Fasc. I - II Volume XLIII 1988

LE MATEMATICHE

VOLUME XLIII
1988

Proceedings of the 1st Catania Workshop on:
"Artificial Intelligence" Catania, may 15-19, 1989
Edited by Angelo Marcello Anile and Alfredo Ferro

DIPARTIMENTO DI MATEMATICA DELL'UNIVERSITÀ DI CATANIA

«LE MATEMATICHE»
Vol. XLIII (1988) - Fasc. I, pp. 79-98

META-INTERPRETING SETL

D. ALIFFI - D. MONTANARI - E.G. OMODEO - M. PELLEGRINI (Bologna)

This paper describes a SETL interpreter written in SETL. This module may be reused as a basis to build debuggers, type checkers, symbolic executers, tracers, and many other general purpose programming tools. Other more advanced uses include experimenting with altered semantics for SETL and building interpreters for multi-paradigm languages, as in the SetLog project, which aims at constructing a language integrating logic programming and set-oriented programming.

1. Introduction

SETL ([8]) is a very-high-level language developed at the New York University in the Seventies. It is a classical imperative language whose main data types are hereditarily finite sets, as well as functions defined on them, and ordered tuples. These data types, combined with a notation very close to the usual mathematical one, give the language a great expressivity. Recently SETL has been the subject of the Esprit project SED([1]), whose aim was to build a programming environment which could make SETL a useful language for application

([1]) Esprit Project No. 1227.

prototyping. Among others, SED has led to the implementation of a fairly complete interactive and/or batch SETL meta-interpreter, which is described in this paper.

The use of meta-interpreters has a long tradition in LISP and Prolog programming. It could be equally useful to SETL programmers, but the implementation of a SETL meta-interpreter is a considerably harder task. A minimum requirement is that the meta-interpreter ought to be able to prevent the occurrence of *any* kind of run-time error. More generally, the meta-interpreter should provide flexible means of controlling the execution. Some exploitations of a meta-interpreter are:

- the collection of global information about execution. In the present implementation, a dynamic type-finding facility is already present, allowing the recording of the types taken by a variable during the execution, and the determination of the (complex) type of the variable. Statistics on the operations involving complex data objects can guide a heuristic choice of an optimal internal representation;
- the amalgamation of procedural programming with other programming paradigms. For example, the SetLog Project ([7]) aims at an integration of set-oriented and logic-oriented programming, and uses the SETL meta-interpreter as one of the main building blocks (see section 6);
- as a starting point for the development of tools for the static analysis of SETL programs, like type checkers, data flow analyzers, and program transformers;
- the experimentation with altered semantics or with new types or representations. During the development of the meta-interpreter, many semantic obscurities of the original SETL specification have been recognized and solved;
- as a basic to support higher level debugging facilities for SETL.

Our meta-interpreter is able to treat input/output statements, assignments and from instructions, structured control constructs, and

procedure calls, where procedures are either «hardwired» into the meta-interpreter or created at run time as «ephemeral» lambda-expressions.

Besides the goals mentioned above, within the SED project we had the following aims:

- to clarify the semantics of SETL, in particular for what concerns the run-time type model and the conception of an abstract machine;
- to have a fully blown scheme on which to base the construction of several meta-interpreters;
- to have a script command language homogeneous with the implementation language, to be exploited in testing and demonstrating SETL libraries.

2. Meta-Interpreter Architecture

The meta-interpreter has three main units, which:

- parse the input and manipulate the resulting abstract syntax trees (ASTs);
- build the symbol tables taking into account type, variable, and procedure declarations;
- interpret the ASTs, and modify the symbol tables accordingly.

The detailed structure of these components and their dependencies are shown in the appendix.

The meta-interpreter works on ASTs, represented by nested SETL tuples. These are fed to the meta-interpretation modules by the *parser* (written in SETL). The parser itself is a widely reusable library, which may be adapted to parse input following any operator precedence grammar (actually, it can handle operators with double precedences as in Prolog [10]).

The treatment of control structures is still slightly incomplete; at

present, in fact, only the structured constructs `if`, `while` and `until` can be meta-interpreted, whereas no provisions have been made concerning `forall`, `quit`, `continue` and `goto`. Control constructs are not expressed in their ordinary form, but are supplied to the meta-interpreter in a conventional syntax, closer to the structure of an AST than to the concrete syntax of SETL.

The *symbol table handler* holds two tables: one contains the definitions of programmer-defined types; the other one contains the variable and procedure declarations. The *main program* initializes the symbol tables, and calls the *declaration interpreter* to accept declarations from the user. During the execution phase, the *evaluator/interpreter* calls the *symbol tables handler* to check the consistency between a value and the declared type of a variable, before the latter is assigned the value.

The actual interpretation of ASTs supplied by the parser is performed using the *pattern matching machine* to recognize the instructions. Each internal instruction is then processed by the *language extender* and the *type disambiguator* to reduce it to a simpler form, accepted by the *meta-SETL machine*. The *control-flow graph constructor* builds a control-flow graph of a program (or an ephemeral procedure); the *control-flow graph navigator* traverses this graph during the execution.

A few basic components of the meta-interpreter deserve a brief description. Since an accurate description of a non-redundant abstract SETL machine is hard to find in the literature – if it exists at all –, we have put particular care in the design of a

- *Meta-SETL machine*

  This module emulates a small kernel of fundamental SETL operations (union, intersection etc.) and assignment instructions which, *with the addition of control alone,* are adequate to support the definition of all the other SETL constructs. As a rule, the operations implemented at this level are *not* overloaded, except `=`, `type`, and some minor ones.

A rather unusual feature of SETL is the heavy overloading of its operators. This justifies the existence of a

- *Types disambiguator*

  This module receives as input the name of a SETL operation along with its operands, which are already fully evaluated, but under anomalous circumstances may require a type coercion. After resolving the overloading of the operator and performing type coercion whenever necessary, this module triggers the appropriate action of the meta-SETL machine. It also determines the type of the result. Only nullary, unary, and binary operations are implemented at this level, including assignment operations. Control instructions are not present at this level, and only gross types are taken into account. These are:

```
om, nuls,     (singleton scalar types);
atom, boolean, integer, real, string,
                    (remaining scalars);
tuple, smap, mmap, set, set3,
                      (aggregate types);
```

  plus an error mark, named `error`, and a wild-card, named `general`, which stands for any gross type. `set3` is a representation for sets which makes it easier to extend the semantics of operations typical of (single- or multi-valued) maps to sets of any kind. Inclusions among gross types are as follows:

```
om, atom, boolean, integer, string < general;
integer ≪ real;
tuple ≪ smap < mmap < set ≪ set3 ≪ general;
nuls < smap; string ≪ smap;
```

where inclusion denoted by ≪, unlike others, require type coercion. Thus, to translate a smap to an mmap or to a set, no coercion is required, whereas one coercion is needed to translate a tuple to an mmap, and two coercions would be necessary to translate a tuple to

a `set3` (if this translation were ever needed). We add direct inclusion `set < general, tuple < general`, to bypass type coercions in these two cases.

This is just one aspect – not very problematic indeed, since it refers to *gross* types exclusively – of the run-time type model of SETL. Other more intricate issues concerning types in SETL are examined later in this paper.

We stress again that our meta-interpreter is mainly meant to be a *scheme* to be followed in the implementation of fancier (and more useful) meta-interpreters. Nonetheless, even in its present form, it has «hooks» enabling extensions and modifications of the language surface. The following component plays an essential rôle in this respect.

- *The language extender* The language extender is a layer placed between the evaluator/interpreter and the types disambiguator, in order to increase the number of language constructs, or to strengthen the semantics of some of the constructs in the kernel. Examples of language enrichments typically supported by this layer are:
  - `notin, range`, and other constructs that can be expressed as simple combinations of other constructs provided by the kernel (e.g. `range(f)` stands for `f[domain f]`); the alternated relators <=, >=, / =, are here too;
  - (gross) type testing operations, such as `is_atom, is_smap`, etc.; enhanced implementations of the operations `with, f(x)`.

  The connectives `and` and `or` are not implemented at this level, nor at the meta-SETL machine level. In fact, since the evaluation of these constructs is optimized in the SETL semantics, they must reside at the level of all control constructs.

Another opening in the existing meta-interpreter is the possibility of hardwiring any new construct implemented as a SETL procedure (e.g. a `prime` predicate based on a fast primality test) into it. In

addition to these procedures, that constitute the initial endowment of the meta-interpreter, the end-user can define his own procedures «on the fly», as we will explain in the section on ephemeral procedures.

3. Manipulating abstract syntax trees in SETL

In this section we describe some of the components of the meta-interpreter for their own sake, as reusable libraries and modules. From this perspective, it makes sense to hint at pieces of software that are still in our mind or that have been developed independently of the meta-interpreter, to make the assembly of the various pieces in several different contexts easier.

- *The all-purpose parser*

  The parse generates ASTs from strings. Trees are *raw* in the sense that they are encoded by «nested tuples», and identifiers and keywords inside them are kept in their native string form.

The programmable parser has several features:

- the concrete grammar accepted by the parser contains virtually no «syntactic sugar» and its variety of constructs should faithfully reflect the taxonomy of the ASTs;
- concessions to common practice can be made, to avoid cumbersome amounts of parenthese. These are: infix, prefix and postfix use of some operators, and the use of priorities;
- for efficiency reasons, the syntactic analyzer is, by and large, an operator precedence parser; but nonetheless it is able to support grammars richer than:
  * Prolog's grammar, including its widespread list notation (which will be internally brought into the usual «vineyard tree» representation, but without fully enforcing the identification between lists and nested uses of a binary `cons` operator);

* the grammar of the expressions of SETL, including constructs for restricted quantifications and set- and tuple- formers of the usual richness, but *not* including constructs, such as the `if...then...else` expressions, that are too remote from an operator-precedence parsing approach;
* the language of first-order predicate logic, including quantification constructs, extended with constructs typical of the theory of sets and classes, among which abstraction terms of various kinds;
* a minimal grammar into which Mentor's trees could be unparsed in the most «universal» and trivial manner. Essentially, this new requirement only imposes deviating from Prolog's ordinary list notation to accept head-less lists of the form $[|H]$).

- *The synthesizer*

  The synthesizer constructs a raw AST whose subtrees are given and whose root has specified characteristics.

- *The evaluators*

  The evaluators carry out computations of many different kinds, driven by systematic exploration(2) of the raw AST. The series of conceivable evaluators is open-ended. Already available are:

  – An `unparser` that converts an AST into a string concretely representing the same expression.

  – A meta–evaluator of trees that represent ground SETL terms involving, in addition to integer, boolean, and string constants, only the constructs $[-,\ldots,-]$, $\{-,\ldots,-\}$.

  When combined with a meta-parser which is an instance of the all-purpose parser mentioned above, this evaluator provides a «meta-read», which is preferable to ordinary `read` of SETL, because it can never cause a run-time error.

(2) «Systematic exploration» actually means structural recursion based upon the analyzer.

- An evaluator which constructs an internal table of *mode declarations* from a forest of syntax trees that represent such declarations expressed in a simplified DRSL (Data Representation SubLanguage) of SETL, called *mini*-DRSL. Variable declarations are also evaluated, and the results of the evaluation are stored in a symbol table for variables.

- A meta-interpreter for straight SETL code consisting of assignments and input/output statements. This has been combined with the mini-DRSL evaluator just described, to form an executable specification of the run-time type model of SETL enhanced to deal with recursive ~~type~~ types on the one hand and to perform «wild» type coercions (e.g. conversions of tuples and strings into smaps) on the other hand.

  The meta-interpreter has also been «married» to a control flow graph constructor which is able to treat `if`, `while`, and `until` constructs. This marriage gave birth to a meta-interpreter for a restricted, but functionally complete, version of SETL, which emulates well-structured forms of control.

- The `pattern-matching machine` is a particularly useful evaluator, which performs high-level pattern matching on raw ASTs, taking a base of tree pattern definitions into account. Gives a yes/no answer to reflect success or failure. The response can be used to reject expressions that would otherwise be acceptable on purely syntactic grounds.

- *The analyzer*

  This module performs low-level pattern matching on raw ASTs, based on a rather rich taxonomy of nodes. In case of success, supplies the «neighborhood» of the analyzed node (e.g. associated operator, arity, sons).

- *The AST transformer*[3]

  Syntactic transformations of AST take place very frequently in programs, and most of the time they take the form of

[3] This module has yet to be designed.

quite straightforward structural recursions. Performing one such transformation amounts to «evaluating» a tree to produce another tree, by systematic calls to the tree synthesizer. A language in which simple syntactic transformations can be succinctly expressed in the form of «rewrite rules», and an interpreter for such language must be designed, on top of both the synthesizer and the analyzer (calls to the analyzer drive the construction of the transformed tree, while calls to the synthesizer perform the construction). The transformation language ought to be declarative in style, similar to the language of the pattern-matching machine on one side, and inspired by Prolog on the other side. Inspiration for the conception of this module may come from the RAPTS system ([6]), but we have something much simpler than RAPTS in mind here. Infact, RAPTS works on a much less naive representation of AST, and carries out higher-level, semantics-sensitive transformations.

It is remarkable that a system of the complexity of a SETL meta-interpreter can be based on a simple-minded representation of ASTs like the raw trees hinted at above. Efficiency concerns, or subtler implementations than an interpreter for an imperative language, may require more sophisticated representations of abstract syntax trees. To provide support for a more general setting, SETL libraries have been implemented to perform the following tasks:

- *The equalities detector*

  Determines equal sub-expression given in the form of raw abstract syntax tree.

- *Well-done abstract syntax tree generator*

  Obtains from the raw representation of an abstract syntax tree a more explicit representation where each syntax node is encoded as a SETL atom, and suitable maps defined on such nodes take the place of «nesting». This representation is more convenient in cases when one has to perform complex «evaluations» of syntax trees such as unification or data-flow analysis of some kind.

  By exploiting the equalities detector, it is possible to represent

equal expressions by the same atom, so that the abstract syntax tree becomes, in fact, a directed acyclic graph.

The Prolog interpreter implemented as part of SetLog exploits an even more refined representation of ASTs. However, this representation is too much *ad hoc* and, as a consequence, less reusable than the rest.

4. Modeling SETL types in SETL

The work described in this section is aimed at providing answers to the following questions:

- What is a recursive type in SETL?
- What is the type of a SETL value?
- When is a SETL value compatible with a type defined in the Data Representation Sub Language (DRSL), extended with alternated and recursive type definition [11, 3]?

A SETL program has been implemented in an effort to come out with satisfactory answers to these questions, and to experiment with an internal SETL representation of DRSL statements (both type declarations and variable declarations). This representation has been designed independently to be the support, inside SETL meta-interpreters, of a well-understood run-time type model.

Partial answers indeed came from the implementation of this program. The answers given are, in fact, an adaptation to the multi-sorted universe of SETL values of the notion of *rank*, classically referring to the framework of von Neumann's type-free hierarchy of sets [5].

The question «what is the type of a SETL value» still remains partially open due to two reasons:

- the ambiguous status of maps, which are sets subject to certain restrictions, where the restrictions are hard to frame in type

theory;

- the interference of pragmatic concerns (efficiency of a type-finder for SETL) with the hope of obtaining a very sound – although reasonably simple – mathematical answer.

To see other problematic aspects of the type notion, notice that the «history» of how a value has been calculated may affect his type or – better to say – the type of a target variable. For example, in

$$x_1 := \{[1,2],[3,4],5,6\} \texttt{ less } [1,2] \texttt{ less } 5 \texttt{ less } 6$$

$$x_2 := \{[1,2],[3,4],5,6\} \texttt{ less } [1,2] \texttt{ less } 5 \texttt{ less } 6$$

x_1 and x_2 are assigned the same value, but it makes sense to regard x_1 as a set and x_2 as a single valued map. Even the way a computation is *specified* may interfere with type determination. For instance, the instructions:

$x := \{1\};$

`(for` i `in` $[1...0]$`)`$x := x$ `with` x`; end;`

suggest that x has the recursive type

`mode` *type_x:* `set (INTEGER) | set` *(type_x);*

while the «equivalent» instruction

$$x := \{1\};$$

suggest that x has the simpler type

`mode` *type_x:* `set (INTEGER)`

The following subsections briefly describe some details of the implementation. The first one provides the formal description of the declarations language actually used, while the following two describe which type coercions and consistency checks on type declarations are performed.

4.1. Declarations language.

Type definitions, variable and procedure declarations are supplied during the first phase of the meta-interpreter, and are expressed in a language similar but not identical to DRSL. This language is called *mini-DRSL*, because in some respects it is more restricted than ordinary DRSL, although it is richer in that it provides alternated and recursive types.

In extended Backus-Naur formalism, mini-DRSL can be specified as follows:

```
⟨Decl⟩                  ::= ⟨mode_decl⟩|⟨var_decl⟩|⟨proc_decl⟩
!                         |⟨base_decl⟩
⟨mode_decl⟩             ::= mode ⟨mode_id⟩':' ⟨alternands_list⟩
⟨var_decl⟩              ::= ⟨progr_var⟩':' ⟨alternand⟩
⟨proc_decl⟩             ::= ⟨progr_id⟩':'proc ⟨alternands_list_tuple⟩
                          |⟨proc_id⟩':' ⟨alternands_list_tuple⟩ proc
                          |⟨alternands_list⟩
!⟨base_decl⟩            ::= base ⟨base_id⟩':' ⟨alternand⟩
 ⟨alternand_list_tuple⟩ ::= ⟨alternand_list⟩
                          |'['⟨alternands_list_list⟩']'
 ⟨alternands_list_list⟩ ::= ⟨alternand_list⟩
                          |⟨alternands_list_list⟩',' ⟨alternands_list⟩
 ⟨alternands_list⟩      ::= ⟨alternand⟩
                          |⟨alternands_list⟩',' |⟨alternand⟩
 ⟨alternand⟩            ::= boolean | atom | integer | real | string | '*'
                          | set ⟨alternands_list⟩|tuple ⟨alternands_list⟩
                          | ⟨alternand⟩ smap ⟨alternands_list⟩
                          | ⟨alternand⟩ mmap ⟨alternands_list⟩
!                         | elmt ⟨base_id⟩
                          | ⟨mode_id⟩
 ⟨mode_id⟩              ::= ID
 ⟨progr_var⟩            ::= ID
 ⟨proc_id⟩              ::= ID
!⟨base_id⟩              ::= ID
```

where the production rules preceded by an exclamation mark are unimplemented yet, and '*' means 'general'.

4.2. Type coercions.

The SETL meta-interpreter is able to perform – under request – type coercions well beyond the capabilities of the standard interpreter. Three levels of type coercion are available. *Ordinary* type coercions transform integers to reals, smaps to mmaps, mmaps to sets. *Mild* type coercions extend to arbitrary sets the applicability of any operation normally applicable to maps: domain, range, lessf, $f(\cdots)$, $f[\cdots]$, $f\{\cdots\}$, etc.; in particular, smap operations become applicable to mmaps; *Wild* type coercions may transform strings and tuples into maps in order to apply certain operations. For instance, the block of instructions

$$[s := \texttt{'a'}, s(3) := 9]$$

will yield the smap value $\{[1, \texttt{'a'}], [3, 9]\}$ `for` s.

4.3. Symbol-tables handler.

After the translation of the declarations into a suitable internal form, the following global consistency checks are performed:

- every mode identifier occurring in the declaration of another mode is itseld defined;
- no mode identifier is defined more than once;
- recursive mode declarations are non-circular.

When a SETL value is about to be assigned to a variable or to a procedure parameter, a compatibility check is performed with the declarations. Note that type declarations may be recursive (which is not allowed in ordinary SETL); for instance

```
mode a : b smap a
mode b : a smap b,
```

which makes the compatibility checks non-trivial. Nonetheless any

such check always terminates. For example, given the declarations,

```
mode h_f: set h_f
mode h_f_i: set h_f_i | integer
```

the following (in) compatibilities will be recognized:

VALUE	*h_f*	*h_f_i*
$\{\#T\}$	NO	NO
$\{\{\},\{2,\{3\}\},3\}$	NO	YES
$\{\{\},\{\{\}\}\}$	YES	YES

This component of the meta-interpreter also has the ability to determine the type of any SETL value, without taking into consideration any type declaration.

For example, $\{0,\{\},\{[1,2]\},\{[1,2],[3,4]\}\}$ will be assigned the type

```
set(integer | (integer smap integer));
```

$\{\{[1,2,]\},\{[1,2],[1,a]\}\}$ will be assigned the type

```
set(integer mmap (integer | string))';
```

$\{\{[1,2],[3,4]\},\{[*,3]\}\}$ will be assigned the type

```
set set tuple integer.
```

An ability that the program is currently lacking, is the following: given e.g. the value $\{\{1\},\{\{3,1\},1\},2\}$, it should be able to define the type

```
mode h_f_i: set (h_f_i) | integer
```

and to assign it to the given value.

5. Ephemeral procedures.

Expressions and control flow graphs are treated by the meta-interpreter as special values that can be assigned to program

variables. After a control flow graph has been assigned to a variable, the latter becomes executable in a way similar to the way a procedure is invoked; for this reason, and since it is as «volatile» as any other SETL value, a control flow graph which has been stored in a program variable is called an «ephemeral procedure». An ephemeral procedure contains straight-code blocks connected by (conditional or unconditional) jumps, and a header formed by its formal parameters. In the current implementation, ephemeral procedures cannot be recursive.

Expressions of the form

$$\_\mathrm{L}\langle \mathrm{var}_1\rangle,\ \langle \mathrm{var}_2\rangle,\ldots,\langle \mathrm{var}_N\rangle\ \langle \mathrm{expr}\rangle$$

where each $\langle \mathrm{var}_I\rangle$ is an (optional) identifier, are called *lambda-expressions*. Such an expression denotes an ephemeral procedure, whose formal parameters are denoted by the $\langle \mathrm{var}_I\rangle$'s, and whose body is denoted by $\langle \mathrm{expr}\rangle$.

A lambda-expression can be translated into an ephemeral procedure by means of a `Compile` function.

Currently, an ephemeral procedure can only be executed after it has been compiled; eventually, it will be executable even in its expressions form.

6. Mixing Prolog and SETL in SetLog.

SetLog is the name of a Logic Programming system which has been implemented in SETL to combine Prolog with various meta-SETL interpreters. The goal is to originate a spectrum of bi-paradigm programming languages enhancing both SETL and Prolog, at least with respect to the expressive power of both. This extended Prolog ought to be compatible with C-Prolog (and perhaps with MU-Prolog too), because one first use of SetLog is the design of tools for the semantic analysis of SETL. Such tools have been designed in Typol [1], a language for semantics specifications, which is automatically translated into a version of C-or MU-Prolog. Once SetLog is sufficiently

developed to be a surrogate for C- or MU-Prolog in a Typol application, the entire world of reusable SETL softwere would be disclosed to the Typol programmer. Basically, we intend to produce various hybrids of SETL and Prolog, which in turn may be the basis for producing SETL-Typol hybrids able to support tools for the semantic analysis of SETL whose sophistication is beyond reach of today's Typol.

Furthermore, SetLog can be the framework in which to design extensions to the Prolog interpreter in directions that may facilitate modularity, meta-programming, knowledge representation, etc.. It is quite hard, in general, to modify Prolog interpreters written in C, and therefore a language for quick prototyping, such as SETL, looks very useful in the specifications of a new Prolog interpreter/compiler endowed with all desired features. Interpreting Prolog in Prolog is a fairly easy task because of the equivalence between data and programs, while it is certainly prohibitive to simulate SETL using Prolog since SETL is based on side effects. The decision to use SETL for interpreting both SETL and Prolog gives us a reasonable trade-off between ease of implementation of the two main components of the system.

The Prolog interpreter developed for SetLog consists of a *kernel*, constituted by those system predicated which are directly implemented inside the interpreter, and one or several Prolog libraries that extend this initial endowment of Prolog. For compatibility with C-Prolog, it is planned that the Prolog interpreter part of SetLog contains the SETL implementation of the following familiar Prolog primitives:

```
fail     true     !        halt        abort
see      seeing   seen     tell        telling
told     save/1   assert   clause/3
functor  arg      integer  nl
read     write    protect  break/exit
trace    notrace           =      ==
is       =:=      =\=      <  =<       >  >=
```

The primitive `save/2` and `dbreference` [9] are important for an effective use of the system and will be included in the above list. All other primitives and application oriented predicates are defined as Prolog procedures.

Prolog's `bagof` and `setof` and «lazy» variants of these will be implemented more easily and effectively than in ordinary Prolog, thanks to extensions with SETL constructs.

A Prolog interpreter in SETL is useful in advanced research projects based on logic programming for which an high level easily modifiable Prolog engine is needed.

The following is an example of use of the new primitives integrating Prolog and SETL:

$$edges := \{[g, h], [g, d], \ldots, [e, d]\}.$$

$$reach(V, V).$$
$$reach(V, W) :-$$
$$[V, U] \in edges,$$
$$reach(U, W).$$

We use the set `edges` to describe a directed graph; the reachability predicate `reach` uses a membership test on this set.

7. Conclusions.

One of the original goals of the Esprit project SED was to implement an enhanced Typol in SETL. Later on, it was decided to design specific semantic analysis tool for SETL, making direct use of SETL itself. However, most tools of this kind have been designed to work on quadruple code, while it was our belief that many advantages would ensue from working directly at the level of AST. This project has confirmed that working on AST in SETL is simpler than in most of the traditional programming languages. The only missing feature of the existing SETL is the unification mechanism. Prolog is the natural candidate to enhance SETL in this respect, and SetLog is an attempt to pursue this goal.

8. Acknowledgements.

Thanks are due to Ph.Facon, who participated in stimulating conversations about type models for SETL, F. Jean contributed to the characterization of sound recursive type declarations. The usefulness of meta-interpreting SETL emerged from discussions with J.P. Keller.

9. Appendix.

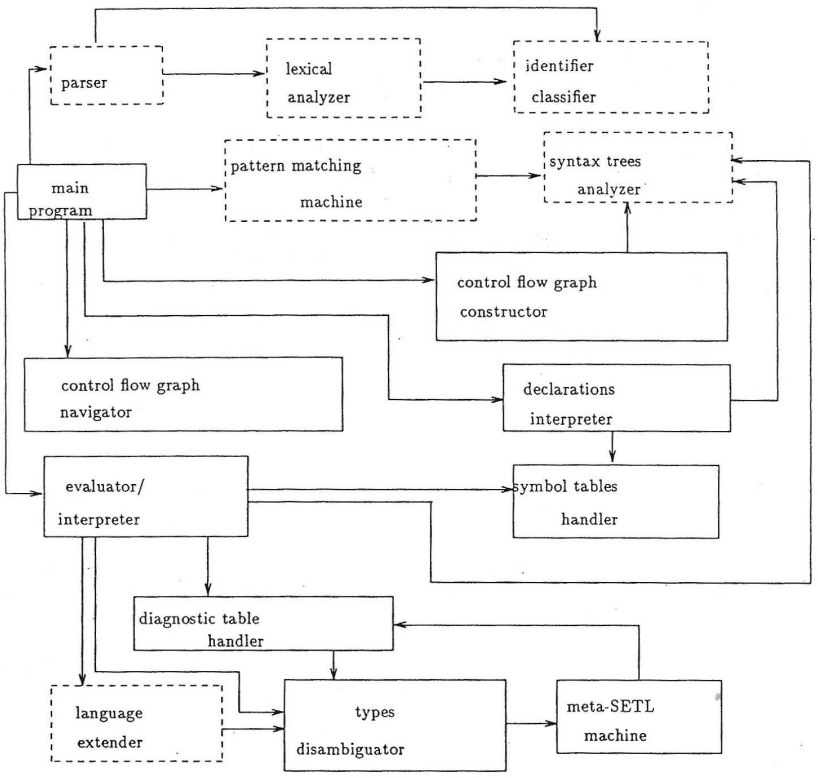

Figure 1 - Meta-interpreter dependency graph.

REFERENCES

[1] Despeyroux T., *Typol a Formalism to Implement Natural Semantics,* INRIA Tech. Rep. Draf Version, April 28, (1987).

[2] Donzeau-Gouge V., Kahn G., Lang B., Melese B., *Documents structure and modularity in Mentor,* Proc. ACM SIGSOFT-SIGPLAN Software Eng. Symp. Prac. Software Develop. Env., Apr. 1984 (141-148).

[3] Omodeo E., Facon Ph., *Preliminary considerations regarding type-checking and type-finding for SETL within SED,* SED internal report, 1987.

[4] Ghezzi C., Jazayeri M., *Programming Languages Concepts,* John Wiley & Sons, (1982).

[5] Manin Yu.I., *A Course in Mathematical Logic,* Springer Verlag, 1977.

[6] Paige R., *Transformational programming - applications to algorithms and systems,* Proc. 10th ACM Sym. on Principles of Programming Languages, 1983.

[7] Pellegrini M., Sepe R., *SetLog,a logic tool in a SETL environment,* submitted tch. rep., University of Rome, DIS, 1989.

[8] Schwartz J.T., Dewar R.B.K., Dubinsky E., Shonberg E., *Programming with Sets an introduction to SETL,* Springer Verlag, 1986.

[9] *C-prolog User's Manual,* University of Edinburgh, Edinburgh.

[10] [Tpro] Kluźniak F., Szpakowick S., Bień J.S., *Prolog for programmers,* Academic Press, 1985.

[11] *Recursive Data Types in SETL: Automatic Determination, Data Language Description and Efficient Implementation,* Tech. Rep. 201, Courant Institute of Mathematical Sciences, New York, 1984.

Gerald Weiss

Enidata
Bologna (Italy)