

The compilation of SL, a set-based logic language for generic parallel architectures

Giancarlo Succi¹ and Carl Uhrík²

¹*DISA, Università di Trento, Via Inama 5, I-38100 Trento, Italia*

²*Department of Technology Programs Faculty, University of Phoenix, Englewood, CO, USA*

Set-based languages have emerged as a powerful means for expressing not only programs but also requirements, test cases and so on. However, a uniform compilation schema for sets has not yet been completely developed. The present paper tries to overcome this lack using a set-based logic language, SL (set language), as target.

The approach is based on an imperative abstract machine, the SAM (set abstract machine). The translation from SL to SAL (SAM assembly language) is described and all the possible optimizations, both at source code level and at assembly code level, are detailed. The potentials for identifying parallel flows of computations are analysed. Several examples of compilations are presented and discussed.

Keywords: ? keywords?

Sets have been emerging as a powerful means for the whole software development process, from requirements collection to testing. This amounts to saying that they can be viewed as a kind of uniform expressing paradigm throughout the whole software production life cycle, thus leading to more straightforward connection between the different phases (van Vliet 1993). Furthermore, most programmers already have considerable experience with sets, from their mathematical background, and they can capitalize on such experience using a set-based programming language.

Sets show a natural suitability for parallel implementations, in particular data parallel ones. For example, the search for an element satisfying certain conditions can be executed in one shot: first storing each set element in a processor and requiring the processors to verify the matching of their elements with the desired one, then collecting the results. Similarly, any quantification/iteration operation over set members is parallelizable. A useful property specific to subset equational language (SEL) is that by requiring all variables in equational and subset assertions to be universally quantified over terms rather than expressions, depth-first exploration of subset assertions with innermost reduction of expressions and breadth-first exploration are equivalent for the operational semantics (Jayaraman 1992). Thus, SEL provides some flexibility for considering process parallelism and data parallelism to be equally applicable.

One of the many advantages of logic languages over imperative languages is that parallelism is implicit and a compiler can recognize it without great difficulty. Given the availability of an increasing number of hardware platforms supporting parallelism, it makes sense to look for ways to exploit this innate parallelism of sets and logic languages by seeking to implement them on parallel architectures.

From 1 July 1997 Carlo Succi's address will be Department of Electrical and Computer Engineering, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1W4 (Giancarlo.Succhi@enel.calgary.ca; Giancarlo.Succhi@acm.org).

0963–9306 © 1997 Chapman & Hall

SL (set language), a language developed at the Computer Science Departments at the University of Trento and the University of Genova, is closely akin to SEL. It represents a fairly expressive set-based logic language, yet makes use of only a simplified associative–commutative matching mechanism (rather than a full-blown set unification mechanism). It was selected as the subject of the initial **parallel** implementations described in this paper. Later it is hoped to apply insight gained from these to implement a subsequent language incorporating set unification and set terms in relational assertions such as in Log or SURE.

In section 1, the history of sets as a core programming structure is outlined, and in section 2 the SL language is described from the view of syntax and execution. In sections 3 and 4 a general overview of the compiler strategy and the phases of the compilation are presented, showing how SL is translated to SEL by a precompilation which is in turn translated to assembly code for SAM (a set-oriented abstract machine). In section 5, details of using SAM instructions and features to implement the SEL code are covered. Section 6 briefly describes patterns for the actual compilation, and in section 7 we consider interference patterns in the set matching algorithm which is the centre point of the compiler. Some platform independent optimizations are performed in section 8, and section 9 shows a sample compilation. Finally, section 10 briefly introduces a user environment, and section 11 summarizes and offers concluding remarks.

1 History of sets

Sets were first introduced in Pascal by Wirth and then replicated in Modula 2 (Wirth 1985). However, in such languages their use is highly constrained.

In the late 1970s a group of researchers at New York University designed a set-based language, SETL (Schonberg et al. 1979, Schonberg et al. 1981, Freudenberger et al. 1983). SETL allowed the programmer to define a set type, adopting an automatic strategy to single out the best representation. It demonstrated how expressive sets are and how many applications can be considered in this way. Other declarative languages, such as Prolog (Sterling and Shapiro 1986), added set handling to their capabilities, but with rather unsatisfactory results. In Prolog, for instance, sets are represented as lists, imposing an arbitrary order on their elements, so they are managed as ordinary terms, and the built-in predicates that support them are not very efficient.

Several other alternatives to adding set constructs to logic programming have been proposed. A practical motive for these extensions to the declarative paradigm came from a desire to marry the work on extending relational database theory into more general structures with the work on using logic programming as a database language (Kupers 1988). Kupers (1987) suggested **quantification over a set** as a key concept, which had the particular appeal of cutting complexity by providing a logical, non-procedural equivalent to iteration over set elements. Another alternative is represented by LDL (logic data language) (Beeri et al. 1987), a Horn clause programming language intended for data intensive knowledge-based applications with complex terms, which emphasizes the use of **set construction**. A third alternative is simply to introduce a number of new set-theoretic primitive predicates in the language. Kupers proved that, given auxiliary predicates and syntactic restrictions to guarantee unique minimal models for programs, such languages are essentially equivalent.

In 1987, Jayaraman and Plaisted developed a logic-equational language called SEL (subset equational language) (Jayaraman and Plaisted 1987, Jayaraman and Plaisted 1988). SEL augments the well established equational paradigm with the idea of using subset assertions devoted to functions that have sets as

result. Sets handled in this way show a particular capacity to express parallel computations implicitly, particularly using data parallelism. Since SEL employs only simple matching over set terms, a natural subsequent development is SURE (subsets, relations and equations) (Jayaraman 1990, Jayaraman 1991) to incorporate extensions to permit full **set unification** operations in order to support set terms within relations. Similarly, Log (Dovier et al. 1991), an extended logic language based on definite Horn clauses with no extralogical constructs, adds special **set terms** to allow the **extensional** representation of finite sets. It possesses distinguished predicates representing set membership and equality and incorporates a unification algorithm (proven to terminate) that can cope with set terms.

2 SL: a logic language based on sets

SL is a set-based programming language recently developed at the University of Trento and the University of Genova. It provides the opportunity for a programmer to write programs dealing with sets easily, using the usual notations of set theory. Though SL may at first seem simplistic, it is intended as a language that permits the writing of efficient and concise programs with flexibility for applications in diverse fields. Thus, it is hoped that compact declarative programs may aid program understandability along the lines discussed in Cimitile (1994). Furthermore, SL programs should run on many different architectures, sequential or parallel, with little or no adaptation to the source language. SL is closely related to SEL (Jayaraman and Nair 1988), but whereas SEL involves subset assertions and equational assertions, SL is based solely on equational assertions but adds additional constructs (described below) to make it closer to the widely familiar mathematical set notation.

2.1 Matching

As stated above, SL is composed of equational assertions that have the form:

$$f(\text{terms}) = \text{general_expression}$$

In the top-level (command level) environment, assuming a program has been defined, SL allows queries:

$$? - \text{expr}$$

where the meaning of the result returned is a ground instance of *expr* (i.e. one with all variables bound to constant terms) which is a logical consequence of the program's equational assertions and a closed-world assumption. The assertions effectively cause replacing equals with equals, left to right, in a way directly analogous to term rewriting. Nested function application is performed innermost first, ultimately targeting arguments that will result in ground terms. If no path of chained assertions leads to an assignment of ground terms to match $f(\text{terms})$, then the closed world assumption assigns $f(\text{terms}) = \{\}$ for a set-based function. Similarly for any element-based function, the closed world assumption assigns $f(\text{terms}) = ?$, which propagates consequences upward via the rule $h(\dots, ?, \dots) = ?$ for any element-based function or constructor h , and in the set-valued case, via $h(\dots, ?, \dots) = \{\}$ and $\{? | s\} = s$. Furthermore, SL follows SEL in using only restricted associative-commutative matching. In this way, SL only requires one-way matching, rather than unification. SL intrinsically represents sets using the set constructor $|$ and the empty set atom $\{\}$. The construct $\{h | t\}$ refers to a non-empty set and is equivalent to $\{h\} \cup t$. In SL sets, as in mathematical sets, order is not significant, nor are any notationally redundant members. Thus,

$$\{1 | \{2 | \{3 | \{4 | \{\}\}\}\}\} \equiv \{1, 2, 3, 4\} \equiv \{1, 1, 2, 3, 4, 4\} \equiv \{4, 2, 3, 1\} \equiv \{1\} \cup \{2\} \cup \{3\} \cup \{4\} \cup \{\}.$$

Since equality in the SL matching scheme is based only on the associative and commutative properties of set unions, but not the idempotent property, matching of $\{h \mid t\}$ with $\{a, b, c\}$ could give one of three bindings:

$$h \leftarrow a, \quad t \leftarrow \{b, c\}, \quad (h \leftarrow b, t \leftarrow \{a, c\}), (h \leftarrow a, t \leftarrow \{a, b\}),$$

but not a binding such as

$$(h \leftarrow a, t \leftarrow \{a, b, c\}).$$

This restriction aids in treating recursive assertions. Another consequence of the matching scheme when multiple matches are provided as just mentioned occurs for assertions of the form

$$\text{equational_assertion} ::= \text{function}(\text{terms}) = \text{expr}$$

or

$$\text{equational_assertion} ::= \text{function}(\text{terms}) = \text{expr} \cup \text{expr}.$$

In these cases, only one of the potentially many matches is considered in reducing the assertion, and because the program is declarative, it is assumed that the result will be independent of which match is used. Thus, the user should be aware that confluence has been assumed for these equational assertions. For example,

$$\text{cardinality}(\{\text{member} \mid \text{tail}\}) = 1 + \text{cardinality}(\text{tail})$$

is safe, since any assignment to member and tail is equally valid in counting, but

$$\text{minus}(\{\text{member} \mid \text{tail}\}) = \text{member} - \text{minus}(\text{tail})$$

is not, since the assignment of member and tail drastically affects the result.

Similarly to Prolog and SEL, SL assigns special significance to the anonymous variable ‘_’ in matching. It generates a unique variable with unknown or irrelevant name to serve as a wild card place holder to complete a pattern, with each occurrence unconnected to any other ‘_’ in an assertion. Thus, for example, $\{_ \mid _ \}$ matches any non-empty set.

2.2 Set grouping using relative sets

SL allows building a set with the typical mathematical set notation, which specifies set terms using variables arising from generators coupled to conditions, as in:

$$\begin{aligned} \text{cartesian_product}(S, Z) &= \{(X, Y) : X \text{ in } S, Y \text{ in } Z\}. \\ f(S) &= \{h(X) : X \text{ in } S; X < 3\}. \\ \text{flatten}(S) &= \{X : X \text{ in } Z, Z \text{ in } S\}. \end{aligned}$$

The first line above produces a set containing all of the Cartesian product pairs of two input sets, whereas the second line returns a set which has members whose values result from calling the function h with arguments being elements of the given set S which respect the condition that they be less than 3. flatten takes a set of sets as argument. The result is the set of all the elements of the inside sets.

2.3 Range sets

Assertions such as the following are used to build a set whose elements are a sequence of integers, in some cases with a given step:

$$\begin{aligned} f(X, Y) &= \{X, \dots, Y\}. \\ g(X, Y, Z) &= \{X, Z, \dots, Y\}. \end{aligned}$$

In the first situation, the set corresponding to $\{X, X + 1, X + 2, \dots, Y - 1, Y\}$ is constructed. In the latter situation, Z is the second element of the sequence: the step is therefore given by $Z - X$; thus, the set $\{X, Z, X + 2 * (X - Z), X + 3 * (X - Z) \dots, Y\}$ is constructed.

2.4 Pattern matching

Pattern matching can be applied on set elements, as in the following assertion:

$$aFatherLieutenant(\{father(X, -) \mid -\}, \{lieutenant(X) \mid -\}) = X.$$

This assertion identifies a person occurring in a father construct within the first argument set and in a lieutenant construct within the second argument set. It is assumed that only one such X satisfies this condition; otherwise there is lack of confluence.

2.5 The *is* operator

To facilitate pattern matching, the *is* operator has been added. The following example helps to explain its role:

$$\begin{aligned} allFathersLieutenants(Fathers, Lieutenants) \\ = X : F \text{ in } Fathers, L \text{ in } Lieutenants; F \text{ is } father(X,), L \text{ is } lieutenant(X). \end{aligned}$$

2.6 Taking a union

The operator *union* is used to construct new sets in assertions such as

$$f(X) = g(X) \text{ union } h(X).$$

where f , g and h return sets as results.

2.7 A simple database example

In this section we present an example of a database application. It is a simple program to extract from a database of employees the ones whose job level is higher than class 3 and who earn less than \$4000 a month, and then look up in a database of degrees to determine those who have university undergraduate degrees. Let us assume that the first database is organized in a set of triples (*person, income, level*) and that the second is a set of pairs (*person, {kind_of_degree1(location1), kind_of_degree2(location2), ... }*).

A sample of the first database might be:

$$FirmDB = \{('Cooper', 3800, 5), ('Lee', 10000, 4), ('Morgan', 2000, 2)\}$$

and a sample of the second could be:

$$\text{DegreeDB} = \{ ('Cooper', \{ \text{high_school}('St. Barbara'), \text{bachelor_degree}('Princeton') \}), ('Lee', \{ \text{high_school}('Boston'), \text{bachelor_degree}('Boston') \}), ('Paul', \{ \text{high_school}('Genoa'), \text{bachelor_degree}('Harvard'), \text{doctor_degree}('Harvard') \}) \}.$$

The assertion *underpaid* is defined in the following way:

$$\text{underpaid}(\text{Firm}, \text{Degrees}) = \{ \text{Name} : (\text{Name}, \text{Income}, \text{Level}) \text{ in Firm}, \text{Name}, \{ \text{bachelor_degree}(_) \mid _ \} \text{ in Degrees}; \text{Income} < 4000, \text{Level} > 3 \}.$$

A query can be expressed as:

$$? - \text{underpaid}(\text{FirmDB}, \text{DegreeDB}).$$

leading to the resulting set $\{ 'Cooper' \}$.

3 Overview of the compilation process

Because declarative languages do not directly specify how execution has to be performed, they permit even more flexibility in writing source code portable to many kinds of architectures than imperative languages. The programmer entrusts a compiler with the task of producing an efficient and reliable executable code (assembly language) that can run on a desired machine taking full advantage of its architecture. However, since usually the target assembly language is an imperative language, an ‘isomorphic’ compilation is impossible. Moreover, attempts to build machines suited to the execution of logic languages have not had great success.

Thus, a natural approach is to execute declarative languages on a standardized virtual architecture, using an executor, known as an abstract machine, modelling the structures needed for a conceptually efficient implementation. This is, for example, the standard implementation technique for the G-Machine (Burn et al. 1988, Johnsson 1987), for Tim (Wray and Fairbairn 1989), for GRIP (Peyton Jones 1987, Peyton Jones 1988), and for the WAM (Warren 1983, Ait-Kaci 1990).

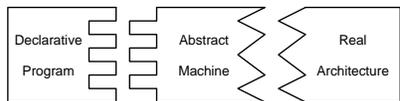


Figure 1: The role of the abstract machine

Figure 1 gives a pictorial representation of such a structure. The abstract machine acts quite like an adapter which takes as input the structure of a declarative program and adapts it to the real architecture, whatever it may be. Adopting an abstract machine temporarily frees one from the limits of a real architecture on which the program will be executed. The code produced by the compiler is executable on

Table 1: Overview of the compilation phases

Phase	Purpose	Input	Output
Precompilation	Translating source code to SEL intermediate code	SL code	SEL code
Compilation	Translating SEL code to SAM assembler	SEL code	SAL code
Optimization	Optimizing SAM assembler SAL code	Optimized SAL code	
Codification	Translating optimized SAL code to SAM numeric code	Optimized SAL code	SAM numeric code

different implementations of the abstract machine, and in each one, the abstract assembly instructions are implemented in a potentially different way to exploit the best features of the real architecture.

The methodology used here employs an abstract machine called the set abstract machine (SAM), inspired by the SEL-WAM (Jayaraman 1992), an earlier abstract machine for set-based logic languages based on the WAM. Translation of SL source code to numerically coded SAM instructions consist of four phases, each one aimed at passing from a higher level of abstraction to a lower one, as summarized in Table 1:

- Precompilation of SL code to SEL intermediate code, as defined by Jayaraman (1992) and augmented by Succi (1993). The aim of this phase is twofold:
 - to obtain a homogeneous treatment of sets
 - to identify those portions of code executable in process parallelism and those in data parallelism.
- Compilation of SEL code to SAL (SAM Assembly Language – the assembly language of the SAM abstract machine); the aim of this phase is to make the control flow of the program explicit in terms of abstract machine instructions.
- SAL code optimization applies several abstract analysers, namely for optimized register allocation (ORA), redundant instructions elimination (RIE), environment trimming (ET) and last call optimization (LCO) in order to produce more efficient code by cutting out operations that are not strictly needed.
- Numerical codification: SAL optimized code is simply turned into SAM machine code as compact binary images.

Each phase solves problems that arise in passing from the declarative paradigm of the SL language to the imperative one of the abstract machine. Subsequently, as a final process, the SAM numerical code for the abstract machine is implemented on a real architecture. This has currently been done for three cases: CM2 (SIMD), Transputer (MIMD) and SUN4 workstation (sequential).

In the next section, the precompiler will be briefly described. Then, in section 5, the reader will be introduced to SAM, presenting its main features, before discussing the details of compilation and optimization in sections 6 and 7.

4 Precompilation

As mentioned earlier, precompilation is the first step in compiling an SL program. The input of this phase is SL code and the output is subset equational language (SEL) intermediate code. The aims are:

- to explicitly expand sets that are defined by the programmer
- to express set constructs in a format that is still declarative but reduces them to a simplified homogeneous form, in particular for analysing pattern matching and isolating independent subclauses in assertions
- to identify, still in a declarative format, the presence of data and of process parallelism.

4.1 A brief overview of SEL

Before describing the translation of SL to SEL, it is worth first introducing the basics of SEL. SEL differs from SL mainly in that it does not include the general set expression form $\{term: generators; conditions\}$ nor the use of ‘. .’ notation within sets.

SEL bases its programming paradigm on two kinds of assertions.

Equational assertions Equational assertions are of the kind

$$f(arguments) = expression,$$

such as

$$parents('Bill') = \{father('Fred'), mother('Wilma')\}.$$

$$is_empty(\{\}) = true.$$

$$is_empty(_) = false.$$

Subset assertions Subset assertions have the structure

$$f(arguments) \supseteq expression,$$

as in

$$intersection(\{X | _ \}, \{X | _ \}) contains \{X\}.$$

$$mix(\{X, Y | _ \}) contains \{X + Y\}.$$

$$mix(\{X, Y | _ \}) contains \{X - Y\}.$$

Variables appearing in right-hand sides must also be contained in left-hand sides. When both an equational assertion and a subset assertion match a function call, the equational assertion takes precedence.

Declarative meaning of equational assertions is that for each ground instance of the *arguments* declared in the left-hand side of the assertion, the *f* function being defined returns one and only one ground term defined by the right-hand side of the assertion. As in SL, when confluence does not hold for equational

assertions involving sets as arguments, the result may not be unequivocally and deterministically singled out.

A subset assertion, on the other hand, groups as result all the ground instances of the right-hand side of the assertion related to all the matching configurations satisfying the *arguments*.

These behaviours can be illustrated by contrasting the answers given to the query $? - inc(\{10, 2, 17\})$ given the two assertions as separate programs.

Program 1: $inc(\{X | _ \}) = \{X + 2\}$.

Program 2: $inc(\{X | _ \}) \text{ contains } \{X + 2\}$.

The answers given to these assertions are quite different. Program 2 returns unequivocally the set $\{12, 4, 19\}$ by virtue of the fact that *contains* causes the alternate bindings for X to be taken across unions of the right-hand side expression. However, Program 1 is flawed since it suggests returning one of three sets $\{12\}, \{4\}, \{19\}$ according to unspecified actions left up to a given implementation of SAM. Had the assertion been $inc(\{X | _ \}) = \{X + 2\}$ it would be perfectly acceptable, since the match for the head of a list is unique.

4.2 From SL to SEL

The major translations that are performed are:

- resolution of the explicit set assertions
- substitution of the set of sequences
- rewriting of the *union* clauses
- elimination of all the syntactic sugars defined in SL, such as macro definitions, constant definitions and file inclusion.

Constructs peculiar to SL and their SEL translation are now presented.

Sequences In SL it is possible to build a set whose elements are a sequence of integers, possibly with a given step. Two significant examples are:

$$f(X, Y) = \{X \dots Y\}.$$

$$g(X, Y) = \{X, m(X, Y) \dots h(Y)\}.$$

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

In both the above cases, the SEL translation contains the predefined *intSequence_/3* clause:

$$f(X, Y) = \text{intSequence_}(X, 1, Y).$$

$$g(X, Y) = \text{intSequence_}(X, m(X, Y) - X, h(Y)).$$

(By convention clauses are identified by *name/arity*, where *arity* is the number of arguments. In actual code, the arity is 1 greater because an additional argument accounts for the returned result value.)

The definition of *intSequence*₃ in SEL itself is given by:

$$\begin{aligned} \text{intSequence}_{3}(First, Step, Last) = & \text{if}(First \leq Last) \\ & \text{then}\{First \mid \text{intSequence}_{3}(First + Step, Step, Last)\} \\ & \text{else}\{\}. \end{aligned}$$

Unions The SL *union* operator is used in assertions such as:

$$f(X) = g(X) \text{ union } h(X).$$

where *f*, *g* and *h* return sets as results. The SEL code for this assertion is of the kind:

$$\begin{aligned} f(X) \text{ contains } & g(X). \\ f(X) \text{ contains } & h(X). \end{aligned}$$

SEL incorporates a **collect-all assumption**, which states that the result of a function application to ground terms is the union of all the subsets obtained by all the subset assertions matching the ground terms over all of the possible matchings. Thus, the above two SEL assertions are equivalent to the preceding SL one. While SL is more compact and closer to set theory notation, SEL shows more explicitly the possibility for parallelism inherent in such assertions. In fact on a MIMD architecture, in principle, the above SL assertion can be executed in parallel by two processes, one executing the first SEL assertion and the other executing the second.

Set assertions A third construct in SL allows building a set with the usual set notation, as in the assertions:

$$\begin{aligned} f(S, Z) = & \{(X, Y) : X \text{ in } S, Y \text{ in } Z\}. \\ p(S) = & \{h(X) : X \text{ in } S; q(X)\}. \end{aligned}$$

This is easily translated into SEL code as follows:

$$\begin{aligned} f(\{X \mid _ \}, \{Y \mid _ \}) \text{ contains } & \{(X, Y)\}. \\ p(\{X \mid _ \}) \text{ contains if } q(X) \text{ then } & \{h(X)\} \text{ else}\{\}. \end{aligned}$$

is operator The translation of the *is* operator has already been described.

5 SAM, the set abstract machine

In this and the following sections, a detailed description of the subset of instructions in SAM which are identical in the WAM instruction set is omitted in order to concentrate more on those instructions dealing explicitly with sets. A more detailed description of the entire SAM instruction set can be found in Succi and Marino (1991) and Succi (1993).

Programming the abstract machine requires using an imperative language that makes execution control and parallelism explicit. SEL makes explicit only the **potential parallelism**, leaving execution flow control still implicit. The process of translation from declarative SEL language to an imperative one such

as SAL bridges this gap. Subsequent to this translation, some architecture independent optimizations can be applied before producing execution control code which can be further optimized under the implementation on specific architectures. After a brief overview of SAM data structures, the various instructions devoted to matching and managing data and process parallelism will be covered, first considering pattern mapping and pattern searching in depth and afterwards process control. Subsequently, in sections 6 and 7, the compilation and optimization algorithms will be described.

5.1 Overview of the architecture of SAM

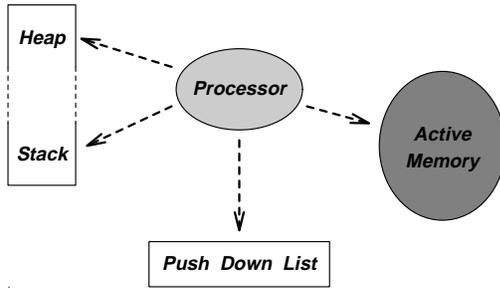


Figure 2: Structure of SAM.

Figure 2 outlines the structure of SAM. Since its structure is quite similar to that of the WAM, we merely mention the main differences between the WAM and SAM, referring to Ait-Kaci (1990) and Succi (1993) for further details.

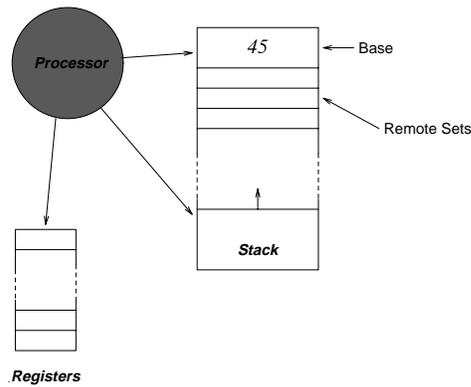


Figure 3: Structure of the active memory.

The heap stores all data except for sets, which are stored in the active memory (AM), a particular structure devoted not just to holding sets but also to executing data parallel operations on them: in fact, its cells both store data and perform computations. Actually, this duality is fully realized only on data parallel implementations, such as on the CM2, where each AM cell is composed of a storage element

and a processing element, as shown in Figure 3. On other implementations, the AM is implemented in standard memory structures and operations on set elements are performed sequentially.

The stack stores control flow choice points and the environment of the assertion. An assertion environment is allocated and placed on the stack whenever there is a function call in the right side of an assertion that can overwrite registers. Since backtracking is not allowed in SEL, the use of choice points is different from that in the WAM. SAM choice points are used to implement the collect-all assumption, i.e. whenever there is more than one match for the same ground terms in a subset assertion, and the result is given by the union of all the subsets resulting from the evaluation of all that assertions. To allow a correct evaluation it is then necessary to save the contents of the argument registers (i.e. the ground terms) and a pointer to the code of the subsequent subset assertion that has to be evaluated.

The push-down list (PDL) is used here to perform pattern matching in assertions involving sets: it stores the patterns that have to be searched for in a set.

5.2 Pattern matching and pattern searching instructions

Pattern matching and pattern searching instructions constitute a class of instructions that allow verifying or seeking pattern sequences satisfying certain conditions. To begin with a simple case, consider the assertion:

$$\text{couple_of_equals}(\{(X, X) \mid _ \}) = X.$$

This matching situation can be addressed by using either of the following simple pattern matching instructions:

```
match_set Za Zp Zb
match_set_rem Za Zp Zc Zb
```

The first of these two instructions searches for an element matching the pattern Zp in the set Za , storing in Zb the Boolean result of the matching. The second does the same but also stores the remainder of the set in Zc , accordingly taking more execution time.

The full SAL code for the assertion $\text{couple_of_equals}/2$ is shown in Figure 4. (The reader is reminded that the arity in the code is $1 + \text{functor_arity}$ to include an argument for the return value. Each SAL instruction is followed by a semicolon and its description; however, sometimes the whole description or some of the details of the arguments may be omitted to make reading easier.) After the *get* phase, the *start_set_match* begins the building on the PDL of the pattern (X, X) , storing in $Z3$ the address of the PDL where the pattern is stored. The pattern building is performed by progressively pushing the elements of pattern onto the PDL stack via *store_pdl* instructions: first the tuple constructor $()/2$ signalling two elements, then the address of the pre-allocated X variable address is taken from $Z4$ with an indication that that this variable is to be instantiated, and finally the same address again but with an indicator that the value is to be verified. Then, the *match_set* instruction matches the pattern (stored between $Z3$ and top of stack on the PDL), searching for it within the set $Z1$, reporting in $Z5$ the Boolean result. The instruction *fail* checks the contents of $Z5$, managing a possible failure. If the matching is successful, the *store_ind_value* stores the result $Z4$ into the location addressed by the register $Z2$. Finally, the *proceed_eq* returns the control flow to the caller.

Now consider a more complex case, for which the simple *match_set* instruction will not work:

$$\text{get_grandpa}(\{\text{father}(\text{Grandpa}, \text{Dad}), \text{father}(\text{Dad}, _) \mid _ \}) = \text{Grandpa}.$$

```

couple_of_equals/2:
  get_set Z1 A1      ; get the set argument in Z1
  get_variable Z2 A2 ; get result var address in Z2
  start_set_match Z3 ; PDL pointer in Z3, start of pattern
  store_pdl_functor ()/2; (? , ?) pattern on PDL stack
  store_pdl_variable Z4 ; (X, ?) pattern on stack
  store_pdl_value Z4 ; (X, value(X)) pattern on stack
  match_set Z1 Z3 Z5 ; look for pattern match in set
  fail Z5           ; back up to last choice point if no match
  store_ind_value Z4 Z2; otherwise store result
  proceed_eq       ; and return

```

Figure 4: SAL code for *couple_of_equals/2*.

This situation cannot be compiled into two consecutive instances of the simple pattern matching operation described above (i.e. using variables *Grandpa* and *Dad* respectively), because it cannot be assured **independently** that both the matched patterns will be satisfied simultaneously. For instance, if the query were:

$$? - \text{get_grandpa}(\{\text{father}('Tom', 'Sally'), \text{father}('John', 'Phil')\},$$

the atom *Tom* is not a solution of *get_grandpa/1*, but *father('Tom', 'Sally')* matches the pattern *father(Grandpa, Dad)*. There is therefore a need for a construct that allows withdrawing a previous commitment, so that, after having tried with *Tom* to satisfy the variable *Dad*, the matching can retry with *John*. The WAM implements this backtracking by means of an extra data structure, the **trail**; however, as previously pointed out, since SAM does not require full unification capabilities, these forms of matching can be performed more directly by abstract machine instructions.

Thus, the following instructions are devoted to the more complex situations, when backtracking may be required:

```

search_set Za Zi Zp Zb end
continue_search Zt Zi Zp Zb start end
end_search Zt Zi Zp Zb start
search_set_rem Za Zi Zp Zc Zb
continue_search_rem Zt Za Zi Zp Zc Zb start end
end_search_rem Zt Za Zi Zp Zc Zb start

```

The registers have the same meaning as for *match* instructions, adding the register *Zi* as an index within the set and the register *Zt*, used by *end_search* and *end_search_rem*, and *continue_search* and *continue_search_rem* to check if the preceding matching has been successful. The search for the desired pattern in the set is performed by a cycle delimited by *search_set* and *end_search* instructions. In the sequential implementation of the SAM, this is performed with a loop (but in a parallel implementation, the

```

get_grandpa/2 :
  get_set Z1 A1          ; get the set argument in Z1
  get_variable Z2 A2     ; get result var address Z2
  start_set_match Z3     ; PDL pointer in Z3: 1st pattern
  store_pdl_functor father/2 ; father(?,?) pattern on PDL stack
  store_pdl_variable Z4   ; father(Grandpa,?) pattern
  store_pdl_variable Z5   ; father(Grandpa,Dad) pattern
  start_set_match Z6     ; PDL pointer in Z6: 2nd pattern
  store_pdl_functor father/2 ; father(?,?) pattern on PDL stack
  store_pdl_value Z5     ; father(value(Dad),?)
  store_pdl_dummy        ; father(value(Dad),_)
  search_set Z1 Z7 Z3 Z8 end ; search set for 1st pattern Z1 (copied in Z7)
start:
  match_set Z1 Z6 Z9     ; verify match of 2nd pattern in Z6 – Z9 is the fail flag
  end_search Z9 Z7 Z3 Z8 start; if 2nd match failed, redo 1st
                                ; if all Z1 has been analysed without success then set Z8
end:
  fail Z8                ; if both fail, fail the clause
  store_ind_value Z4 Z2  ; otherwise store result
  proceed_eq             ; and return

```

Figure 5: SAL instructions for *get_grandpa/2*.

loop would be parallelizable). For simplicity, this implementation is used as reference point for the explanation of the SAM instruction set. As in the simple matching case, the loop begins with a *start_set_match* instruction and *store_pdl* instructions to build a pattern on to the PDL. Inside the loop, first an element is picked out of the set, then its congruence with the pattern is tested incrementally; if a failure occurs within a loop cycle, *end_search* at the end of the loop, or *continue_search* inside it, retry the search from the beginning, i.e. backing up to the last relevant point where *search_set* was executed, selecting a new element from the set as long as there is one yet untried. If no suitable element is found, the clause fails.

For example, referring to the *get_grandpa/1* SEL assertion above, the sequence of SAL instructions is as in Figure 5 (Remember that arity in the code is $1 + \text{functor} - \text{arity}$ to include an argument for the return value. Thus, *get_grandpa* appears as *get_grandpa/2* in the code).

After the *get* phase and the placing of the patterns on the PDL, the *search_set* instruction searches in the set Z1 for an element matching the pattern stored on the PDL at the location referred to by Z3, using Z7 as an index for the search (particularly to be used in backtracking). The Boolean result of the search is stored in Z8: if the search is not successful, i.e. there is no set element matching the pattern, the control is passed to the *fail* instruction at the line labelled *end* which manages the failure. If, on the contrary, the search succeeds, the *match_set* instruction selects an element of Z1 matching the pattern stored on the PDL at the location referenced by Z6, storing the Boolean result in Z9. The cycle is ended by *end_search* which checks the contents of Z9, and, if it signals a failure, retries the first search by restarting from the index register Z7 and finding the next element that matches the pattern *father(Grandpa,Dad)*, storing the Boolean result in Z8. If the search is successful control passes to the instruction labelled by *start*, retrying

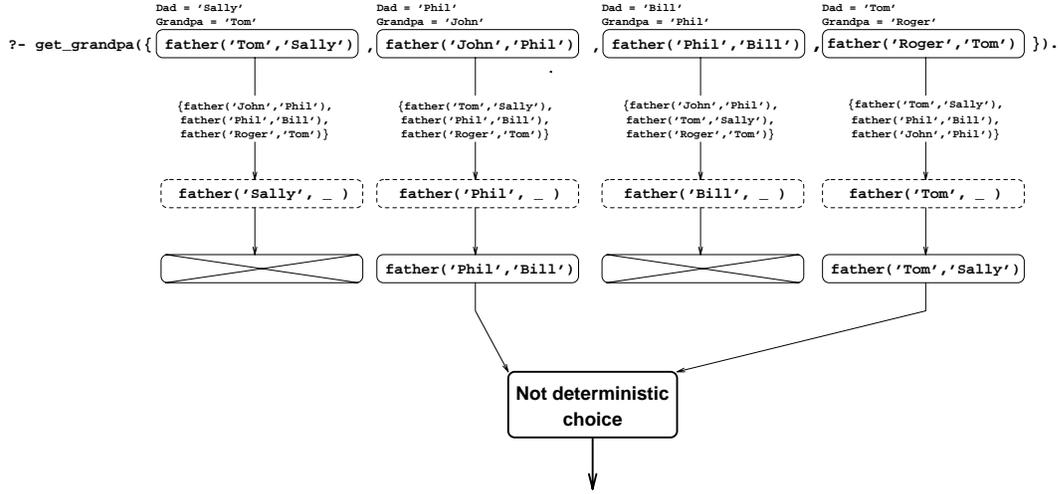


Figure 6: Parallel search execution scheme for assertion `get_grandpa/2`.

the matching of the pattern referred by Z6; otherwise, control passes to the following *fail* instruction which handles failure in the event that all matches fail after having iterated through all elements of the set.

In a SIMD architecture such as the CM2, this iteration can be executed in parallel on each element of the set. The *search_set* instruction can be performed by distributing matching across many CM cells – one for each element in the set potentially satisfying the match. The result of the search operations will be extracted from the elements satisfying internal matches. This operation is depicted in Figure 6. Each line in the figure represent a step in the data parallel execution. In this case all the set elements are processed in parallel, leading to four streams. All of them satisfy the first pattern, generating therefore four ‘remainder sets’. Over these new sets a new pattern matching is performed, again in parallel, aimed at identifying whether the son of the first clause is also a father. Only two among the starting four selections satisfy this new condition. The end of the pattern matching is then reached, and there are two potential candidates for being ‘the result’ of this clause: a non-deterministic choice is therefore applied.

5.3 Pattern mapping instructions

Unlike the pattern matching instructions, which aim to find a result related to only one of the possible element configurations matching a given sequence of patterns, the mapping instructions aim to generate a new set whose elements are a function of the elements of the given set. To be a part of the final set, some conditions can also be imposed on these elements.

These instructions are used to execute assertions such as

$$inc(\{X \mid _ \}, Inc) \text{ contains } \{X + Inc\}.$$

This assertion performs a **constant space mapping**, since it just scans the set. It may be carried out using the following three instructions:

map_over Za Zi Zm end

```

inc/3:
  get_set Z1 A1      ; get the set argument in Z1
  get_variable Z2 A2 ; get the INC argument in Z2
  get_variable Z3 A3 ; get result var address in Z3
  map_over Z1 Z4 Z5  ; map set getting element X in Z5
start: put_value Z5 A1 ; put X in arg1 register
      put_value Z2 A2 ; put INC in arg2 register
      put_variable Z6 A3; set up result to be in Z6
      operation + /3   ; X + INC → Z6
      insert Z6 Z3     ; Z3 union {Z6} → Z3
end_map Z4 Z5 start ; continue map, w/ X elements
end: proceed_eq      ; return

```

Figure 7: Code for implementation of *inc/3* assertion.

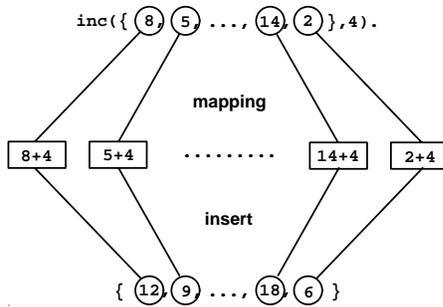


Figure 8: Mapping execution scheme for assertion *inc/3*.

```

continue_map_over Zt Zi Zm start end
end_map Zi Zm start

```

where Za is the set argument, Zi is an index register to be used in iterating through the set elements, Zm stores the current element, and Zt stores the result of a previous matching operation to verify the correctness of the choice. Again, the idea of a cycle is implicit for a sequential algorithm (which is to be parallelized in the parallel architecture) and the functions are analogous to the *search*, *continue*, *end* sequence except that no pattern has to be set in the PDL since every element of the set is to be considered. Thus, an implementation of the above *inc/3* assertion is as shown in Figure 7.

The overall result is stored in register $Z3$ while each processed element is stored in $Z6$. Note the *insert* instruction inserts an element in the result set. Had there been conditions restricting the results to be return, it is easy to see how these would have been placed.

Now, consider the parallel implementation of the same set of map instructions. Figure 8 shows an execution scheme of assertion *inc/3* on a SIMD architecture along lines similar to those mentioned in the last section.

If a mapping assertion requires managing the remainder of a set, one has to perform either a **quadratic** or a **linear space mapping**, according to circumstances. In general quadratic space is required when a number of copies of the set equal to its cardinality must be built; linear space, on the other hand, is required when the remainder is not used in the answer or is only referred to as a whole. Instructions for quadratic space mapping are as follows:

```
map_generating_copy Za Zi Zm Zc end
continue_map_generating Zt Za Zi Zm Zc start end
end_map_generating Za Zi Zm Zc start
```

and the instructions for linear space mapping are:

```
map_overriding_copy Za Zi Zm Zc end
continue_map_overriding Zt Za Zi Zm Zc start end
end_map_overriding Za Zi Zm Zc start
```

where all registers have interpretations identical to the simple map instructions already given, but with the addition that the *Zc* register stores the remainder of the set.

Some assertions require processing only particular elements of the set. For example, in the following,

```
calculate({(X, Y, -) | Rem}) contains {(X, Y, interpolate(X, Y, Rem))}.
fathers_with_two_children({father(F, -), father(F, -) | -} contains {F}.
```

in the assertion *calculate*/1, only elements of the argument set that are triples are to be considered, or in *fathers_with_two_children*/1, only elements that correspond to the terms with *father*/1 at the top level should be put in the result. In these cases, one needs to iterate only on elements that respect a given pattern. For this reason, all of the above **mapping instructions** have counterparts that perform both matching and iteration, as listed below.

```
map_over_matching Za Zi Zp Zc end
continue_map_matching Zt Za Zi Zp Zc start end
end_map_matching Za Zi Zp Zc start

map_generating_matching Za Zi Zp Zc end
continue_map_generating_matching Zt Za Zi Zp Zc start end
end_map_generating_matching Za Zi Zp Zc start

map_overriding_matching Za Zi Zp Zc end
continue_map_overriding_matching Zt Za Zi Zp Zc start end
end_map_overriding_matching Za Zi Zp Zc start
```

All of these instructions have register interpretations consistent with the earlier definitions, with the exception that the index pointed to by *Zi* and the element contained in *Z* satisfies the match to the pattern indexed by *Zp*. A more comprehensive description of all instructions presented in this section can be found in Succi (1993).

5.4 Instructions for process managing

So far, the behaviour of instructions on MIMD architectures has been omitted in the course of discussions on SAM instructions. On MIMD architectures, matching, searching and mapping instruction sequences are executed as on a sequential architecture. The only further difference is that one can single out independent processes that can be executed in parallel. In effect, each code region belonging to a parallelizable process can be marked to be executed on a different processor.

In the SL program:

$$\begin{aligned} \text{get_abs_grt_than}(X, \text{Nums}) &= \text{get_grt_than}(X, \text{Nums} \text{ union } \text{get_grt_than}(X, \text{inv}(\text{Nums}))). \\ \text{get_grt_than}(X, \text{Numbers}) &= \{N : N \text{ in } \text{Numbers}; N > X\}. \\ \text{inv}(\text{Numbers}) &= \{-N : N \text{ in } \text{Numbers}\}. \end{aligned}$$

Parallelizable processes can be singled out in the *get_abs_grt_than/3* assertion, where the two disjoint processes related to the *get_grt_than/3* assertion may be executed simultaneously.

SAL instructions devoted to delimit parallelizable code regions are the following:

```
do_me_thenaddr
last_do
```

To fully understand the meaning of these assertions, it is necessary first to consider the SEL translation:

$$\begin{aligned} \text{get_abs_grt_than}(X, \text{Numbers}) &\text{ contains } \text{get_grt_than}(X, \text{Numbers}). \\ \text{get_abs_grt_than}(X, \text{Numbers}) &\text{ contains } \text{get_grt_than}(X, \text{inv}(\text{Numbers})). \\ \text{get_grt_than}(X, \{N \mid _ \}) &\text{ contains if } (N > X) \text{ then } \{N\}. \\ \text{inv}(\{N \mid _ \}) &\text{ contains } \{-N\}. \end{aligned}$$

In this case it is clear that there are two independent flows of control represented by the two different *get_abs_grt_than/3* assertions. The translation to SAM code marks these accordingly. Thus, SAL imperative code for the *get_abs_grt_than/3* assertion is as shown in Fig 9.

Note that at this point, some additional instructions (*try_sub_and*, *allocate*, *deallocate*) have been added at the beginning and the end of the code. These represent standard conventions for making function calls and will be mentioned again in the next section. The result of the execution of *get_abs_grt_than/2* is the union of the sets resulting from the execution of the two calls to the *get_grt_than/3* code. The *try_sub_and* instruction is used to link the execution of more than one subset assertion with the same name and arity that work together in building the result. However, when two or more subset assertions have not only the same name and arity, but even the same arguments, it is possible to perform a parallel execution of the right side code: the *do_me_then* and *last_do* instructions delimit the code regions that can be executed in parallel. Note that the *Get* phase (including allocation of a stack activation record and arguments set-up), since it is common to both the assertions, is performed once and for all before the *do_me_then* instructions, saving memory space and execution time. Figures 10 and 11 show the alternative execution flows of *get_abs_grt_than/3* on sequential/SIMD and MIMD architectures respectively.

```

get_abs_grt_than/3:
  try_sub_and - 1      ; this is the only clause
  allocate             ; allocate the environment
  get_variable Z1 A1   ; get X in Z1
  get_variable Z2 A2   ; get Number in Z2
  get_variable Z3 A3   ; set Z3 as result var
  do_me_then secondPart ; mark region 1
  put_value Z1 A1     ; set up args, X
  put_value Z2 A2     ; and Number
  put_variable Z4 A3   ; set Z4 as result register
  call get_grt_than/3 ; 1st get_grt_than/3 call
  union Z4 Z3         ; union result into Z3
secondPart:
  last_do             ; mark region 2
  put_value Z2 A1     ; get Number
  put_variable Z5 A2   ; set Z5 as result
  call inv/2          ; inv(Number) → Z5
  put_value Z1 A1     ; set up args, X
  put_value Z5 A2     ; and inv(Number)
  put_variable Z4 A3   ; set Z4 as result register
  call get_grt_than/3 ; 2nd get_grt_than/3 call
  union Z4 Z3         ; union result into Z3
  deallocate          ; deallocate environment
  proceed_sub         ; return

```

Figure 9: SAL imperative code for *get_abs_grt_than/3* assertion.

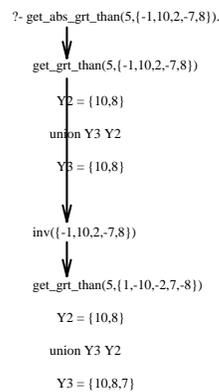


Figure 10: *get_abs_grt_than/3* Execution flow on sequential/SIMD machine.

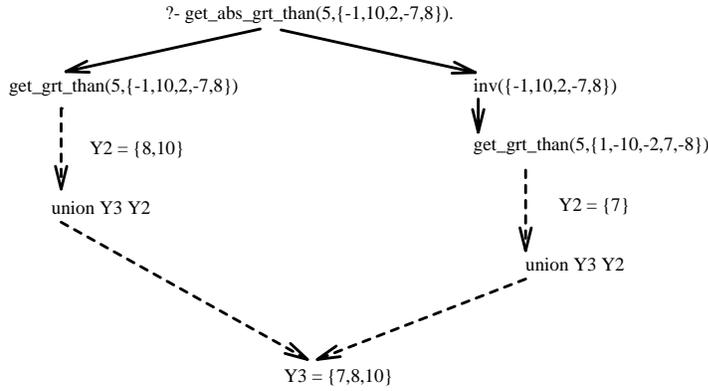


Figure 11: Execution flow of `get_abs_grt_than/3` on a MIMD machine.

6 Primary compilation schemas

This section aims to present a set of schemas for the translation processes in close connection with control flow and parallelism management. First we consider how equational assertions without sets are treated; then sets in equational assertions and subset assertions are covered.

6.1 Compilation of the simple equational assertions without set terms

As previously mentioned, equational assertions are assertions of the form:

$$f(X_1, X_2, \dots, X_n) = \text{expression}.$$

Such assertions without sets are handled in the same way that the WAM handles Prolog clauses with $n + 1$ arguments: the '+1' is due to the fact that in the SAM we have to handle the result also. The general pattern for generating SAM code for such an assertion is shown in Figure 12 (At this general level, almost all concepts are similar to those for the WAM; for a more complete discussion regarding how clauses are compiled, refer to the Kaci tutorial (Ait-Kaci 1990). In particular, sometimes in the code shown in Figure 12 the registers will be indicated by Z_i , without distinction between temporary and permanent registers –see Ait-Kaci (1990) and Succi (1993).)

This compilation scheme is quite similar to that of the WAM, except for the fact that in SEL there is no unification, so there are no *unify* instructions: after the establishing links to other equational assertions of the same arity (via *try_equ_else*) and allocation of an activation record on the stack, the *get* instructions copy the ground terms to which the argument registers A_i refer into the registers Z_i . Then, the *expression* on the right side of the assertion is evaluated and the result is stored in the register Z_{n+1} . Finally, the environment is deallocated from the stack and the control flow returns to the caller. All equational assertions, except for the ones having sets as arguments, follow this scheme.

6.2 Compilation of subset-based assertions

This section addresses the principles underlying the compilation of assertions dealing with sets, drawing upon the prior discussion of SAM instructions for implementing SEL code. We first consider equational assertions dealing with sets before looking to subset assertions.

```

f/n + 1
  try_equ_else - 1      ; link to other f/n + 1 assertions
  allocate              ; allocate an activation record for f
  get_variable Z1 A1    ; store 1st arg from reg A1 in reg Z1
  get_variable Z2 A2    ; the same for the second argument
  ...                  ; similarly for all other arguments
  get_variable Zn An    ; the last argument
  get_variable Zn + 1 An + 1; result

computation of expression and storage of the result in register Zk

  put_value Zk Zn + 1   ; move result to result variable
  deallocate            ; deallocate the activation record
  proceed_eq            ; return control to the caller

```

Figure 12: General pattern for generating SAM code.

6.2.1 Equational assertions dealing with sets

Equational assertions with sets need to perform **pattern matching** and/or **pattern searching**. Pattern matching solves simple situations as in

$$f(\{g(X) \mid _ \}) = X.$$

The function $f/1$ extracts from a given set an element matching the pattern $g(X)$, returning an instantiated X which is the argument of g . This case corresponds to the simple match instructions outlined in section 5.2. As mentioned earlier, sometimes it is not possible to obtain a correct result with a consecutive sequence of simple independent pattern matches, as for example in the assertion

$$\text{may_not_be_ordered}(\{h(1, X), h(Y, 2) \mid _ \}) = X - Y.$$

which identifies two elements in a given set matching the functor $h/1$, one having the first argument equal to 1 and the other having the second equal to 2, returning the difference between these arguments. Suppose one makes the query:

$$? - \text{may_not_be_ordered}(\{h(3, 5), h(1, 5), h(1, 2), g(2, 1)\}).$$

The expected answer is 4, but since sets are intrinsically not ordered and pattern matching is non-deterministic, it is possible that the pattern $h(1, X)$ matches the element $h(1, 2)$, so that the subsequent matching of $h(Y, 2)$ fails and the result is not correct. The system therefore needs to perform the search for set elements exhaustively by allowing the withdrawal of a previous choice if a successive matching fails: this is what is meant by pattern searching.

Hence, the general SAL code scheme for an equational assertion dealing with sets is:

simple matching of elements whose searches can be ordered

```

building on PDL of patterns for the first nested search cycle
first nested search cycle
building on PDL of patterns for the second nested search cycle
second nested search cycle
:

```

(Ordered search means that the pattern search can be performed without affecting the subsequent searches.) First, the patterns that can be simply matched are extracted from the set. Then, after building the patterns to search for onto the PDL, the search cycle is executed using the search instructions outlined in section 5.2. Naturally, there can be more than one search cycle, because there can be multiple separate groups of patterns whose search can require a withdrawal (search reinstatement and continuation) of a previous pattern match. A search cycle begins with a *search_set???* instruction and has the following structure:

```

search_set???
matching of patterns whose search can be ordered;
first nested search cycle
continue_search???
second nested search cycle
:
end_search???

```

The question marks used here mean that the instruction can have a suffix to discriminate between versions of the same instruction that deal with different cases. The structure is quite similar to that of the first level, except for the building of patterns on the PDL, which is made on the first level once and for all, and the instruction *continue_search???*, which separates the search cycles at the same level.

6.2.2 Subset assertions

Subset assertions require mapping one set into another. This is performed by **pattern mapping**, which allows us to iterate the extraction of set elements, using instructions previously described in section 5.3. The general SAL code scheme for a subset assertion is:

```

matching phase
searching phase
building on PDL of patterns for the mapping cycle
mapping cycle

```

A mapping cycle begins with a *map???* instruction and has the following structure:

```

map???
matching phase in mapping cycle

```

```

    searching phase in mapping cycle
    mapping cycle in mapping cycle
    end_map???
```

The matching and searching phases are the same as for equational assertions, but with a *continue_map* instead of a *continue_search* or a *fail*. Also, at the top and bottom of a subset assertion's code frame, there will be a *try_sub_and* and *proceed_sub* instruction in place of the *try_equ_else* and *proceed_sub* respectively. These mechanisms set up choice points at the call boundary to facilitate withdrawal of bindings even on success of matching, in order to exhaust all of the successful matches which must occur in the subset assertion matching.

7 Implicit optimization algorithm for SAL code production

At this point, given a fairly powerful set of instructions to execute on different kinds of architectures and a general scheme for generating code for matching over sets, it remains to outline an algorithm for compiling SAL code from SEL code. Such an algorithm should show the best sequence of searching, matching or iterating instructions, according to the following three goals:

- the sequence of operations over the pattern must lead to a correct implementation of a targeted high level construct
- both searching and mapping nesting level must be minimized
- the number of instructions that produce a remainder set must be minimized.

The first aim maintains the correctness of the proof, whereas the two others limit the computational overhead of the system. The second goal avoids the need to stack frames too deeply, thus leading to a greatly streamlined flow of execution. The third goal limits both the execution and the space requirements, since the number of remainder sets risks being huge. This implies that whereas the first aim is a strict requirement of the system, the second and third are desirable features pertaining to the realm of optimization strategies.

Before going into details about the **implicit optimization algorithm** (IOA), the main problems characterizing the optimized use of matching, searching and mapping instructions will be introduced. For this purpose, the two pivotal concepts are **pattern dependence** and **pattern interference**.

7.1 Pattern dependence and interference

Pattern dependence is closely related to the unifiability characteristics of patterns. First consider the definition of **ground unifiability**:

A pattern t_1 is ground unifiable to another one, t_2 if the set of ground terms that are unifiable to t_1 is a subset of the ground terms that are unifiable to t_2 .

Now consider three kinds of pattern dependence.

Element dependence A pattern depends on another one such that the latter is ground unifiable to the first. For example, in:

$$h(\{r(X), r(a) \mid _ \}) = \dots$$

the pattern $r(X)$ has an element dependence upon the pattern $r(a)$ since the ground terms satisfying $r(X)$ include $r(a)$. Patterns that are unifiable with each other, as in

$$h(\{s(X), s(Y) \mid _ \}) = \dots$$

are element dependent upon one another as well.

Rest dependence. The remainder of a set depends on all the other patterns being matched to the set. In

$$f(\{s(X), r(Y) \mid R\}) = \dots$$

the remainder R depends on matching both $s(X)$ and $r(Y)$.

Dummy rest dependence even a dummy remainder, represented by ‘ $_$ ’, depends on all the patterns of the set. In the first example,

$$h(\{r(X), r(a) \mid _ \}) = \dots$$

the remainder $_$ depends on both $r(X)$ and $r(a)$.

Note that here pattern dependence regards only patterns of the same set.

Two patterns cannot be matched separately (i.e. independently) if they are **interfering**. Two patterns are interfering if the choice of one poses a constraint to the choice for satisfying the other. So, for instance, in the assertion:

$$\text{get_grandpa}(\{father(Grandpa, Dad), father(Dad, _) \mid _ \}) = Grandpa,$$

$father(Grandpa, Dad)$ and $father(Dad, _)$ are interfering by means of the term Dad . In particular, three kinds of interference are worth singling out.

Rest by rest interference If an unbound variable is declared as the remainder of two different sets, then each of the patterns of the two sets can not be matched separately from the others because if the matching of the two interfering remainders fails, each matching must be retracted. For instance in the assertion

$$\text{foo1}(Bound, \{f(X) \mid Bound\}, \{Y \mid Bound\}) = (X, Y).$$

there is no ‘rest by rest interference’ because the remainders are bound by the first argument of the assertion, whereas in the assertion

$$\text{foo2}(\{f(X) \mid Unbound\}, \{Y \mid Unbound\}) = (X, Y).$$

the two set remainders are interfering.

Element by rest interference If a pattern contains an unbound variable that is also the remainder of a set, then each of the patterns of the set can not be matched separately from the others, for the same reason as for ‘rest by rest interference’. An example is:

$$\text{foo3}(\{h(X) \mid R\}, \{set(R) \mid _ \}) = X.$$

Element by element interference This is the case of *get_grandpa/1* above involving a dependency of two elements containing the same unbound variable.

Two patterns cannot be matched separately even if their matching operations cannot be ordered, as in the assertion:

$$\text{may_not_be_ordered}(\{h(1, X), h(Y, 2) \mid _ \}) = X - Y.$$

To fully understand the ordering which lies implicitly in the pattern matching and in pattern mapping instructions, the notion of the **most general unifier (MGU) space** of a pattern must be defined:

The **MGU space** of a pattern is the set of all terms matching the given pattern.

The rule is that two pattern matching operations may be ordered if the MGU space of a pattern is strictly contained in the MGU space of the other.

The reason for such a rule is that when it is applied, it is never the case that a potential candidate for pattern matching is missed just because the search space has been wrongly limited. For instance, matching operations of the patterns $p(a, X, b)$ and $p(X, Z, b)$ may be ordered because the MGU space of $p(Y, Z, b)$ strictly contains the MGU space of $p(a, X, b)$. It is obvious that pattern ordering does not concern only patterns belonging to the same set. Pattern matching ordering preserves result correctness and limits the number of nested searches.

Now, since correctness is preserved, it is possible to turn to optimization. The IOA follows these fundamental steps:

- (a) Definition of a graph representing the patterns to be matched.
- (b) Initialization of internal graph structures.
- (c) Extraction of matchable patterns.
- (d) Construction of particular pattern sets called **interference classes**.

For each interference class *do* (applying this loop first to the search and then to the map patterns)

(e) Selection of the best pattern to search or map.

(f) *if* the associated interference class is empty

then stop

else re-apply, from step (c), this algorithm on patterns contained in this newly defined interference class.

Each step is executed differently according to the type of assertion that is analysed: if the assertion is a subset assertion then the algorithm works in **mapping mode**, otherwise it works in **searching mode**. In searching mode, each pattern selected in step (e) will be searched using a *search_set* instruction. In mapping mode, a pattern is mapped only if it takes part in the construction of the result, i.e. if it contains an **unbound** variable that is also contained in the right hand side of the subset assertion. Each of these steps (a–f) outlined above is detailed in the following subsections (7.2–7.6).

7.2 Definition of patterns and initialization of internal graph structures

The algorithm needs two graph structures, the **interference graph** and the **dependence graph**. The former summarizes all the interference relations between patterns appearing in the left side of the assertion, and the latter describes their matching order. Each vertex in the graphs corresponds to a different pattern,

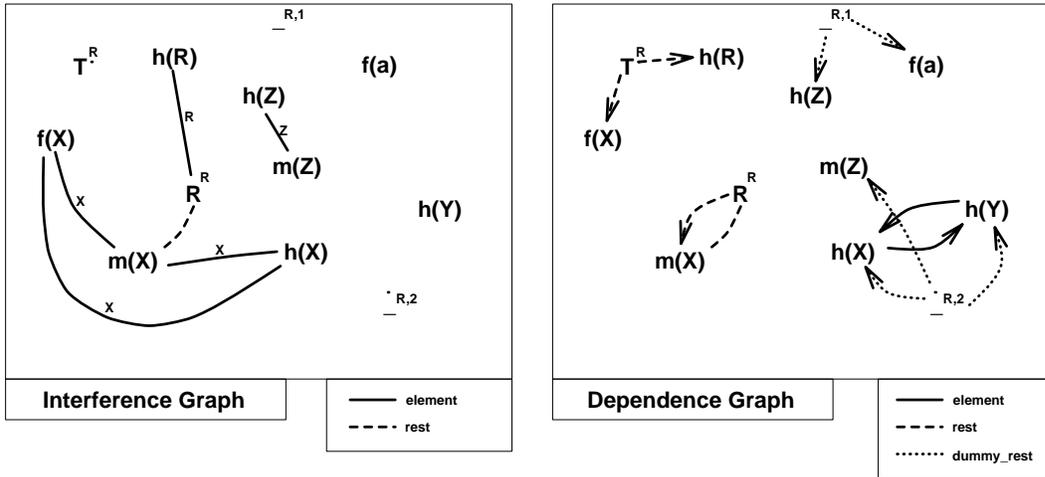


Figure 13: Interference and dependence graphs for the assertion *foo4/3*.

while edges represent interference or precedence relations between two patterns. The dependence graph is oriented, in order to show precedence; the interference graph, on the other hand, is not oriented since two interfering patterns must be selected in the same search cycle and the order of selection is not important.

Figure 13 shows the interference and dependence graphs for the assertion head:

$$foo4(\{f(X), h(R) \mid T\}, \{f(a), h(Z) \mid -\}, \{m(X) \mid R\}, \{m(Z), h(X), h(Y) \mid -\}) \dots$$

In this diagram each ‘remainder’ is denoted by an R followed by the number of unbound variables depending from it.

The patterns $f(X)$, $m(X)$ and $h(X)$ are interfering ‘element by element’ by means of the unbound variable X ; the pattern $m(X)$ is also interfering ‘element by rest’ with the remainder R , which is ‘element by rest’ interfering with $h(R)$. Each edge is indexed by the variable causing interference. In the dependence graph, edges are oriented, pointing from the dependent pattern to the one on which it depends: the remainder T , for example, is ‘rest depending’ on $f(X)$ and $h(R)$, and the patterns $h(X)$ and $h(Y)$ depend upon one another. More formally, an **interference graph** of a head of a clause is a graph defined as follows.

- **Nodes:** all the elements in the sets and all the remainders
- **Arcs:** between each node sharing a variable and between each node and the associated remainder, if such remainder is connected with at least one element arc to any other node.

A **dependence graph** is formed with the same nodes as the interference graph and its arcs are defined by the three kinds of dependence relations (element, rest, dummy rest).

7.3 Extraction of matchable patterns

Many patterns do not require further manipulation; they can be matched directly by a *match_set* instruction. Since pattern matching has to be ordered, the matching of one pattern might leave another pattern yet

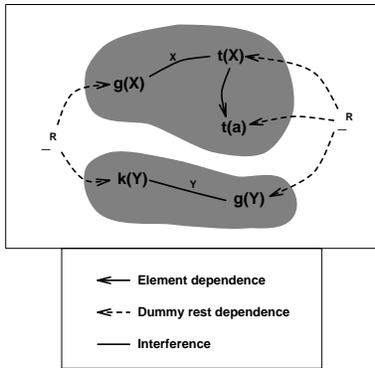


Figure 14: Interference classes for the assertion *foo5/2*.

to be matched. For this reason, the matchable pattern extraction is repeated until some patterns become ground instances. A pattern is matchable if it is ground or if there is at least a pattern interfering with it or preceding it in matching order.

After each selection of a searching or a mapping pattern (to be discussed later), the matchable patterns extraction is repeated since such selection can make some patterns becomes ground instances.

The assertion

$$foo5(\{g(X), k(Y) \mid _ \}, \{t(a), t(X), g(Y) \mid _ \}) = f(X, Y)$$

helps to make these concepts clearer. Figure 14 shows the interference classes, for which the SAL code is given in Figure 15.

The pattern $t(a)$, being ground, can be matched with a simple *match_set_rem* that uses $Z5$ to store the remainder of the set for the next searches. Then, the pattern $g(X)$ is searched for in the set $Z1$; after it has been selected, the pattern $t(X)$ is ground and can be simply matched with a *match_set*. Now the instruction *store_ind_functor* stores on the heap a functor, copying into register $Z3$ a reference to it. The following *store_value* copies onto the heap the contents of $Z6$ and $Z7$, completing the building of the result.

7.4 Construction of interference classes

The following definitions pertain to interference classes.

The **order of an interference subclass** is the number of set elements that it contains.

Each pattern element forms a **candidate interference subclass** containing the element itself, having order 1. Dummy rests are an exception to this rule, not counted as true patterns.

Two interference subclasses or candidate interference subclasses belong to a higher order interference subclass composed of their union if even just one element of the first subclass interferes with or depends on an element of the second subclass.

We define an **interference class** as the interference subclass having the maximum possible order.

```

foo5/3:
  try_eq_else - 1          ; This is the only clause foo5/3
  get_set Z1 A1           ; 1st argument
  get_set Z2 A2           ; 2nd argument
  get_variable Z3 A3      ; result
  start_set_match Z4      ; start the set match: Z4 is the ref to PDL
  store_pdl_functor t/1   ; store t/1 in PDL for matching
  store_pdl_const char a  ; store a in PDL for matching
  match_set_rem Z2 Z4 Z5 Z6 ; match Z4 on Z2 putting the remainder in Z5 and set Z6 if fail
  fail Z6                 ; fail the clause if Z6 is set
  start_set_match Z4      ; start new match: Z4 is ref
  store_pdl_functor g/1   ; store g/1 in PDL
  store_pdl_variable Z6   ; store ref to Z6 in PDL
  start_set_match Z7      ; start a new match: Z7 is ref
  store_pdl_functor t/1   ; store t/1 in PDL
  store_pdl_value Z6      ; store value of Z6 in PDL
  search_set Z1 Z8 Z4 Z9 end ; start the searching process in the set Z1 over the first pattern identified by Z4
                          ; - fail flag is Z9 - Z8 is a reference to Z1
                          ; if cannot match goto end

start:
  match_set Z5 Z7 Z10     ; match Z7 against Z5 and set Z10 if fail
  end_search Z10 Z8 Z4 Z9 start ; if Z10 try a new value in Z8 that matches Z4 and goto start else set Z9
end:
  fail Z9                 ; fail if Z9 is set
  start_set_match Z4      ; start new match: Z4 is ref
  store_pdl_functor g/1   ; store g/1 in PDL
  store_pdl_variable Z7   ; store ref to Z7 in PDL
  start_set_match Z8      ; start a new match: Z8 is ref
  store_pdl_functor k/1   ; store k/1 in PDL
  store_pdl_value Z7      ; store value of Z7 in PDL
  search_set Z5 Z9 Z4 Z10 end1 ; start the searching process
                          ; over the first pattern identified by Z4 - fail flag is Z10

start1:
  match_set Z1 Z8 Z11     ; match Z8 against Z1
  end_search Z11 Z9 Z4 Z10 start1 ; if Z11 try a new value in Z9 and goto start1 else set Z10
end1:
  fail Z10                ; fail if Z9 is set
  store_ind_functor f/2 Z3 ; store the functor f/2 in the place of the result
  store_value Z6          ; store Z6 as 1st arg. of f
  store_value Z7          ; store Z7 as 2nd arg. of f
  proceed_eq              ; return

```

Figure 15: SAL code for interference classes.

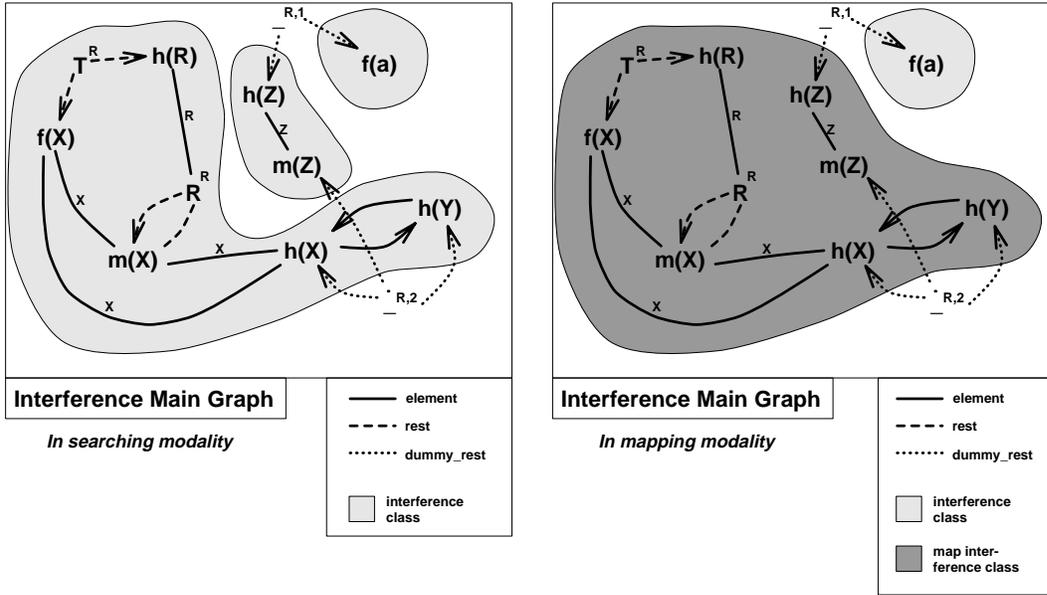


Figure 16: Interference classes in equational and subset assertion *foo4/3*.

An interference class groups together the patterns that cannot be satisfied by a simple matching operation, requiring a *search_set* or a *map* construct to yield a correct and complete solution. From the IOA point of view, an interference class groups interfering patterns, patterns whose matching depends on the matching of interfering patterns, i.e. patterns interfering with a dependent pattern.

Interference classes may be singled out by searching connected graphs in a composite graph made from the union of the interference and dependence graphs. Each connected sub-graph corresponds to an interference class. Graph connectivity does not involve *dummy_rest* edges, since they are only used to discriminate between instructions that return the remainder of the set or not. For instance, interference classes for *foo4/3* can be seen in Figure 16. The *dummy_rest* edges are visible, but do not affect the interference classes definition.

In ‘mapping mode’, two patterns could be seen as interfering if both contain an unbound variable present in the right side of the assertion since these elements can yield variable substitutions that take part in building the result. All the other elements will be selected with a *match_set* or a *search_set* instruction since one has only to verify their presence in the set. This strategy produces the nesting of mapping constructs needed to have a complete result while optimizing the execution time.

An example is given by the following two assertions:

$$\begin{aligned}
 &foo6(\{g(X),k(Y) \mid -\}, \{t(a),t(X),g(Y) \mid -\}) \text{ contains } f(X,Y). \\
 &foo7(\{g(X),k(Y) \mid -\}, \{t(a),t(X),g(Y) \mid -\}) \text{ contains } f(X).
 \end{aligned}$$

Figures 17 and 18 show the interference classes for these two assertions. Since the variable *Y* does not appear in the right side of *foo7/3*, there are two interference classes, whereas one class is associated to *foo6/3*. Hence the SAL code for *foo6/3* is as shown in Figure 19.

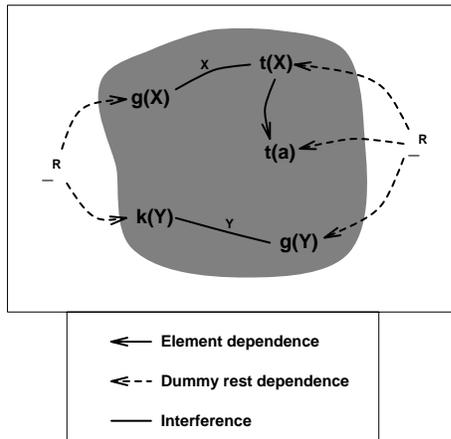


Figure 17: Interference class for the assertion *foo6/3*.

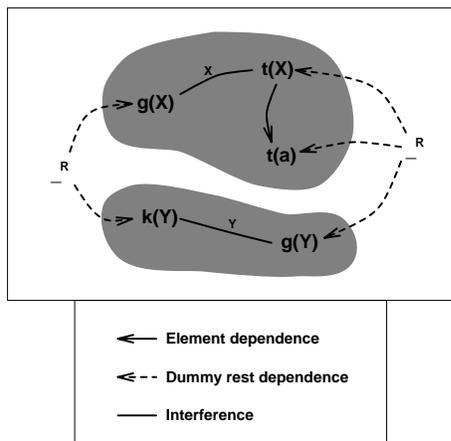


Figure 18: Interference classes for the assertion *foo7/3*.

```

foo6/3:
  try_sub_and - 1 ; This is the only clause foo6/3
  get_set Z1 A1 ; 1st argument
  get_set Z2 A2 ; 2nd argument
  get_variable Z3 A3 ; result
  start_set_match Z4 ; start a new match: Z4 is ref
  store_pdl_functor t/1 ; store t/1 in PDL
  store_pdl_const char a ; store a in PDL
  match_set_rem Z2 Z4 Z5 Z6 ; match Z4 against Z2, remainder in Z5 if fail, set Z6
  fail Z6 ; if Z6 is set fail the clause
  start_set_match Z4 ; start a new match: Z4 is ref
  store_pdl_functor g/1 ; store g/1 in PDL
  store_pdl_variable Z6 ; store a ref to Z6 in PDL
  start_set_match Z7 ; start a new match: Z7 is ref
  store_pdl_functor t/1z ; store t/1z in PDL
  store_pdl_value Z6 ; store the value of Z6 in PDL
  start_set_match Z8 ; start a new match: Z8 is ref
  store_pdl_functor g/1 ; store g/1 in PDL
  store_pdl_variable Z9 ; store a ref to Z9 in PDL
  start_set_match Z10 ; start a new match: Z10 is ref
  store_pdl_functor k/1 ; store k/1 in PDL
  store_pdl_value Z9 ; store the value of Z9 in PDL
  map_over_matching Z1 Z11 Z4 end1 ; start the outer matching
start1:
  match_set Z5 Z7 Z12 ; match Z7 against Z5, if fail set Z12
  continue_map_matching Z12 Z11 Z4 start1 end1 ; if Z12 is set then goto start1 and try a new value
  map_over_matching Z5 Z13 Z8 end2 ; start the inner match
start2:
  match_set Z1 Z10 Z14 ; match Z7 against Z5, if fail set Z12
  continue_map_matching Z14 Z13 Z8 start2 end2 ; if Z12 is set then goto start1 and try a new value
  store_sda_functor f/1 Z12 ; store in the SDA f/1 and save its ref in Z12
  store_sda_value Z6 ; store in SDA the value of Z6
  store_sda_value Z7 ; store in SDA the value of Z7
  insert Z12 Z3 ; Z3 union Z12 → Z3
  end_map_over_matching Z13 Z8 start2 ; end the inner match
end2:
  end_map_over_matching Z11 Z4 start1 ; end the outer match
end1
  proceed_sub

```

Figure 19: SAL code for *foo6/3*.

After the matching of the ground term $t(a)$ and the building of the other patterns on the PDL, the mapping cycle begins with the selection of $g(X)$ performed by the first level *map_over_matching*. Then, the pattern $t(X)$, now matchable since X is ground, is selected with a simple *match_set*. The *continue_map_matching* separates the **matching in mapping phase** from the following nested mapping cycle, that begins with a *map_over_matching* to select the pattern $g(Y)$: now the pattern $k(Y)$ is matchable, and if its matching succeeds, the element $f(X, Y)$ is built on the heap and a reference to it is inserted into the result set (indexed by Z). This is repeated until all the matches are satisfied. Note that the building of the result is performed at the end of the innermost mapping cycle, when all the patterns have been successfully matched. The *store_sda* instructions build the element of the resulting set in a zone of the current environment called **set dynamic area** (SDA), which is then inserted in the set by the *insert* instruction. The SDA is used to allocate sets of unbounded size, e.g. those generated by calls to further set-valued set assertions.

The SAL code for *foo7/3* is given in Figure 20. The situation is now quite different: the variable Y does not appear in the result. Therefore, there are two interference classes, as shown in Figure 18: the one grouping elements bound by the variable Y refers to the search cycle, while the other refers to the mapping cycle. As always, first the matchable pattern, $t(a)$, is selected with a *match_set*, then patterns $g(Y)$ and $k(Y)$ are built on the PDL for the search cycle. After the search cycle, patterns $g(X)$ and $t(X)$ are built on the PDL for the mapping cycle, in which, after each selection of two elements matching the patterns, the functor $f(X)$ is built on the heap and the reference to it is inserted in the set returned as result.

Each interference class is separately processed in such a way as to reduce nesting of mapping or searching cycles. Whenever a pattern is selected to produce **searching** code, the interference class may be divided into subclasses. Each of these subclasses is processed independently of the others so that a searching nesting may contain additional non-interfering searching nesting at the same level. This reduce the execution time, since a failure with a subsequent withdrawal in a searching nesting does not affect the other searching cycles, performed independently. The smaller the subclasses are, the more the computational order of searching paradigm is reduced. For instance, consider the assertion:

$$foo8(\{m(X, Y), p(Z), h(X, Z), f(Y) \mid -\}) = (X, Y, Z).$$

Its SAL translation is given in Figure 21. Note the pattern chosen for the first level search cycle is $h(X, Z)$, which divides the remaining patterns into two subclasses, one composed of the pattern $p(Z)$, which, being ground, is managed by a *match_set*, and the other by the patterns $f(Y)$ and $m(X, Y)$, which originates a search cycle at the same level as the *match_set*.

The same reasoning is valid for subset assertions, with one stipulation. As already seen, patterns taking part in the construction of a subset assertions' result produce a nesting of mapping cycles in such a way as to produce a complete result. Selecting one of these patterns could partition the mapping interference class into simple interference subclasses containing patterns that do not take part in the result construction. These interference classes produce a search code at the same level as the next mapping nesting since the patterns belonging to these classes are only constraints to the choice of the right elements to produce the result. To allow such a construction, interference classes are processed before mapping interference classes. Compare for example the difference between the SAL code of the following assertions:

$$\begin{aligned} &foo9(\{h(X, Z), p(Z), f(Y), m(X, Y) \mid -\}) \text{contains} \{X\}, \\ &foo10(\{h(X, Z), p(Z), f(Y), m(X, Y) \mid -\}) \text{contains} \{X, Y\}. \end{aligned}$$

```

foo7/3:
  try_sub_and - 1 ; This is the only clause foo7/3
  get_set Z1 A1 ; 1st argument
  get_set Z2 A2 ; 2nd argument
  get_variable Z3 A3 ; result
  start_set_match Z4 ; start a new match: Z4 is ref
  store_pdl_functor t/1 ; store t/1 in PDL
  store_pdl_const char a ; store a in PDL
  match_set_rem Z2 Z4 Z5 Z6 ; match Z4 against Z2, rem. in Z5, if fail set Z6
  fail Z6 ; if Z6 is set, fail the clause
  start_set_match Z4 ; start a new match: Z4 is ref
  store_pdl_functor g/1 ; store g/1 in PDL
  store_pdl_variable Z6 ; store a ref to Z6 in PDL
  start_set_match Z7 ; start a new match: Z7 is ref
  store_pdl_functor k/1 ; store k/1 in PDL
  store_pdl_value Z6 ; store the value of Z6 in PDL
  search_set Z5 Z8 Z4 Z9 end ; match Z4 against Z5 and set Z9 if fail
start:
  match_set Z1 Z7 Z10 ; match Z7 against Z1 and set Z10 if fail
  end_search Z10 Z8 Z4 Z9 start ; repeat until either Z10 is not set
  ; or set Z8 is completely analysed
end:
  fail Z9 ; if Z9 is set fail the clause
  start_set_match Z4 ; start a new match: Z4 is ref
  store_pdl_functor g/1 ; store g/1 in PDL
  store_pdl_variable Z7 ; store a ref to Z7 in PDL
  start_set_match Z8 ; start a new match: Z8 is ref
  store_pdl_functor t/1 ; store t/1 in PDL
  store_pdl_value Z7 ; store the value of Z7 in PDL
  map_over_matching Z1 Z9 Z4 end1 ; match Z4 against Z1(copied in Z9)
start1:
  match_set Z1 Z8 Z10 ; match Z8 against Z1 and set Z10 if fail
  continue_map_matching Z10 Z9 Z4 start1 end1; if Z10 set then goto start1 and try a new value
  store_sda_functor f/1 Z12 ; store f/1 in SDA and make Z12 refer to it
  store_sda_value Z7 ; store the value of Z7 in SDA
  insert Z12 Z3 ; Z3 union Z12 → Z3
  end_map_over_matching Z10 Z4 start1 ; repeat until Z10 is completely analysed
end1:
  proceed_sub ; return

```

Figure 20: SAL code for *foo7/3*.

```

foo8/2:
  try_eq_else - 1          ; This is the only equation
  get_set Z1 A1           ; 1st argument
  get_variable Z2 A2      ; result
  start_set_match Z3      ; start a new match: Z3 is ref
  store_pdl_funcutor h/2  ; store h/2 in PDL
  store_pdl_variable Z4   ; store a ref to Z4 in PDL
  store_pdl_variable Z5   ; store a ref to Z5 in PDL
  start_set_match Z6      ; start a new match: Z6 is ref
  store_pdl_funcutor p/1  ; store p/1 in PDL
  store_pdl_value Z5      ; store the value of Z5 in PDL
  start_set_match Z7      ; start a new match: Z7 is ref
  store_pdl_funcutor f/1  ; store f/1 in PDL
  store_pdl_variable Z8   ; store a ref to Z8 in PDL
  start_set_match Z9      ; start a new match: Z9 is ref
  store_pdl_funcutor m/2  ; store m/2 in PDL
  store_pdl_value Z4      ; store the value of Z4 in PDL
  store_pdl_value Z8      ; store the value of Z8 in PDL
  search_set Z1 Z10 Z3 Z11 end0 ; match Z3 against Z1 and set if fails
start0:
  match_set Z1 Z6 Z12     ; match Z6 against Z1 and set Z12 if fail
  continue_search Z12 Z10 Z3 Z11 start0 end0; if Z12 set then goto start0 and try a new value
  search_set Z1 Z13 Z7 Z12 end1 ; match Z7 against Z1 and set Z12 if fails
start1:
  match_set Z1 Z9 Z14     ; match Z9 against Z1 and set Z14 if fail
  end_search Z14 Z13 Z7 Z12 start1 ; repeat until either Z14 is not set
  ; or set Z13 is completely analysed
end1:
  fail Z12                ; fail if Z12 is set
  end_search Z12 Z10 Z3 Z11 start0 ; repeat until either Z12 is not set
  ; or set Z10 is completely analysed
end0:
  fail Z11                ; if Z11 is set fail the clause
  store_ind_funcutor (/)/3 Z2 ; store the functor (/)/3 in the region
  ; of the result identified by Z2
  store_value Z3          ; store Z3 as first argument of (/)
  store_value Z8          ; store Z8 as second argument of (/)
  store_value Z5          ; store Z5 as third argument of (/)
  proceed_eq             ; return

```

Figure 21: SAL translation of *foo8* assertion.

```

foo9/2:
  try_sub_and - 1          ; This is the only equation
  get_set Z1 A1           ; 1st argument
  get_variable Z2 A2      ; result
  start_set_match Z3      ; start a new match: Z3 is ref
  store_pdl_functor h/2   ; store h/2 in PDL
  store_pdl_variable Z4   ; store a ref to Z4 in PDL
  store_pdl_variable Z5   ; store a ref to Z5 in PDL
  start_set_match Z6      ; start a new match: Z6 is ref
  store_pdl_functor p/1   ; store p/1 in PDL
  store_pdl_value Z5      ; store the value of Z5 in PDL
  start_set_match Z7      ; start a new match: Z7 is ref
  store_pdl_functor f/1   ; store f/1 in PDL
  store_pdl_variable Z8   ; store a ref to Z8 in PDL
  start_set_match Z9      ; start a new match: Z9 is ref
  store_pdl_functor m/2   ; store m/2 in PDL
  store_pdl_value Z4      ; store the value of Z4 in PDL
  store_pdl_value Z8      ; store the value of Z8 in PDL
  map_over_matching Z1 Z10 Z3 end ; match Z3 against Z1 (copied in Z10)
start:
  match_set Z1 Z6 Z11     ; match Z6 against Z1 and set Z11 if fail
  continue_map_matching Z11 Z10 Z3 start end; if Z11 set then
  ; goto start and try a new value
  search_set Z1 Z12 Z7 Z11 end1 ; match Z7 against Z1 and set Z11 if fail
start1:
  match_set Z1 Z9 Z13     ; match Z9 against Z1 and set Z13 if fail
  end_search Z13 Z12 Z7 Z11 start1 ; repeat until either Z13 isnot set
  ; or set Z12 is completely analysed
end1:
  continue_map_matching Z11 Z10 Z3 start end ; if Z11 set then goto start and
  ; try a new value
  insert Z4 Z2            ; Z2 union Z4 → Z2
  end_map_over_matching Z10 Z3 start ; repeat until Z10 is completely analysed
end:
  proceed_sub             ; return

```

Figure 22: SAL code for *foo9/2*.

The SAL code for *foo9/2* is given in Figure 22 and for *foo10/2* in Figure 23.

The pattern $h(X, Z)$ is chosen as the best pattern to map since the interference class composed of all the patterns is divided into two subclasses, one formed by the pattern $p(Z)$, which is now ground and can be matched with a simple *match_set*, and the other formed by $f(Y)$ and $m(X, Y)$. This latter subclass bears a searching cycle at the same level as the preceding *match_set*, since the variable Y does not take part in the result building. Then the *continue_map_matching* manages a possible failure, and finally there is the inner mapping cycle, which is in this case is simply composed of the phase of building of the result, the instruction *insert*.

This time both X and Y take part in building the result. Now there is a mapping cycle for the pattern $h(X, Z)$, a matching phase for $p(Z)$, a nested mapping cycle for the remaining patterns $f(Y)$ and $m(X, Y)$.

7.5 Selection of the best pattern to search or map

The selection of the pattern to be searched or mapped is a crucial step of the algorithm. The selection must produce as many partitions of the interference class as possible, in order to have many small subclasses to limit the complexity of the algorithm, thus saving execution time.

First a definition is needed:

A pattern is said to be **more ground** than another if the vertex related to the former has more entering dependence edges than the vertex of the latter.

For instance, in:

$$p(\{h(a, b, X), h(a, Y, X), h(Z, Y, X) \mid _ \}) = (X, Y, Z).$$

the pattern $h(a, b, X)$ is **more ground** than $h(a, Y, X)$, which is **more ground** than $h(Z, Y, X)$.

So the criterion to select the best pattern to search for is:

Among the **most ground** patterns of the interference class choose the pattern interfering with the **largest** number of patterns of the class.

To choose the best pattern to map, the criterion is:

Choose the pattern interfering with the **largest** number of patterns of the class among the **most ground** patterns of the interference class containing ‘right side’ unbound variables.

The interference order of a pattern may be retrieved in the same way, counting the number of interfering edges connected to the related vertex of the interference class’s composite interference-dependence graph.

Subsequently, the algorithm retries to select matchable elements and to partition the interference class into subclasses, repeating the same steps considering as interference class each of the subclasses found.

7.6 Termination of the algorithm

The algorithm runs recursively until the interference class is empty. Then it backtracks to a nesting level with at least one non-empty interference class and restart the nested process. When there are no more interference classes the IOA ends its execution.

```

foo10/2:
  try_sub_and - 1          ; This is the only equation
  get_set Z1 A1           ; 1st argument
  get_variable Z2 A2      ; result
  start_set_match Z3      ; start a new match: Z3 is ref
  store_pdl_functor h/2   ; store h/2 in PDL
  store_pdl_variable Z4   ; store a ref to Z4 in PDL
  store_pdl_variable Z5   ; store a ref to Z5 in PDL
  start_set_match Z6      ; start a new match: Z6 is ref
  store_pdl_functor p/1   ; store p/1 in PDL
  store_pdl_value Z5      ; store the value of Z5 in PDL
  start_set_match Z7      ; start a new match: Z7 is ref
  store_pdl_functor f/1   ; store f/1 in PDL
  store_pdl_variable Z8   ; store a ref to Z8 in PDL
  start_set_match Z9      ; start a new match: Z9 is ref
  store_pdl_functor m/2   ; store m/2 in PDL
  store_pdl_value Z4      ; store the value of Z4 in PDL
  store_pdl_value Z8      ; store the value of Z8 in PDL
  map_over_matching Z1 Z10 Z3 end ; match Z3 against Z1 (copied in Z10)
start:
  match_set Z1 Z6 Z11     ; match Z6 against Z1 and set Z11 if fail
  continue_map_match Z11 Z10 Z3 start end ; if Z11 set then goto start and try a new value
  map_over_matching Z1 Z12 Z7 end1 ; match Z7 against Z1 (copied in Z12)
start1:
  match_set Z1 Z9 Z13     ; match Z9 against Z1 and set Z13 if fail
  continue_map_match Z13 Z12 Z7 start1 end1 ; if Z13 set then goto start1 and try a new value
  insert Z4 Z2            ; Z2 union Z4 → Z2
  insert Z8 Z2            ; Z2 union Z8 → Z2
  end_map_over_matching Z12 Z7 start1 ; repeat until Z12 is completely analysed
end1:
  end_map_over_matching Z10 Z3 start ; repeat until Z10 is completely analysed
end:
  proceed_sub             ; return

```

Figure 23: SAL code for *foo10/2*.

8 Architecturally independent optimizations

The global optimization of the matching strategies is one large part of the optimization effort targeting the initial compilation in an architecturally independent way. Subsequently a number of peep-hole optimizations have been effected, yet in an architecturally independent fashion. Particular attention has been devoted to implementing these optimization techniques in the SEL to SAL compiler in order to have a more efficient execution, in terms of both space and execution time.

Last call optimization and **environment trimming** are used to reduce dynamic memory space allocation, similarly to what has been done in the implementation of the WAM (Ait-Kaci 1990). Furthermore, **optimized register allocation** and **redundant instruction elimination** have been developed to reduce code length and to limit machine resource usage.

8.1 Last call optimization

When the execution of an assertion requires calling another assertion, then the current **environment** must be saved on the stack, so as to restart execution from the right point when the call returns. The environment in SAM includes both control flow registers (for instance the program counter) and the main registers, which contain data that must be saved to be used after returning from the call. These are also called **permanent variables**, as opposed to **temporary variables** that do not have to be saved.

Last call optimization (LCO) (Ait-Kaci 1990), is based on the fact that the 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. In this case, the environment can be deallocated just before the last assertion call. The execution of the last call may be performed as the last operation. Thus, the environment is deallocated, if it was allocated, and the address of the last call may be directly assigned to the program counter. This saves considerable memory space when executing nested assertions.

An in-depth description of LCO can be found in Ait-Kaci (1990).

8.2 Environment trimming

Like LCO, environment trimming aims to reduce stack space used to store the environments of assertions whose execution is suspended, saving only the ‘live’ part of the environment, i.e. the registers that will be used after returning from a call. It consists of ordering the permanent variables in the environment of the assertion as well as reflecting the ordering of their last occurrence in the right side of an assertion. More specifically, the variable whose last occurrence is the last of all will be the first on the environment, and so on. This requires numbering the main registers in inverse order of first occurrence. Then each *call* instruction specifies the number of registers to keep saved and the environment of the called function will override all the others resulting in a saving of memory space. Note that the number of main registers to be saved decreases as the execution of the assertion proceeds.

An in-depth description of environment trimming can be found in Ait-Kaci (1990).

8.3 Optimized register allocation

Register optimization aims to reduce the number of registers used for the execution of each assertion. Optimized register allocation (ORA) consists of locating **reusable** registers and reusing them instead of allocating new registers. Location of reusable registers is carried out both in the main compilation phase and in the subsequent optimization phase, considering the scope of the registers used in particular code regions. For example, recalling the SAL code for the assertion *foo5/2* in section 7.3, the register Z6 is

first used as a Boolean register in the *match_set_rem*, then reused to store the variable *X* onto the PDL; registers *Z4* and *Z7*, used to store the patterns built on the PDL for the first search cycle, are reused to store the patterns for the second cycle, since their contents will no longer be needed.

8.4 Elimination of redundant instructions

It sometimes happens that one or more instructions produced in the compilation process turn out to be redundant. Redundant instructions elimination (RIE) consists of assigning temporary registers in such a way as to recognize and make use of values already present in registers. For example, it is possible to:

1. eliminate *get_variableXiAj* lines from the code, replacing everywhere register *Xi* with *Aj*, if *Aj* is not used in the code lines between the *get* and the last reference to *Xi*.
2. eliminate all the *put* instructions in code sequences of the kind:

```

get_variable Xi Aj
sequence of instructions not using Aj
put_value Xi Aj

```

Consider, for example, the assertion:

$$\text{sum}(X, Y) = X + Y.$$

Its initial SAL translation would be:

```

sum/3 :
  try_eq_else - 1
  get_variable X4 A1
  get_variable X5 A2
  get_variable X6 A3
  put_value X4 A1
  put_value X5 A2
  put_value X6 A3
  execute + /3

```

The optimized code is simply:

```

sum/3 :
  try_eq_else - 1
  execute + /3
  proceed_eq

```

since the *Ai* registers already contain the correct values.

This optimization strategy is tightly bound to the sequence of arguments defined in the left side of the assertion. It may happen that between two assertions of the same predicate only one can be optimized using RIE because the other has arguments incompatibly assigned. This can be avoided by knowing the eventual commutative property of the assertion called inside the code. Considering the assertion

$$\text{sum}(X, Y) = Y + X,$$

the same optimized SAL code still gives a correct result. Of course, these considerations are valid only for a restricted set of built-in operations.

9 A sample compilation

The aim of this section is to summarize the feature of the system through an analysis of a whole compilation of a chunk of SL code. Reconsider the SL program:

$$\begin{aligned} \text{get_abs_grt_than}(X, \text{Nums}) &= \text{get_grt_than}(X, \text{Nums}) \text{ union } \text{get_grt_than}(X, \text{inv}(\text{Nums})) \\ \text{get_grt_than}(X, \text{Numbers}) &= \{N : N \text{ in } \text{Numbers}; N > X\}. \\ \text{inv}(\text{Numbers}) &= \{-N : N \text{ in } \text{Numbers}\}. \end{aligned}$$

This program returns a set whose elements are extracted from a given set *Nums* if their absolute value is greater than a given number *X*.

The SL code is first of all translated to SEL intermediate code:

$$\begin{aligned} \text{get_abs_grt_than}(X, \text{Numbers}) &\text{ contains } \text{get_grt_than}(X, \text{Numbers}). \\ \text{get_abs_grt_than}(X, \text{Numbers}) &\text{ contains } \text{get_grt_than}(X, \text{inv}(\text{Numbers})). \\ \text{get_grt_than}(X, \{N \mid _ \}) &\text{ contains if } (N > X) \text{ then } \{N\}. \\ \text{inv}(\{N \mid _ \}) &\text{ contains } \{-N\}. \end{aligned}$$

The SEL code points out clearly that the execution of *get_abs_grt_than/2* leads to two independent flows of control. Therefore, the SAL code resulting from the compilation of this SAL code contains instructions that, when implemented on a MIMD architecture, permit a real parallel execution. The optimized SAL code is given in Figure 24.

10 The SL environment

The SL compiler project has been structured to build an open environment where programmers may execute and test their SL programs on different architectures. This is facilitated by implementing a system that can download SAM code to a number of different hosts on a network. The environment has been structured as a user interface (UI) program that can be connected with the different implementations of SAM running on remote machines (these include SUN4, CM2 and Transputer). The UI exchanges messages with the selected machine in such a way to pass programs, execute or trace them and retrieve results.

The structure of the programming environment is shown in Figure 25. The user interacts with the system through a **command manager** that distinguishes internal commands from SL queries. Internal commands

```

get_abs_grt_than/3:
  try_sub_and - 1           ; this is the only equation
  allocate 5                ; allocate five registers in the environment
  get_variable Y5 A1        ; get the 1st argument in Y5
  get_variable Y4 A2        ; get the 2nd argument in Y4
  get_variable Y3 A3        ; get the reference to the result in Y5
  do_me_then secondPart    ; the two parts can be executed concurrently
  put_value Y5 A1           ; put the value of Y5 in arg. reg. A1
  put_value Y4 A2           ; put the value of Y4 in arg. reg. A2
  put_variable Y2 A3        ; put a reference to Y2 in A3
  call get_grt_than/3       ; call the function
  union Y2 Y3               ; Y3 union Y2 → Y3
secondPart:
  last_do                   ; no further potential process parallel executions
  put_value Y4 A1           ; put the value of Y4 in arg. reg. A1
  put_variable Y1 A2        ; put a reference to Y1 in A2
  call inv/2                ; call the function inv
  put_value Y5 A1           ; put the value of Y5 in arg. reg. A1
  put_value Y1 A2           ; put the value of Y1 in arg. reg. A2
  put_variable Y2 A3        ; put a reference to Y2 in A3
  call get_grt_than/3       ; call the function get_grt_than
  union Y2 Y3               ; Y3 union Y2 → Y3
  deallocate                ; deallocate the environment
  proceed_sub               ; return

get_grt_than/3:
  try_sub_and - 1           ; this is the only equation
  get_variable X9 A1        ; get the 1st argument in register X9
  get_set X8 A2             ; get the 2nd argument (a set) in reg. X8
  get_variable X7 A3        ; get the address of result in reg. X7
  map_over X8 X6 X5 end2    ; map over X8 (copied in X6) using X5
start2:
  put_value X5 A1           ; put the value of X5 in A1
  put_value X9 A2           ; put the value of X9 in A2
  put_variable X4 A3        ; put a reference to X4 in A3
  operation > /2            ; call predefined op >
  fjump X4 j1               ; if X4 goto j1
  insert X5 X7              ; X7 union X5 → X7
j1:
  end_fjump X4              ; end of the jump
  end_map_over X6 X5 start2 ; repeat until X6 is empty
end2:
  proceed_sub               ; return

inv/2:
  try_sub_and - 1           ; this is the only equation
  get_set X7 A1             ; get the 1st argument (a set) in reg. X7
  get_variable X6 A2        ; get the addr. of the res. in reg. X6
  map_over X7 X5 X4 end3    ; map over X7 (copied in X5) using X4
start3:
  put_value X4 A1           ; put the value of X4 in A1
  put_variable X3 A2        ; put a reference to X3 in A2
  operation - /2            ; call predefined op -
  insert X3 X6              ; X6 union X3 → X6
  end_map_over X5 X4 start3 ; repeat until X5 is empty
end3:
  proceed_sub               ; return

```

Figure 24: Optimized SAL code for *get_abs_grt_than/2*.

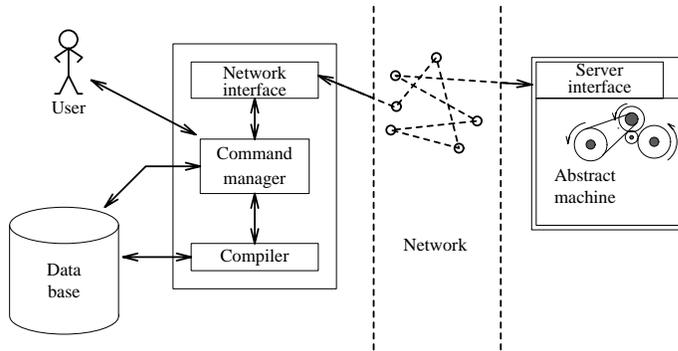


Figure 25: SL environment system scheme

perform SL database analysis, selection of the machine on which the programs will be executed and some other facilities that allow manipulating the interactive environment. The user interface invokes the SL compiler to **consult** SL programs (an SL program is translated into an object code when it is **consulted**).

Figure 26 shows a sample session using the interactive SL environment. The user types in commands on the command line in the middle of the window, while the lower part echoes the command showing its output. The upper part shows the previous commands typed in by the user, who can use it as a history by which to repeat a command, clicking on it with the mouse. When the user types in a query, each object is linked to produce SAM executable code that will run on the selected machine to produce the result. Executable codes, results and tracing values are transferred from the UI to remote the remote SAM or vice versa using a simple character oriented protocol on TCP sockets. This transfer mechanism might be modified in such a way as to reduce transfer time and make transmissions more reliable. However, this goes beyond the present scope.

The system has been implemented using Sicstus Prolog for the compilation from SL to the SAM assembler, and GNU g++ for the executor. The implementation on the CM2 uses C*, and the one on the Transputer uses the Meiko CStools. The SL environment is based on Motif 1.2. A Tk/Tcl-based interface is also under development. The package is available by anonymous ftp from the site <ftp.lii.unim.it>.

11 Conclusions

This paper presents a framework for the implementation of SL, a set-based logic language. The choice of the set as the fundamental data structure of the language is motivated from the intention to exploit parallelism inherent in it, as regards both process and data parallelism. Moreover, sets are very useful as high-level representations of complex data structures, in particular suited for fast prototyping, and there are problems in various fields that can be represented as relations between sets. Unlike many logic languages, such as Prolog, that embody sets as a patch on top of their basic data structures, SL 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.

One criticism of set-based logic languages is that they introduce overwhelming time and space complexity because fully general purpose set matching algorithms tend to run in times exponential in the size of sets involved. However, introducing a number of syntactic restrictions in SL allows us to profit from



```
File Machine Editors Colors
machine.
machine=dist.
machine.
consult('grandpa.sl').
get_grandpa({father('Tom','Sally'),father('John','Phil'),father('Phil','Bill')}).

Command:
|

ines      inactive
harrison  inactive
kevin     inactive
dist      inactive

| ?- machine = dist.
yes

| ?- machine.
ines      inactive
harrison  inactive
kevin     inactive
dist      active

| ?- consult('grandpa.sl').
yes

| ?- get_grandpa({father('Tom','Sally'),father('John','Phil'),father('Phil','Bi
John
yes

| ?-
```

Figure 26: An example of a session on the SL environment system

the characteristics of restricted associative–commutative matching derived from its predecessor SEL (see also Schmueli et al. (1988) for an analogous approach in LDL). In this way, rather than implementing unification or even general ac-matching which is in fact NP-complete (Benanav et al. 1985), the execution of SL programs is made relatively efficient. In practice, these restrictions do not inhibit programming style and add great efficiency gains. For instance, one reflection is the restriction of the *union* operator to the top most level, and emphasis on the use of the $\{term \mid set\}$ construct. Similarly, SL emphasizes iteration over members of a set rather than the power set enumeration for iterating subset – meaning a proliferation of linear rather than exponential complexity in algorithms.

SL is closely related to SEL, but the principle differences between the two are as follows.

- SEL contains equational assertions and subset assertions, whereas SL possesses only equational assertions; however, SL allows the equivalent of the SEL subset assertions by means of an expanded ensemble of set constructs directly related to the traditional mathematical set notations – thus providing a simpler and more intuitive top level view.
- The compiler for SL has a more advanced matching algorithm which optimizes the production of code for the matching process (prior to imposing extensions of the classical peep-hole optimization techniques to it), making use of techniques analogous to (but in substance different from) those described by Schmueli et al. (1988) for enhancing efficiency.
- The compiler for SL searches for and demarcates regions of code where opportunities for process parallelism or data parallelism exist.
- SL has been implemented in a networked environment that permits use with a number of connected remote hosts.

Yet, SL is so closely related to SEL that in fact, in the initial phase of compilation, SL is directly translated into SEL. Furthermore, in the subsequent compilation phase, this SEL code is translated to pseudocode for the abstract machine SAM, a sister of the SEL-WAM (Nair 1988) abstract machine from which it inherits some of its implementation strategies. Both machines belong to the WAM family (Ait-Kaci 1990) since their general structure resembles closely that of the WAM used to implement Prolog and other logic languages in a target-machine-independent way. However, neither SAM nor the SEL-WAM needs full unification capabilities, and each has special support for set data structures.

Thus, in compiling from SL to target SAM, the process has been divided into two parts: the compilation from the source language to the abstract machine assembler (SAL), and the subsequent implementation of the abstract machine on a real architecture.

The compilation is divided into a sequence of phases, which lead from a source code written in SL to an executable pseudocode for an appropriate abstract machine (SAM). Taking this approach, the portability of SL programs is therefore due to the implementation of the abstract machine: the compiler transforms the source code into a sequence of instructions of the (imperative) assembly language of SAM, independently of how the instruction set is implemented. Free from architectural constraints, the compiler can thus first devote attention to the efficiency and reliability of the executable code at a higher level, identifying and flagging possibilities for both process and data parallelism aspects inherent in SL. For instance, an implementation of SAM 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. In any case, this is completely transparent to the compiler, which generates code that can be executed as ‘quasi-efficiently’ as possible across a large number of architectures.

For this purpose, a number of set-based matching instructions were incorporated into the SAM instruction set (even beyond those in the SEL-WAM). Moreover, the SL compiler incorporates a special set of strategies for optimizing the matching algorithm (IOA), by categorizing and recognizing the interdependence of subpattern matching. Subsequent to the initial SAL code generation, several other peep-hole optimization techniques have been applied so as to save time and space in a machine independent way, even before the final machine-dependent optimizations are made in the subsequent implementation of SAM on a real computer architecture.

Although only the compilation has been discussed here, the abstract machine, SAM, has been implemented not only on sequential computers (Sun4 and HP RISC workstations), but also on the CM2 and on the Transputer.

Future research will focus on defining a more efficient parallel implementation on parallel architectures and in integrating further feature in the SL languages, such as arrays, fixed point calculation and so on. One further interesting point of commonality between SL and SEL is the possibility of performing the restricted ac-matching upon which both are based using either depth-first or breadth-first matching strategies, since it has been shown that both are sound and complete (Jayaraman 1992, Jayaraman and Plaisted 1987). This has a further implication for parallelism beyond those already introduced: the choice of breadth-first or depth-first matching strategy might be made based upon the nature of the target architecture (e.g. SIMD vs. MIMD machine). A CM5 implementation is under way, and since the CM5 is a multi-SIMD machine, it will be possible to exploit both process parallelism and data parallelism simultaneously there.

Acknowledgements

The authors wish to thank Bharat Jayaraman who initiated this research trend and has offered fruitful comments wherever requested, and Giuseppe Marino for original ideas on set-based parallelism. Also, thanks go to Mark Foy.

References

- Ait-Kaci, H. (1990) *The WAM: A (Real) Tutorial*. Digital - Paris Research Laboratory.
- Beck, K. and Johnson, R. (1994) Patterns generate architecture, in *Proceedings of the ECOOP'94*, Bologna, Italia, July 1994. Lecture Notes in Computer Science 821, Springer-Verlag, New York, pp.139–48
- Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, O. and Tsur, S. (1987) Sets and negation in a logic database language (LDL1), in *Proceedings of 6th ACM Principles of Database Systems*, pp.234–56,.
- Benanav, D., Kapur, D. and Narendran, P. (1985) On the complexity of matching problems, in *Proceedings of the International Conference on Rewriting Techniques and Applications*, May 1985, pp.417–29.

- Bennett, K.H. (1994) An introduction to software maintenance, in *Proceedings of the Summer School of Engineering of Existing Software*, Capitolo-Monopoli, Italia, June 1994, Giuseppe Laterza Editore, [?place], pp.31–65.
- Biggerstaff, T.J., Mitbender, B.G. and Webster, D.E. (1994) Program understanding and the concept assignment problem. *Communications of the ACM*, ??, 72–82,
- Bird, B. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice Hall International, Hemel Hempstead, UK.
- Boriello, A. (1987) Riscs and ciscs for prolog: A case study, in *Proceedings of ASPLOS II, IEEE Computer Society*, Palo Alto, CA.
- Burn, G.L. Peyton Jones, S.L. and Robson, J.D. (1988) The spineless G-Machine, in *Proceedings of the ACM Conference on LISP and Functional Programming*, Snowbird, UK, pp.244–58 [?? publisher]
- Church, A. (1941) *The Calculi of Lambda Conversion*. [??publisher and place]
- Cimitile, A. (1994) *Proceedings of the Summer School of Engineering of Existing Software*,Capitolo-Monopoli, Italia. Giuseppe Laterza Editore,[?place]
- Debray, S.K. Lin, N.W. and Hermenegildo, M. (1990) Task granularity analysis in logic programs, in *ACM SIGPLAN '90*.
- Delly, W.J. Wills, D.S. and Lethin, R. (1991) Mechanism for parallel computers, in *Proceedings of the NATO ASI School on Parallel Computing on Distributed Memory Multiprocessors*, Ankara, Turkey, (ed. ?? Ozgunter and ?? Ercal), Springer-Verlag, New York, pp.3–25.
- Dovier, A., Omodeo, E.G., Pontelli, E. and Rossi, G. (1991) Log: a logic programming language with finite sets, in *Proceedings of the 8th International Conference [?of what]*, MIT Press, Cambridge, MA.
- Freudenberger, S., Schwartz, J. and Sharir, M. (1983) Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, **5**(1), 26–45.
- Harrison, R. (1992) Reusable software components through data abstraction, in *Proceedings of the WISR '92 Fifth Annual Workshop on Software Reuse*, Palo Alto, CA, (ed. L. Latour, S. Philbrick and M. Stevens). [[publisher etc?]]
- Hillis, W.D. and Trucker, L.W. (1993) The CM-5 connection machine: a scalable supercomputer. *Communications of the ACM*, **36**(11), 30–40.
- Jayaraman, B. (1990) Broader forms of logic programming. Technical report TR90-013, State University of New York at Buffalo, May 1990.
- Jayaraman, B. (1991) The SURE programming framework. Technical report TR91-011, State University of New York at Buffalo, July 1991.
- Jayaraman, B. (1992) Implementation of subset equational programs. *Journal of Logic Programming*, **13**(3), 299–324.

- Jayaraman, B. and Nair, A. (1988) Subset-logic programming: application and implementation, in *5th International Logic Programming Conference*, Seattle, Washington, August 1988.
- Jayaraman, B. and Plaisted, D.A. (1987) Functional programming with sets, in *3rd International Conference on Functional Programming Languages and Computer Architecture*, Portland, OR, pp.194–210.
- Jayaraman, B. and Plaisted, D.A. (1988) Semantics of Subset Logic Programming. Technical report, University of North Carolina at Chapel Hill, July 1988.
- Johnson, R. (1992) Documenting frameworks with patterns, in *Proceedings of OOPSLA '92*, Vancouver BC, October 1992, pp.63–76.
- Johnsson, T. (1987) Compiling lazy functional languages. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg.
- Kuper, G.M. (1987) Logic programming with sets, in *Proceedings of 6th ACM Principles of Database Systems*, pp. 10–14.
- Kuper, G.M. (1988) On the expressive power of logic programming languages with sets, in *Proceedings of 7th ACM Principles of Database Systems*, pp.10–14.
- Kurosawa, K. et al. (1988) Instruction architecture for a high performance integrated Prolog processor IPP, in *5th International Conference and Symposium on Logic Programming*, University of Washington, MIT Press, Cambridge, MA, pp.1506–30.
- McCarthy, J. (1981) *History of LISP*. R.L. Wexelblat, New York.
- Nair, A. (1988) Compilation of subset-logic programs. Master's Thesis, University of North Carolina at Chapel Hill.
- Patt, Y.N. and Chen, A. (1989) A performance comparison between the VLSI-PLM and the MC69020 as prolog processor. Technical report, Computer Science Division, University of California, Berkeley.
- Peyton Jones, S.L. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall International, Hemel Hempstead, UK.
- Peyton Jones, S.L. (1988) Flic – a functional language intermediate code. *SIGPLAN Notices*, **23**(8).[??page nos]
- Schonberg, E., Dewar, R., Grand, A., Liu, S. and Schwartz, J. (1979) Programming by refinement as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems*, **1**(1), 27–49.
- Schonberg, E., Schwartz, J. and Sharir, M. (1981) An automatic technique for selection of data representations in SETL programs. *ACM Transaction on Programming Languages and Systems*, **3**(2), 126–43.
- Shapiro, E.Y. (1989) The family of concurrent logic programming languages. *ACM Computing Surveys*, **21**(3), 413–510.

- Shmueli, O. Tsur, S. and Zaniolo, C. (1988) Rewriting of rules containing set terms in a logic language (LDL). in *Proceedings of 7th ACM Principles of Database Systems*, pp.15–28.
- Sterling, L. and Shapiro, E.Y. (1986) *The Art of Prolog*. MIT Press, Cambridge, MA.
- Succi, G. (1992) Exploiting implicit parallelism of logic languages with the SAM. in *Proceedings of the 1992 ACM Symposium on Applied Computing*, Kansas City.
- Succi, G. (1993) La compilazione di linguaggi dichiarativi basati su insiemi per architetture parallele. PhD Thesis, DIST – Università di Genova.
- Succi, G. Marino, G. and Colla, G. (1993) The CM2 as an active memory to implement declarative languages. *Journal of Programming Languages Design*, **1**(1), 127–42.
- Succi, G. and Marino, J. (1991) A new abstract machine for subset equational languages. Technical report, DIST – Università di Genova.
- Turner, D.A. (1979) A new implementation technique for applicative languages. *Software Practice and Experience*, **9**(9), 31–49.
- Turner, D.A. (1985) Miranda: a non-strict functional language with polymorphic types, in *Proceedings of the IFIP International Conference on Functional Languages and Computer Architectures*, Nancy, France.
- Van Roy, P. (1984) A prolog compiler for the PLM. Master’s Thesis, University of California ,Berkeley.
- van Vliet, H. (1993) *Software Engineering, Principle and Practice*. Wiley, Chichester.
- Warren, D.H.D. (1983) An abstract prolog instruction set. Technical Note 309, SRI International.
- Wirth, N. (1985) *Programming in Modula-2*. Springer-Verlag, New York.
- Wray, S.C. and Fairbairn, J. (1989) Non strict languages – programming and implementation. *Computer Journal*, **32**(2), 142–51.
- Yau, S.S. and Liu, S. (1987) Some approaches to logical ripple effect analysis. Technical Report, SERC, USA.