The compilation of a set-based logic language for generic parallel machines

Giancarlo Succi*
Joy Marino* Alberto Baudino* and Antonio Durante*

+Laboratorio Ingegneria Informatica
Studi Aziendali Università di Trento
email: charmi@ili.unicn.it
Via Zeni 8 I-38068 Rovereto

*Università di Genova
Dipartimento di Informatica Sistemistica e Telematica
email: Joy.Marino, morgoth@dist.dist.unige.it
Via Opera Pia 11A, 16145 Genova, Italy

Keywords: Set, Compiler, Declarative Language, Abstract Machine

The distinctive idea underlying declarative programming is to let programmer work at a high level of abstraction, expressing what his/her program must do rather than how it shall do it. This is the reason why declarative languages offer many advantages compared with the imperative ones:

- various kinds of static analysis are intrinsically simpler;
- the program is easier to understand and its correctness, when possible, can be more simply verified;
- the programs are more concise and readable, which makes declarative languages suited to fast prototyping;
- parallelism is implicit, and the compiler can recognize it without great difficulties.

Furthermore, because of the high level of abstraction of declarative languages, in principle a logic program can be ported on any kind of architecture, since it is up to the compiler transform it to an executable that can run fast and reliable on the desired machine. Actually, the differences between the computational paradigms of declarative languages and the ones peculiar of the existing architecture models make impossible an “isomorphic” compilation: attempts to carry out machines dedicated to the execution of declarative languages had not a great success; at present, declarative languages execute mostly on standard architecture models, being they sequential or parallel, using an executor, usually known as Abstract Machine, modeling the needed structure: this is, for example, the standard implementation technique for Prolog, which uses the WAM (Warren Abstract Machine) as an intermediate mean between the compiler and the real machine.

This paper aims to present an approach to compiler construction for declarative languages, using a set based logic language as target; the choice of set as the fundamental data structure of the language is motivated, first of all, from the intention to exploit parallelism inherent in it, as regards both process and data parallelism; besides, it is general convinckement that sets are a very powerful programming tool, in particular suited for fast prototyping, where it is useful for the programmer to have at his own disposal high level data structures and operation abstractions. Finally, set theory can be used to represent problems in various fields of application.

Unlike other languages, such as Prolog itself, that embodies sets just as an addition to the basis data structures, our target language, called SL (Set Language), is designed around sets, handling them in a clear and simple way and offering the basic operations on them as a part of the language. SL, developed at DIST, Università di Genova, can be considered a superset of SEL (Subset Equational Language), a logic-functional language developed in 1987 by Jayaraman and Plaisted. SEL joined to equational programming, whose paradigm was already consolidated, the idea to introduce subset assertions, devoted to procedures that have sets as result. Sets, handled in such way, show their particular tendency to parallel computation, especially as regards data parallelism. SL adds to programming paradigms of SEL the standard notations of set theory, allowing the user to write easily programs dealing with sets.

SL is based on equational assertions, using three kinds of constructs to deal with sets:

1. The operator union is used in assertions like:

\[ f(X) = g(X) \cup h(X). \]

where \( f, g \) and \( h \) return a set as result.

2. Assertions like the following ones are used to build a set whose elements are a sequence of integer, in case with a given step:

\[ f(X, Y) = \{ X \cdot Y \}. \]
\[ g(X, Y) = \{ X, m(X, Y) \cdot h(Y) \}. \]

In the latter, \( X \) is the first element of the sequence, while \( m(X, Y) \) is the second: the step is therefore given by \( m(X, Y) - X \).

3. The third construct of SL allows to build a set with the typical set notation, as in:

\[ f(S, Z) = \{ (X, Y) : X \in S, Y \in Z \}. \]
\[ f(S) = \{ h(X) : X \in S; X < 3 \}. \]

The implementation of SL is divided in two phases:
- the development of a compiler targeted to an abstract machine,
- the implementation of an abstract machine on the real architecture.

The compilation and execution of SL programs are analogous to that of Prolog: adopting an abstract machine allows to get free from the constraints of the real architecture on which the program will be executed. Portability of SL programs is therefore due to the implementation of the abstract machine: the compiler transforms the source code into a sequence of assembly instructions of the abstract machine, independently from the way the instruction set is implemented. The compiler can thus devote its attention to efficiency and reliability of the executable, giving prominence to both process and data parallelism aspects inherent in SL. An implementation of the abstract machine on a SIMD architecture will better exploit data parallelism, while on a MIMD architecture efficiency will be obtained exploiting process parallelism; on a sequential architecture no kind of parallelism is possible: anyway this is completely transparent to the compiler, that carries out code that can be executed as much efficiently as possible on every kind of architecture.

The SL abstract machine is called SAM (Set Abstract Machine) and its assembly language is obviously called SAL (SAM Assembly Language). The SAM belongs to the WAM family [AK90], since its general structure resembles quite a lot that of the WAM; however it does not need full unification capabilities, therefore there is no need of the "trail". The SAM can be viewed as a sister of the SEL-WAM [A.N88], from which it inherits most of the implementation strategies, which are extended with some new optimization techniques, table of constants and the capability of handling functors.

The process of compilation is articulated into various phases: each of them aims to pass from a higher to a lower level code, solving step by step problems that rise while passing from the declarative paradigms of SL to the imperative ones of SAM. SL is designed for set theory and it therefore lacks explicit control and parallelism, as it is evident in the following SL assertions:

\[
\text{makeUnion}(\text{Set1}, \text{Set2}) = h(\text{Set1}) \cup g(\text{Set1}, \text{Set2}).
\]
\[
\text{prime}(\text{Numbers}) = \{ X : X \in \text{Numbers}; \ \text{empty}(\text{divisors}(X, \text{Numbers}\{X\})) \}.
\]
\[
\text{divisors}(X, S) = \{ Y : Y \in S; Y < 1, (X/Y)\ast Y = X \}.
\]
The first phase of compilation consists in translating SL to SEL: a precompiler transforms SL constructs into sequences of equational or subset SEL assertions. The SEL code for the above-mentioned SL assertions is:

```plaintext
makeUnion(X,_) contains h(X).
makeUnion(X,Y) contains g(X,Y).
prime([X|T]) contains if (empty(divisors(X,T)))
  then [X]
  else {}.
divisors(X,[Y|T]) contains if ((Y<>1) && ((X/Y)*Y==X))
  then [Y]
  else {}.
```

The first SL assertion, where \( h/2 \) and \( g/3 \) have a set as result, is translated into two subset SEL assertions, that can be executed in parallel on a MIMD architecture: the parallelism is therefore more explicit in SEL notation.

Second example shows translation of an assertion that allows data parallel execution. Here is present a remarkable feature of SL, viz. the multiple matching: since set elements are not ordered, a matching of the kind \( X \) in \( \text{Numbers} \) produces the matching of \( X \) with all the elements of the argument set; therefore, the resulting set contains all the elements of the given set that satisfy the condition \( \text{empty(divisors(X,T))} \).

SEL code produced from the precompiler is then processed by the compiling module, whose task is to yield a SAL code, implicitly optimized in order to be efficiently and reliably executed on different kinds of architecture. The focal point of the compilation stands in the translation of assertions dealing with sets; there are two kinds of operation that an assertion can perform on a set:

- mapping one set into another;
- searching for particular elements of a set.

The execution of machine instructions devoted to these operations is onerous in terms of machine resources, most of all on a sequential architecture: hence it is important to optimize the use of such instructions. Moreover, identifying a set element imposes to verify the correctness of the search and to build a code that permits to retract a wrong choice, when the failure of a pattern matching depends on the previous matching of another pattern. Steps that allow to perform such a task form the SAL Code Implicit Optimization Algorithm.

Implicit SAL Code Optimization Algorithm aims to produce a sequence of SAL instructions that ensure correctness and completeness of the result, ordering all the patterns to be matched in a set, in such way to minimize the possibility of failure in matching and to reduce the search space when retracting a wrong choice. The algorithm searches for particular relations between patterns appearing in the left side of an assertion: these relations are synthesized by two graphs: the former is based on matching precedence relations, that take origin from unification properties, verifying if the MGU (Most General Unifier) set of each pattern is contained into the MGU set of another. The latter synthesizes the interference relations that bind two patterns, even declared in different sets, by means of an unbound variable, so that the correctness of the search for an element that matches one of the two patterns depends also from the success of the other pattern matching.

An example can help to understand:

```plaintext
all_grandpa({father(Grandpa,Dad),father(Dad,_)|_}) contains {Grandpa}.
get_grandpa({father(Grandpa,Dad),father(Dad,_)|_}) = Grandpa.
```

In the first assertion the search for an element satisfying one of the two patterns produces a result only if an element that matches the second pattern exists: so \( \text{father('Tom','George')} \) is a correct match for \( \text{father(Grandpa,Dad)} \) only if the set contains an element \( \text{father('George',_)} \).

The difference between the two assertions is that the first iterates all over the given set, giving as result the set of all the Grandpas, while the second extracts just one element that matches the two pattern. In both assertion, however, the patterns are interfering by means of the unbound variable \( \text{Dad} \). The searches are associated to nested structures; taking the first assertion as an example, the nested structure can be modeled by the sequence:

---

1 The arity of a SL predicate is given by the number of arguments + 1 for the result.
This structure is assimilable to that of a cycle, but it can be executed in parallel on a SIMD machine: for example, on a CM2 each element of the set is assigned to a processor to verify the pattern matching.

The algorithm uses informations synthesized by the graphs to single out groups of pattern that must be matched, so that a matching can be retracted if a successive verification fails. Each group can be analyzed separately, thus limiting the number of nested searches. The groups of patterns are called "interference classes" and are singled out analyzing the connective properties of a general interference graph, which is the union of the two above-mentioned graphs. Before selecting a pattern which originates a searching or mapping cycle the algorithm eliminates all matchable patterns, so to limit further on the number of operations that are performed in each cycle.

The SAL code produced by the compiler is the input of an optimizing module, which applies on it the following optimization strategies:

- ORA (Optimized Register Allocation); it consists in trying to use the lowest number of registers, avoiding unnecessary data movements, and to re-use registers whenever it is possible;
- RIE (Redundant Instructions Elimination); it sometimes happens that one or more instructions are correct but redundant and can thus be eliminated;
- ET (Environment Trimming); it consists in ordering the permanent variables on the environment of the assertion as well as to reflect the ordering of their last occurrence in the right side of an assertion: namely, the variable whose last occurrence is the last of all will be the first on the environment, and so on. This allows to deallocate parts of the environment as they are no longer used, saving dynamic space allocation.
- LCO (Last Call Optimization); it is based on the fact that permanent variables allocated to an assertion should no longer be needed after all the arguments for the last assertion call in the right side have been prepared with the put instructions: then, the environment can be deallocated right before the last assertion call.

The translation from SL to optimized executable SAL is performed by a Prolog written compiler running on a SUN workstation. SL programs, so translated, successfully execute on CM2 and SUN/Sparc.

References


