce the set of rules that have to be tried, indexing is applied. This can be realized by looking at the type or the value of one or more arguments. The conventional WAM-based compilers employ *first argument indexing* [1] where the rules are grouped only upon the topmost functional symbol of the first argument. Instead, in *complete indexing* all arguments and all nested subterms of each argument are indexed as well.

The complete indexing method implemented in our system is based on the work done in [8]. This scheme allows an independent analysis of argument positions and of the nested subterms. It views the rules as sets and it assigns to each formal parameter the rule set in which it appears. A new register **`Alt`** is introduced to hold the intermediate sets of a stepwise reduction of the set of applicable rules and a new constituent *alt* in the choice point frame.

4.2 The Conditional Rewriting Engine

In the compilation of term-rewriting some WAM instructions are simplified and others are not needed at all. In particular, *dereferencing*, *permanent variables*, *environment frames*, and the **`allocate`**/**`deallocate`** instructions are not needed. Instead of *deep* rules, only *chain* rules are essential and instead of **`call`** just **`execute`** is necessary. The functional queries do not contain any variables and variable binding is accomplished only in one way. For that reason we have integrated in our abstract machine the *match* instructions **`match_variable`**, **`match_value`**, **`match_local_value`**, **`match_constant`** and **`match_void`** and the *match-get* instructions **`mget_value`**, **`mget_structure`**, **`mget_list`** and **`mget_constant`** which are identical to the corresponding WAM's *unify* and *get* instructions respectively in **`read`** mode.

Since a functional query does not contain any variables, a new simplified (compared to [8]) indexing instruction is required:

switch_on_term X_i, *(clab, calt)*, *(flab, falt)*: jumps to the instruction labeled *clab* or *flab* depending on whether the dereferenced value of the X_i register is a constant or a structure, respectively. Moreover, it intersects the current value in register **Alt** with *calt* or *falt* and assigns the intersected value to **Alt** register.

In the following we briefly discuss our solutions to particular problems of the compilation of term-rewriting.

4.2.1 Variable binding and substitution

Variable binding and substitution can be carried out during compilation. The compiler replaces the functional variables through objects of a class **match_var**. The register **index** (a field of this class) gives the corresponding WAM memory address of this variable. The instruction **eval** (a method of the above class) depending on the value of the **index** register performs an implicit substitution by returning the current binding of the variable built on the heap.

4.2.2 Rewrite rules with variable arity

Another important difference between the SLD resolution and the term-rewriting mechanism is that the logic rules with the same topmost function symbol and different arity are treated as different procedures. The WAM merges therefore only rules with the same topmost function symbol and the same arity to a procedure. Since the functional rules have another semantics, we need consequently an instruction which merges the functional rules with the same topmost function symbol independent of their arity to one procedure. The instruction **switch_on_arity** selects the right procedure depending on the arity of the functional query:

switch_on_arity X_i, $\{(lab_i,\ alt_i,\ n_i) \mid 1 \leq i \leq m\}$: expects a topmost function symbol **f**/*n* referenced by X_i. If there is a tuple $(lab_i,\ alt_i,\ n)$, alt_i is assigned to **Alt** register and then it jumps to lab_i. Otherwise X_i contains already the result.

4.2.3 Conditional rewrite rules

Conditional rewrite rules are a powerful technique in programming with functions. In order to take into account conditions on rewrite rules, a preprocessing is needed at compilation time. The satisfaction of these conditions is ensured by the indexing instructions *retry* and a new modified (compared to [8]) instruction *try*:

try #arg, $\{r_i,\ 1 \leq i \leq m\}$: removes the first element *i* from **Alt** and calls the condition code of rule r_i as long as all conditions of the rule are satisfied or **Alt** is empty. In the former case it returns the right hand side of the rule and in the latter case the result is **fail**.

4.3 The code generator

We support two intermediate levels of translation before the WAM code generation. First, the scanner and parser of the *AlgBench* interpreter transforms the rules into a form suitable for the interpreter. Second, the parser of the compiler takes the programs in this form and transforms them into an intermediate language suitable to achieve an efficient register allocation and code generation to the extended WAM.

The code generation is accomplished in five steps:

1. On the top level, indexing instructions are created and the code generation methods of the class *rule* are called.
2. A method of the class *rule* is called which generates code for a rule. First, it performs the register allocation of the terms.
3. After the register allocation, two methods of the class *term* are called which generate code for the left hand side and the right hand side of the rules, respectively. After that, all registers of the rules are deallocated.
4. The methods of the class *register* generate the WAM instructions with register arguments. In the code generation methods, it is determined by virtue of state variables of the registers if the code to be generated is for permanent or temporary variables and for safe or unsafe values, respectively.
5. On the low level of code generation the methods of the class *code* are used. A method of this class appends the new generated instructions to the existing WAM code. Other methods of this class set labels in code or perform a label jump.

4.4 Compilation of arithmetic Constraints

Our compiler recognizes constraint expressions and generates code for the above discussed abstract machine which is extended with new instructions suitable for the constraint definitions. The unification algorithm is replaced by a constraint solver that checks the satisfiability of a set of constraints and attempts to solve them.

The constraint expressions are first transformed into an intermediate form. The variables occurring in the constraints are replaced by the variables of the class `match_var` (see Section 4.2.1). Furthermore, the constraint expressions are split into a linear, a non-linear and a constant part. These parts are then stored in a list `c`.

New instructions are introduced for the variable and register allocation and for code generation. The generated code consists only of the instruction `add_constraint c`, where `c` is the above mentioned list.

In our object-oriented compiler the constraints are implemented as a new class of terms. The implementation of the constraint solver uses similar techniques as the implementation of the extended WAM (e.g. it is also realized as a class). As in the most implementations of constraint logic languages [12, 11] the constraint solver consists of a memory, a register set and an instruction set. We introduce a new slot in the cell definition of our abstract machine <*SOV*, *cid*> to hold a solver variable. The *cid* field is a unique key to identify a solver variable.

The incremental[5] constraint solver for linear equations implemented in *AlgBench* is based on the *Gaussian elimination* method. A set of equations is represented as $Ax + b = 0$ i.e. it is stored as a coefficient matrix A and a constant vector b.

The solver memory consists of five stacks:

- *choice-point stack*: to store a solver state.
- *log-stack*: book-keeping of any change of the matrix A, needed for the backtracking of the solver.
- *addr-stack*: to hold the addresses of the solver variables <*SOV*, *cid*> in the WAM memory. A solver address is obtained from **`addr-stack`**[*cid*].
- *col-stack*: to store a permutation vector which is used to address the columns of the matrix A
- *row-stack*: to store the pointers to the rows of the matrix A.

4.4.1 A dynamical choice-point strategy

A solver state is stored in a *choice-point stack*. For that reason we extend the choice point frame with a pointer to the top of the choice-point stack. As in WAM, the instructions **`try`**, **`retry`** and **`trust`** allocate and reset a new choice point frame.

The generation of a solver choice point is necessary only when the solver state is changed and not in every WAM choice point allocation. The compiler cannot decide when a choice point allocation is absolutely necessary. Therefore, at the time of the WAM choice point creation, we notify in the choice point frame together with the solver stack pointer that we might call the **`try`** or **`retry`** instruction in case of a solver state change. Unlike this delay of the **`try`** and **`retry`** instructions, the **`trust`** instruction is executed immediately, as the corresponding WAM **`try`** instruction.

In this way, for programs without constraints, no solver choice points are allocated. Thus the execution time of the logic programs without constraints is not increased because of the solver choice points.

5 Conclusions and Future Work

We presented the design of a new language for symbolic computation systems which integrates the functional, logic and constraint paradigms into the experimental symbolic computation system *AlgBench* and provides a new powerful kind of programming for symbolic computation systems. In addition, it offers adequate methods for the solution of combinatorial search problems that are important for users of symbolic computation systems (scientists and engineers). Functional-logic and constraint-logic features are to our knowledge not supported either in commercial nor in experimental research symbolic computation systems.

We realized a compiler for our language which generates code for an extended WAM in an object-oriented way and achieves a speedup from 1.7 to 5.8 over the interpreter, depending on the number of rules involved in a procedure definition. Most of the symbolic computation systems are symbolic interpreters based on imperative languages or on LISP. The system AXIOM provides a compiler, but it is not based on term-rewriting or SLD-resolution.

[5]appending a new constraint in the store does not require to solve old constraints.

We are currently working on adopting in *AlgBench* the residuation control strategy (as in [6]) in order to provide full integration of the constraint, functional and logic paradigms and in order to give an adequate operational model for our language. Furthermore we plan to incorporate a solver for inequality constraints as in the most constraint logic languages.

In our work on compiling *AlgBench* we have not yet taken into account neither segment patterns nor patterns with associative and commutative function symbols. However, using similar techniques it should be possible to build a compiler for the classes of such patterns as well. This will be subject of future work.

Acknowledgments. We would like to thank R. Mäder and J. Christian for many useful suggestions. We also thank M. Levy and C.T. Visser for the errata of [1]. Many thanks to G. Dolge who realized the logic part of the mentioned implementations.

References

1. H. Aït-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction*. MIT Press, 1991
2. H. Aït-Kaci. Towards a Meaning of LIFE. In *Proc of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, Springer-Verlag, LNCS 528, Passau, Germany, August 1991.
3. B. Buchberger. *Mathematica*: A System for doing Mathematics by Computer? Invited Talk. In *Proc. of the 3rd Intern. Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'93),* Springer Verlag, LNCS 722, Gmunden, Austria, September 1993.
4. B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *MAPLE: Reference Manual*. University of Waterloo, 5th edition, 1988.
5. G. E. Collins and R. Loos. *The SAC-2 Computer Algebra System*. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria.
6. M. Fay. First order unification in equational theories. In *Proc. of the 4th Intern. Conference on Automated Deduction (CADE*-4), Academic Press, Austin, Texas, 1979.
7. G. Grivas and R. E. Maeder. Matching and Unification for the Object-Oriented Symbolic Computation System *AlgBench*. In *Proc. of the 3rd Intern. Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'93),* Springer-Verlag, LNCS 722, Gmunden, Austria, September 1993.
8. W. Hans. A Complete Indexing Scheme for WAM-based Abstract Machines. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92)*, Springer-Verlag, LNCS 631, Leuven, Belgium, August 1992.
9. M. Hanus. The Integration of Functions into Logic Programming: A Survey. To appear in *Journal of Logic Programming*.
10. H. Hong. RISC-CLP(Real): Logic Programming with Non-linear Constraints over real numbers. In F. Benhamou and A. Colmeraurer, editors, *Constraint Logic Programming: Selected Research*, MIT Press, 1993.
11. J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. The CLP($\Re$) Language and the System. *ACM Transactions on Programming Languages*, 14(3), 339-395, 1992.
12. J. Jaffar. Constraint Logic Programming: A Survey. To appear in *Journal of Logic Programming*.
13. R. Jenks and R. Sutor. *AXIOM, The Scientific Computation System*, Springer-Verlag, 1992.

14. R. E. Maeder. *AlgBench*: An Object-Oriented Symbolic Core System. In *Proc. of the 2nd International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'92),* Springer Verlag, LNCS 721, Bath, England, April 1992.

15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

16. M. O'Donnell. *Equational Logic as a Programming Language*. Foundations of Computing, MIT Press, 1985.

17. Alexander Schriver. *Theory of Linear and Integer Programming*. John Wiley, 1980.

18. A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

19. D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. of Functional Programming Languages and Computer Architecture FPCA'85*, September 1985.

20. D. H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309, SRI International, AI Center, Computer Science and Technology Division, October 1983.

21. S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, second edition, 1991.