New Generation Computing, 6 (1988) 227-247
OHMSHA, LTD. and Springer-Verlag

Partial Evaluation of Metaprograms in a "Multiple Worlds" Logic Language*

Giorgio LEVI
Dipartimento di Informatica,
Universitā di Pisa.
Giuseppe SARDU
Systems & Management SpA, Pisa.

Received 17 March 1988

Abstract This paper describes a partial evaluation system specifically designed to be used as an automatic compilation tool for metaprograms in a KBMS (EPSILON) based on Prolog. EPSILON main underlying concepts are the extension of Prolog with theories ("multiple worlds") and the use of metaprogramming as the basic technique to define new inference engines and tools. Our partial evaluator is oriented towards theories and metainterpreter specialization. Being designed to be used as an automatic compiler, it does not require declarations from the user to control the unfolding process. It handles full Prolog and provides also an elegant solution to the problem of the partial evaluation of incomplete and self-modifying programs, by exploiting the multiple worlds feature added to Prolog. EPSILON partial evaluation system turned out to be a very useful and powerful tool to combine the low cost and the flexibility of metaprogramming with the performance requirements of a practical knowledge based system.

Keywords: Logic Programming, Metaprogramming, Partial Evaluation, Multiple Worlds, Metainterpreter Compilation, Knowledge Base Management Systems

§1 Introduction

In this paper we describe a partial evaluation system, specifically designed to be used as an automatic compilation tool in a Knowledge Base Management System (KBMS) based on Prolog. In our KBMS, metaprogramming is the basic technique to define new inference engines and tools and partial evaluation is

* Partially supported by ESPRIT, project 530 (EPSILON).

used as a systematic method to "compile" metaprograms. Our approach is similar to other proposals.[29,27] However, our algorithm has some interesting new features, which solve some relevant open problems. In fact, our partial evaluator

(1) handles (an extension of) full Prolog, including builtins, side-effects and cut,
(2) is designed to be used as an automatic compiler, with no need for "declarations" from the user to control the unfolding process,
(3) allows the compilation of partial (open) programs, by exploiting the multiple worlds feature added to Prolog.

The structure of the paper is the following. We first give an overview of the KBMS, to provide a better understanding of the environment, of the potential applications and of the role of partial evaluation. We then discuss partial evaluation of logic programs, in the case of pure logic programs, full Prolog and Prolog metaprograms. The description of our partial evaluation algorithm comes after an informal discussion of issues related to the "multiple worlds" case. We finally give some examples with the results of performance comparison between the original programs and the compiled programs.

§2 The EPSILON Knowledge Base Management System

The partial evaluator, that will be described in the following, is a component of the Knowledge Base Management System developed within the Epsilon project.[2,3,20]

Epsilon is a prototype of a knowledge base management system built on top of commercial PROLOG and Relational Data Base Management systems, running on standard UNIX environments. The main concepts underlying the Epsilon approach are:

(1) the extension of PROLOG with theories (multiple worlds),
(2) the definition of a transparent interface from PROLOG to Relational Data Base Management Systems,
(3) the use of metaprogramming as the basic technique to define new inference engines and tools,
(4) the use of partial evaluation techniques as a systematic method to "compile" metaprograms,
(5) the definition of a graphical user interface on a personal computer.

The *theory* is the basic component of the Epsilon knowledge base. Theories are similar to worlds in MULTILOG[16] and to unit worlds and instances in MANDALA.[8] Namely, they are composed of a chunk of knowledge, associated to a specific inference machine (theory processor). A theory corresponds to a chunk of knowledge contained in a file and is associated to a window in the graphical user interface. The theory processor contains operations to query the theory, to update and search the theory, to load/unload the

theory, considered as an atomic separate object. The theory processor can also contain tools (debugger, tracer, explanator, query-the-user). Epsilon provides two primitive theory processors (or *classes*) : the first one handles the language PROLOG extended with the theory feature, while the second one handles Data Base theories. As we will discuss later, new classes (theory processors) can be defined.

Adding theories to standard PROLOG allows to define a structure on the PROLOG workspace. This mechanism is currently simulated on top of a commercial PROLOG compiler.

The kernel of Epsilon maintains a Knowledge Base Dictionary, which contains a description of the existing theories (in particular, their classes). Theories can communicate by making a reference to the generic operations for querying and updating theories. The kernel uses the Knowledge Base Dictionary to select the operations of the proper inference engine.

Inference engines are handled as first-class citizens in Epsilon, since new inference engines can be defined inside theories. A knowledge base is then composed by homogeneous objects (theories) that can be either user (object level) theories or theory processors for other theories. If a theory T has class C, there exists a theory named C contaning the inference engine of T. It is therefore possible to build in a cleaner and natural way knowledge bases relying on specific domain knowledge and multiple layers of general (control) knowledge, and to extend in a simple and efficient way the features of the system without modifying the kernel.

A theory defining an inference engine for a class of theories must define the programs for querying (call) and for updating (assert and retract). Moreover, an engine can define tools. Metaprogramming is used to define the various inference machines. The definition of "enhanced" metainterpreters[26] is attractive, because it allows to befine new functionalities without modifying the program (the object level knowledge) and the basic interpreter. Enhanced metainterpreters can embed new control strategies, extend the logic language with new useful constructs (for instance, knowledge structure, or uncertainties) and the related inference rules (inheritance or approximate reasoning), or define analysis tools, to provide typical expert systems (explanation, query-the-user, etc.) or interactive monitoring (debugger, tracer, etc.) capabilities. One of the main features of the metaprogramming approach is its ability to extend the language, the inference machine and the environment, without modifying the basic building blocks, i.e. the PROLOG interpreter and compiler. The extensions defined as (PROLOG) metaprograms are easy to define and portable. Their performance is anyway rather poor, if compared to what could be obtained by an ad-hoc implementation of the new language/environment, which, however, is a very expensive solution, and, in addition, is not necessarily open to further extensions and modifications. Metaprogramming is, instead, easy, more flexible and clean, since the knowledge (the rules in the possibly extended language) and

the inference engine (the metainterpreter) are separate and easy to understand, and all the extensions in the inference engine are clearly defined at the meta-level.

A new inference engine conceptually defines a new knowledge representation language. The new language features can either affect the object level description language (as is the case, for instance, of clauses extended with uncertainties and of PROLOG extended with coroutining) or be represented at the meta-level, as relations among theories. We will mainly be concerned with the last case, which is realized in Epsilon defining *links* between theories and by representing them in the Knowledge Base Dictionary.

Some links define "new" inference rules for a theory. In such a case, the inference rule must be embedded in the query metainterpreters of the theory processors. For example, default communication mechanisms between theories T_1 and T_2 are achieved by defining an inheritance link from T_2 to T_1. This link is interpreted by the query handler of T_1 as follows. If a subgoal cannot be solved in T_1, it is solved in T_2. Multiple inheritance is possible. If the two theories share the same inference engine, the result is inheritance of the "object level" knowledge (i.e., the clauses of T_2 are available in T_1). Otherwise, in the case of theories having different inference engines, the subgoal is solved by the inference engine of T_2.

The real drawback of metaprogramming is performance. There exists, however, an interesting technique (partial evaluation of metainterpreters), which allows to combine the low cost and the high flexibility of metaprogramming with performance. This technique will be discussed in this paper, starting with the case of pure logic programs.

§3 Partial Evaluation in Logic Languages

Partial evaluation (Ershov's mixed computation[4,5]) is a procedure, which, given a logic program P and a (partially instantiated) goal G, derives a new program P′, which behaves like P under the partial instantiations in G. Logic languages (and, more generally, all the unification-based languages) are naturally handled by partial evaluation, since the partial evaluation inference rule is the same as the standard evaluation rule. In fact, the language supports unbound inputs (represented by logical variables), unfolding (i.e. resolution) is possible even if the arguments are partially determined and unification directly supports forward and backward data structure propagation. Moreover, input values are not forced to be constant values but can be partially determined data structures. The strong relation between partial evaluation and standard interpretation suggests that the partial evaluator should naturally be definable as a metainterpreter.

The first attempts to apply partial evaluation to logic programs were

- the definition of a PROLOG partial evaluator derived from a

specification of a PROLOG abstract machine (interpreter) in META-IV.[17,18] The partial evaluator transformations were formally proved to preserve the meaning of programs.

- the derivation of a PROLOG compiler in LISP[14,15] from a PROLOG interpreter and a LISP partial evaluator written in PROLOG.
- a partial evaluator for full PROLOG (including builtins and cut), defined as a PROLOG metainterpreter,[31] and its application to data base query optimization.

Let us first consider partial evaluation in the case of pure Horn clauses. Given a logic program P and a simple query :–g, consider one SLD-tree for :–g and assume that all the paths are either success paths of finite failure paths. Finite failure paths are evaluated as much as possible, i.e. when an atom in a goal cannot be rewritten, the other atoms in the goal are expanded. Each path i of the tree has an associated substitution λ_i and a leaf node labeled by a (possibly empty) goal :–g_i. The partial evaluation of P with respect to the goal :–g is then the set of clauses:

$$[g]\lambda_1 :\!-g_1,\ [g]\lambda_2 :\!-g_2,\ \ldots,\ [g]\lambda_n :\!-g_n.$$

In the case of recursive programs, the SLD-tree will generally have infinite paths and partial evaluation would be nonterminating. It is therefore necessary to select a suitable finite subtree, using some termination strategy. It is worth noting that the program derived for any strategy is always a logical consequence of the original program. However, only some termination strategies allow to derive a program which is equivalent to the original one, under the partial instantiations in :–g. The problem of defining a correct termination strategy is undecidable for the general case and has known solutions only for restricted classes of logic programs.[6]

The termination of recursive programs is just one case where one is faced with the problem of defining a strategy to inhibit the unfolding process. Similar problems arise when trying to control the size of the transformed program or the number of backtrackings (see below) and in "incremental" partial evaluation (see Section 6). A partial evaluation algorithm for logic programs is therefore essentially a strategy to control unfolding. Such a strategy can fruitfully be based on program analysis techniques,[13] which could provide the information about "when to inhibit unfolding".

Let us now briefly discuss the problems related to backtracking. The program derived by partial evaluation defines SLD-derivations which usually require a lower number of inferences (because some procedures are expanded). However, nondeterministic branchings are moved earlier in the SLD-tree, as shown by the following example. Consider the program P, defined by the clauses

1. A(X) :–B(X), C(X)
2. C(X) :–D(X)

3. C(X) :–E(X)

The partially evaluated version P′ of P with respect to the goal :–A(X) is defined by the following clauses

4. A(X) :–B(X), D(X)
5. A(X) :–B(X), E(X)

A single derivation path for a call of the procedure A in P′ is shorter than the corresponding path in P. However, with a search rule based on backtracking, if the call of the procedure D fails, a call of the procedure B is repeated, while trying to apply rule 5. This would not be the case for program P, where a failure on D would only require an attempt to evaluate E. A satisfactory solution, only when the different cases are characterized by the same variable bindings, is the introduction of the OR operator, as suggested in Ref. 31. In our example, we would obtain the clause

6. A(X) :–B(X), (D(X); E(X)).

The above sketched situation arises, whenever in the SLD tree one generates a goal of the form :–A_1, A_2, ..., A_n, such that, for some index j,

(i) at least one of the atoms A_1, ..., A_{j-1} is delayed (i.e. cannot be unfolded),
(ii) the partial evaluation of A_j generates new branchings in the tree.

In such a case, it is better not to expand atom A_j. Our solution,[12] similar to the one proposed in Ref. 10, is based on the following algorithm (*multiple partial evaluations algorithm*):

When a goal of the form :–A_1, A_2, ..., A_n is generated and A_1 cannot be unfolded, perform the following steps:

(i) (new predicate introduction) Add to the program the clause
newP(X_1, ..., X_r) :–A_2, ..., A_n,
where X_1, ..., X_r are all the variables occurring in A_2, ..., A_n and either in A_1 or in the goal at the root of the SLD-tree.
(ii) (folding) Replace the conjunction A_2, ..., A_n in the goal by the new clause head. The resulting goal is :–A_1, newP(X_1, ..., X_r).
(iii) (separate partial evaluation) Perform a separate partial evaluation of the goal
:–newP(X_1, ..., X_r).
which returns a set C of new clauses for newP.
(iv) (unfolding) If C contains one clause only, unfold the call to newP in the goal with the new clause. Otherwise
(v) (binding propagation without unfolding) Compute the atom a, which is the *minimal generalization* of all the heads of the clauses generated for newP. A generalization of a set of atoms {a_1, ..., a_n} is an

atom a, such that there exist substitutions λ_1, ..., λ_n such that $[a]\lambda_1 = a_1$, ..., $[a]\lambda_n = a_n$. A generalization a is minimal if there is no other generalization h such that $[a]\lambda = h$. The binding propagation is achieved by unfolding the atom new$P(X_1, ..., X_r)$ with the clause a :-a.

As we will show in the next section, steps (iv) and (v) become more complex, if we take into account the properties of specific PROLOG primitives.

§4 Partial Evaluation in PROLOG

When moving from pure Horn clauses to PROLOG, we are faced with the problem of handling the various primitives.

In the partial evaluation of primitive calls, two cases may occur.

- the primitive call can be evaluated, resulting in either success or fail,
- the primitive call must be delayed, i.e. left unchanged in the current goal.

For each primitive, the partial evaluator knows under which conditions the primitive call can be evaluated. For example, a primitive call of the form *clause*(X, Y) can only be evaluated if its first argument is not a variable. A call of the form *functor*(T, F, A) can be evaluated if T is not a variable or if F and A are bound to a functor and to a natural number respectively. In all the other cases, the primitive calls are delayed. Of course, calls of primitive operations which cause side-effects (such as assert, retract, read and write) are always delayed.

Steps (iv) and (v) in the multiple partial evaluation algorithm must be modified to correctly handle calls of primitives which cannot be subject to backward binding propagation (unification). The problem arises because some primitives do not define relations and some parameters can only be used either as inputs or as outputs. In such a case, in fact, we cannot allow backward propagation of variable bindings to that atom. Let us consider an example, in which, the SLD tree contains a goal of the form :-*Prim*(X, Y), B(X), C(Y), where *Prim* is a builtin whose arguments are both input parameters. The procedure call *Prim*(X, Y) is delayed, but the expansion of procedures B and C is not allowed to bind variables X and Y, since this would change the input-output behaviour of the call to *Prim* in the partially evaluated program. If this is the case, variables X and Y become "frozen". Moreover, as noted in Refs. 22) and 32), values cannot be backward propagated over builtins which cause side-effects.

Steps (iv) and (v) must then be extended to check that the unfolding and/or the binding propagation do not instantiate any frozen variable. Note that all the variables occurring in calls of primitives with side-effect are frozen.

There is still an open problem related to the primitives assert and retract: some "self-modifying" programs cannot be correctly partially evaluated, as shown by the following example:

```
p(X) :-q(X).
p(X) :-assert(q(a)).
q(b).
```

The result of partial evaluation

```
p(b).
p(X) :-assert(q(a)).
q(b).
```

is not equivalent to the original program, since it will never compute the answers X=a to the query p(X). Our solution to this problem will be discussed in Section 6.

A last relevant problem is related to the partial evaluation of meta-level primitive calls (such as *call*, *not*, *if-then-else*, etc.) and of the pure control primitives *cut* and *fail*. We will discuss our treatment of *cut* later.

§5 Partial Evaluation of Metaprograms

Partial evaluation techniques were recently applied to metaprograms in the framework of PROLOG,[28,29,10,27,7] Flat Concurrent Prolog[23,24,25,30] and of a functional language.[13] As we will discuss in the following, it can be viewed as a powerful, systematic and clean compilation technique. It has been used to effectively derive an efficient compiler-compiler[13] and to define the various virtual machines of LOGIX.[23,24,25] Our interest is verifying the feasibility of combining partial evaluation with metainterpreters, used to define language extensions or tools. The same approach is being pursued in Refs. 28), 29), 27) and 7).

In partial evaluation of metaprograms, the partial input values are procedure calls. The partial evaluation of the metaprogram M applied to a call of the procedure P generates a specialization of M, which can be viewed as a new version (P′) of P. The new definition of P is a version of P, embodying some of the features relevant to M. This allows to replace a metacall to P (by means of M) by a direct call to P′.

Let us consider, as an example, a metainterpreter eval, which takes a procedure call and returns

- true, if the procedure call succeeds,
- false, if the procedure call finitely fails.

```
1. eval(true, A) :-!, A=true.
2. eval((A, B), C) :-!, eval(A, E), eval(B, F), and (E, F, C).
3. eval(A, B) :-syspred (A), !, call(A), B=true.
4. eval(A, B) :-clause(A, D), eval(D, B).
5. eval(A, false) :-not clause(A, D).
```

Consider now an "object level" procedure member, defined as follows:

6. member(X, X.L).
7. member(X, Y.L) :-X\==Y, member(X, L).

The partial evaluation of the goal eval(member(X, L), Y). derives the clauses

8. eval(member(A, [A|B]), true).
9. eval(member(A, [B|C]), true) :-A\==B, eval(member(A, C), true).
10. eval(member(A, [B|C]), false) :-A\==B, eval(member (A, C), false).
11. eval(member(A, B), false) :-not clause 1(member(A, B), C).
12. clause 1(member(A, [A|B]), true).
13. clause 1(member(A, [B|C]), (A\==B, member(A, C))).

If we define member1(X, L, Y)=eval(member(X, L), Y), we obtain a version of the member procedure which contains the extra information provided by eval:

8′. member1(A, [A|B], true).
9′. member1(A, [B|C], true) :-A\==B, member1(A, C, true).
10′. member1(A, [B|C], false) :-A\==B, member1(A, C, false).
11′. member1(A, B, false) :-not clause1(member(A, B), C).

Each metacall of the form eval(member(X, L), Y) can now be replaced by a more efficient direct call of the form member1(X, L, Y).

The example shows the key aspect which makes partial evaluation interesting in the case of metaprograms. Metainterpreters, in fact, introduce some loss of efficiency, since metacalls are more expensive than direct calls. The flexibility of metaprogramming can then be combined with efficiency, if partial evaluation is used to transform all the metacalls into direct calls.

The construction, based on the first Futamura projection,[9] is the following.

- metaprogramming can be viewed as a method to implement a new language L′, by defining an interpreter of L′, written in the (already implemented) language L.
- the partial evaluation of the interpreter applied to a program in L′ generates a compiled program which can directly be executed on the abstract machine associated to L.

The partial evaluation of a goal consisting of a metainterpreter M applied to a call of the procedure P returns a new version P′ of P, such that the direct execution of P′ is equivalent to the execution of P through the metainterpreter M.

If M is the pure metainterpreter (without new inference rules or extended features), P′ must be equivalent to P. Partial evaluation is therefore essentially the first reflection principle,[1] which reflects a metalevel proof (simulation) into

the equivalent object level proof (direct execution).

If the metainterpreter M contains additional inference rules (and the corresponding additional features), partial evaluation compiles the new features in the procedure P. For example,

- If M is the explanation metainterpreter, the procedure P′ is a version of P, providing the explanation feature, when executed by the standard interpreter.
- If M is a debugger metainterpreter, P′ is the version of P "instrumented" to allow the debugging with the standard execution.
- If M is a "query-the-user" metainterpreter, P′ is the version of P which queries the user when executed by the standard interpreter.
- If the "new" language contains negative atoms, in the form of calls to a metainterpreter evalnot, which defines the "negation as finite failure" inference rule, P′ is a version of P which allows to compute negative atoms through direct calls (i.e. in the original language).
- If the "new" language contains structuring concepts, such as multiple worlds and inheritance links, supported by metainterpreters embodying the corresponding inference rules, the language can be compiled to the original unstructured language.

§6 Partial Evaluation in a Multiple Worlds Logic Language (Epsilon)

The general case of partial evaluation of theories as viewed by metaprograms is shown in Fig. 1, where T3 is obtained by partially evaluating the knowledge in T1, under the metainterpreter demo, belonging to the inference

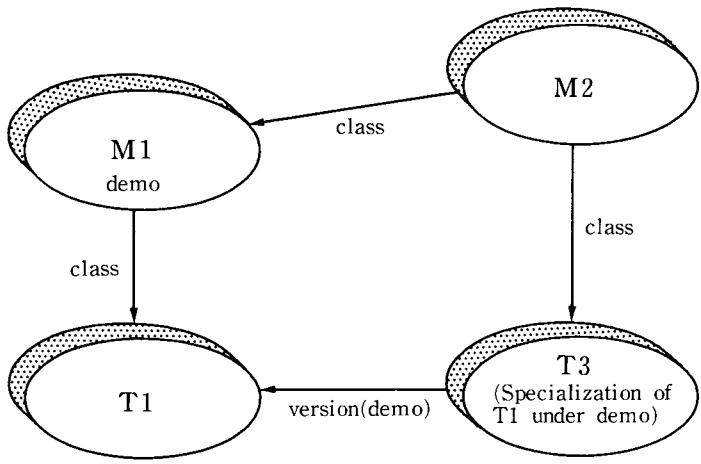

Fig. 1 Partial evaluation of theories with metaprograms.

engine M1 of T1.

T3 is linked to T1 by the link *version(demo)*. The semantics of such a link is twofold. When a call of demo applied to some predicate in T1 is found, it is converted into the corresponding call in T3. Moreover, when T1 is updated, T3 must be kept consistent with T1 updatings. Note that the compilation of one interpretation layer is shown by the fact that the inference engine of T3 is now M2.

Partial evaluation of a theory, as viewed by a specific metaprogram, is achieved by a procedure

compile(Source_theory, Meta_program, Target_theory),

which, given a Source_theory and a Meta_program (in the inference engine of Source_theory), creates a derived theory Target_theory, linked to Source_theory by the link

version(Meta_program).

Assume that Source_theory contains a definition for the predicates p_1, ..., p_k with arities n_1, ..., n_k, and assume that the metaprogram m has the form m(*"object-level-atom"*, ...).

The derived theory will contain the result of partial evaluation, in Source_theory, of all the goals of the form $m(p_i(X1, ..., Xn_i), ...)$, i.e. each goal is the application of the metaprogram m to a call of a procedure p_i, with no constraints on the inputs.

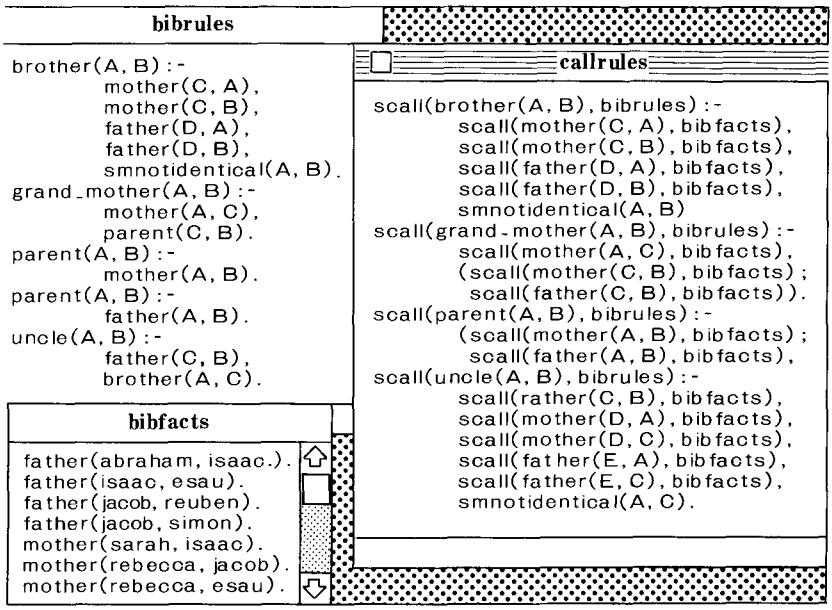

Fig. 2 Two theories and the result of partial evaluation.

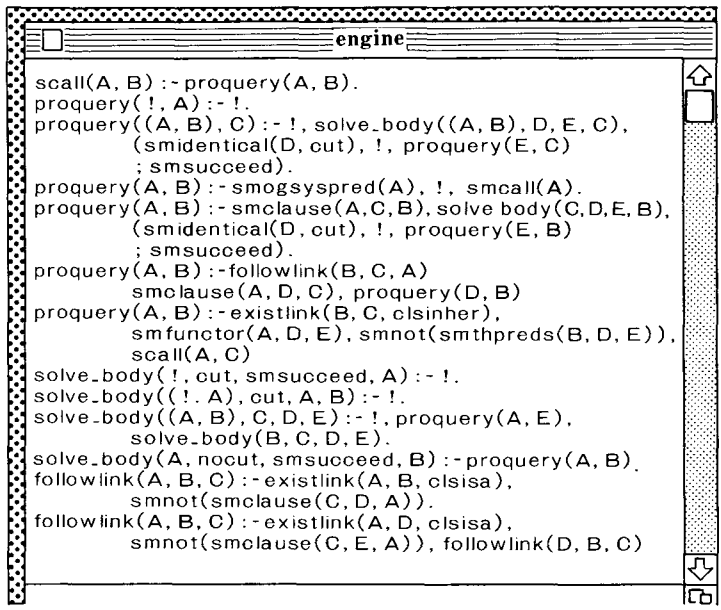

Fig. 3 The query metainterpreter of the inference engine **engine**.

As an example, let us consider the theory *bibrules* (in Fig. 2), whose inference engine (the theory *engine*) contains the metaprogram (query-metainterpreter) *scall* (in Fig. 3).

scall defines a language which is essentially PROLOG (including cut), extended with some inheritance rules, based on the links *clsinher* and *clsisa*. If T1 inherits from T2 through the link *clsinher*, then queries failed in T1 can be solved in T2. If T1 inherits from T2 through the link *clsisa*, then queries failed in T1 can be solved (in T1) using clauses of T2. The language used in all the examples is the PROLOG variant used in our project, where PROLOG primitives have generally the prefix "sm".

A call to the partial evaluator of the form *compile*(*bibrules*, *scall*, *callrules*) creates in the theory *callrules* the result of the partial evaluation of the goals

scall(brother(X, Y), bibrules),
scall(grand_mother(X, Y), bibrules),
scall(parent (X, Y), bibrules),
scall(uncle(X, Y), bibrules).

A program (theory) can be made parametric with respect to some of the theories it uses or inherits. This is relevant to the optimization of incomplete knowledge, which can be encapsulated into a theory T*, which may be made visible to a theory T, by defining a suitable link interpreted by the inference engine of T. Such a link, however, is not visible to the partial evaluator, which,

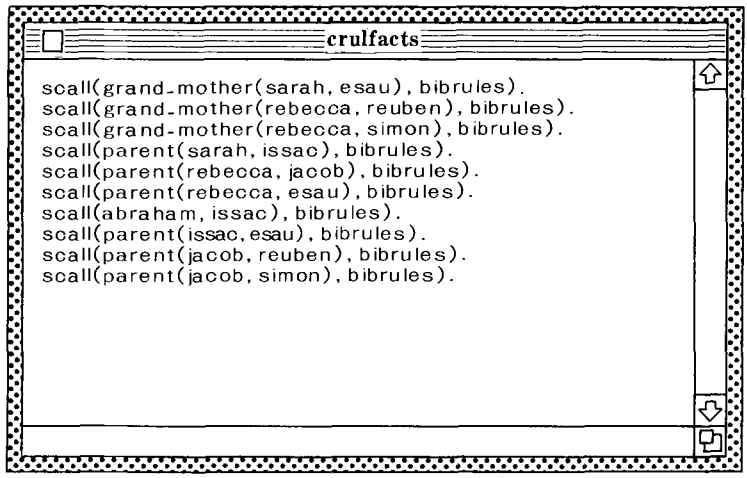

Fig. 4 Partial evaluation, with *bibfacts* visible.

when applied to T, cannot evaluate all the references to T*. Thus the partial evaluation of T is independent from the current content of T*.

In our example, the theory *bibfacts* can be inherited from *bibrules* (link *clsinher*). If we make *bibfacts* not visible to partial evaluation, the result of *compile*(*bibrules*, *scall*, *callrules*) is that shown in the theory *callrules* in Fig. 2. Programs in *callrules* are much more efficient than the application of the metainterpreter *scall* to the original programs in *bibrules*. For example, computing all the answers to the query *parent*(X, Y) in *bibrules* requires 138 logical inferences, while the same task in *callrules* requires 72 logical inferences only. All the metainterpreter components which can be "statically" evaluated disappear from the "compiled program". Note also that the metalevel simulation of the standard PROLOG interpreter (first 4 clauses of *proquery*) is not needed. In particular, since the main functor of the first argument of *scall* is known, all the accesses to clauses in *bibrules* can statically be solved and the only metacalls are those to the invisible theory *bibfacts*.

The result of the same partial evaluation, with theory *bibfacts* visible, is shown in the theory *crulfacts* in Fig. 4. Computing all the answers to the query *parent*(X, Y) in *crulfacts* requires now 7 logical inferences only. However, the result depends on the current content of *bibfacts*.

One important feature of our partial evaluation method is that not only the inputs but the program itself can be partially specified. This feature is relevant to the case of optimization with incomplete knowledge, which is typical of artificial intelligence applications. Another example comes from deductive databases, where the set of tuples (or ground facts) is stored in secondary storage and is usually subject to modification. The optimization of the deductive component (logic program) should not be affected by the current contents of the database. We can therefore split the logic program into separate theories, with

links interpreted by the inference engines, yet not visible to the partial evaluator. All the references to theories which are not visible to the partial evaluator will then be delayed.

It is worth mentioning that our mechanism provides an elegant solution to the problem of partial evaluation of "open programs",[27] which is either faced by requiring explicit declarations from the user[31,28] or by-passed by assuming only complete closed programs.[27] In our case, incomplete knowledge can only be related to non visible theories, while theories visible to partial evaluation are considered "closed-worlds". With our technique, we obtain an "incremental compilation": the result of partial evaluation does only depend upon the current contents of visible theories and must accordingly be kept consistent with updating on those theories only.

This feature can also be exploited to provide a solution to the self-modifying programs problem, mentioned in Section 4. The knowledge (of a theory) can be splitted into a static component P (the program) and a dynamic component S (the modifiable program state), represented as separate theories. The theory S is inherited by P and, when P is partially evaluated, S is made invisible. The example of Section 4 can now be rephrased as follows:

theory P: p(X) :–q(X).
p(X) :–smassert(q(a), S).

theory S: q(b).

The partial evaluation of P is now the theory

p(X) :–smcall(q(X), S).
p(X) :–smassert(q(a), S).

which correctly interacts with the updates in S.

§7 The Partial Evaluation Algorithm

This section describes the partial evaluation algorithm used within *compile*. Such an algorithm is specifically oriented towards theories and meta-programs. In principle, the algorithm for metainterpreter specialization could be simpler than the partial evaluation algorithms used for program optimization. In fact, in metainterpreter specialization we are faced with three possibly independent subproblems:

(1) removing all the book-keeping associated with one layer of interpretation,
(2) optimization of the metainterpreter algorithm,
(3) optimization of the source program.

Subproblem (1) (i.e. pure metainterpreter specialization) can be satisfactorily solved by a very simple algorithm (see, for example, Ref. 27), which

- requires almost no unfolding,
- has no backwards unification and termination problems,
- generates target programs whose sizes are similar to the sizes of the source programs.

We are essentially interested in pure metainterpreter specialization, even if we obtain some further optimization by using a more liberal unfolding control and by allowing backward substitutions. In fact, pure metainterpreter specialization allows us to eliminate the loss of performance introduced by the metainterpreter, which is usually at least one order of magnitude. More sophisticated partial evaluation algorithms would probably require user interaction and would not allow their use as an automatic compilation tool.

Our algorithm is based on two mutually recursive procedures, *parteval* and *unfold*, that are informally described in the following.

parteval(G, Source_theory, Target_theory)

Asserts in Target_theory all the clauses which are obtained by partially evaluating the goal G in Source_theory. For each path in the SLD-tree of G, *parteval* provides for the generation of a call to *unfold*(*G*, *Newg*) and asserts the clause *G* :–*Newg*. Moreover, it contains the treatment of occurrences of *cut* in Newg (see below).

Goals submitted to *parteval* within the same call of *compile* on Source_theory are stored, so as to avoid multiple partial evaluations (and clause generations). Namely, if the current goal G is a variant of a goal already submitted to *parteval*, it is ignored.

unfold(Input_goal, Output_goal)

Instantiates Output_goal to the unfolding of the Input_goal. Unfolding is the repeated application of leftmost SLD resolution, until the first atom in the current goal is delayed, i.e. it cannot be rewritten.

An atom A is delayed in *unfold* if one of the following conditions is satisfied:

- A is a (recursive) procedure call, which is an instantiation of a previous call within the same *unfold* process. If A is a metainterpreter call, recursion is detected on its "procedure call" argument.
- A is a primitive call which cannot be evaluated. As already mentioned, unfold contains an explicit description of all the conditions which must be verified for the evaluation of primitive calls.
- A is the call of a procedure which could be resolved with clauses whose bodies contain occurrences of *cut* (and A is

different from the goal argument of the most recent call of *parteval*). In such a case, the unfolding would not, in fact, be correct.

When in the current goal G_1, G_2, ..., G_n, the atom G_1 is delayed, *unfold* is suspended and a new ***parteval*** process is created, according to the multiple partial evaluations algorithm described in Section 3. If the new predicate call can be unfolded, the corresponding clause generated by *parteval* is retracted from Target_theory. The analysis on the applicability of unfolding and binding propagation can only be made if suitable information is associated to delayed atoms. Namely, for each delayed atom we must know whether it has side-effects and which are the frozen variables. Note that delayed calls of recursive procedures (and procedures containing *cut*) have no frozen variables and are analyzed to decide whether they have side-effects.

As already noted, a procedure call G, such that its unfolding would use clauses whose body contains *cut*, is unfolded only if G is the argument of the most recent call of *parteval*. Each clause produced by the unfolding is then analyzed by *parteval*. If a clause has a *cut* as the first atom, the *cut* is removed. Moreover, the remaining alternative unfoldings are ignored, if the head of the currently generated clause is a variant of the original goal submitted to *parteval*.

If G is not the argument of the most recent call of *parteval*, it is delayed by *unfold* and a new activation of *parteval* on G is created (this will allow to unfold G separately). If the resuly of *parteval* is a set of clauses which do not contain *cut*, the unfolding of G (with the new clauses) can be resumed.

It is worth noting that metainterpreters usually contain several disjoint cases, implemented by means of *cuts*. The "compilation" of metainterpreters, strongly relies upon the ability to eliminate unnecessary *cuts*. This is why our algorithm is more concerned with *cut* elimination than with *cut* optimization. Note also that all the existing partial evaluation systems oriented towards metaprograms[29,27] do not handle cuts, thus imposing too strong constraints on the language to be used for metaprogramming.

§8 One Example: Partial Evaluation of a Forward Chaining Metaprogram

The metaprogram contained in the inference engine *engine* in Fig. 5 defines a procedure *forw*, which, given a new fact *Fact*, generates all its consequences in the theory *Th* and puts them in the theory *ThRes*.

The procedure *forw* defines the following algorithm. For each clause in the theory (smthpreds (Th, P, N) is a backtrackable operation defined on a theory Th which returns the name P and the arity N of each predicate defined in Th), whose body contains an atom which is unifiable with *Fact*, if all the

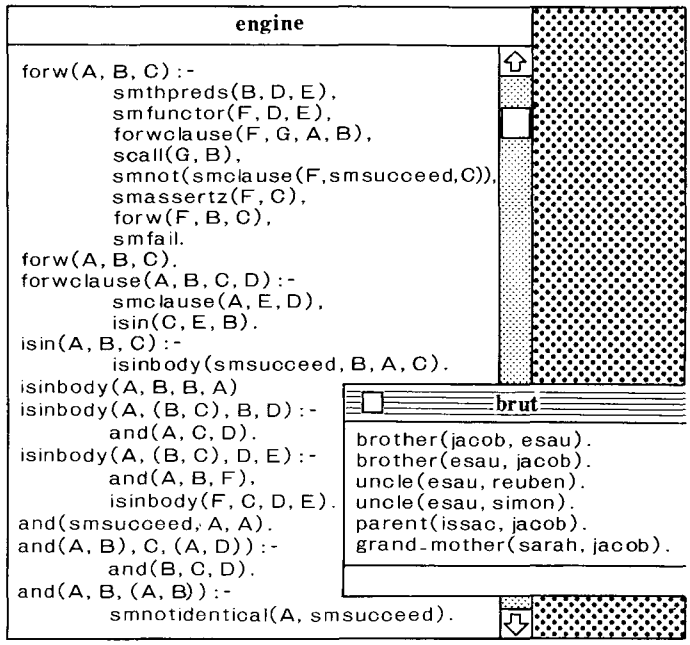

Fig. 5 The metaprogram *forw* and the result of its application to *bibrules*.

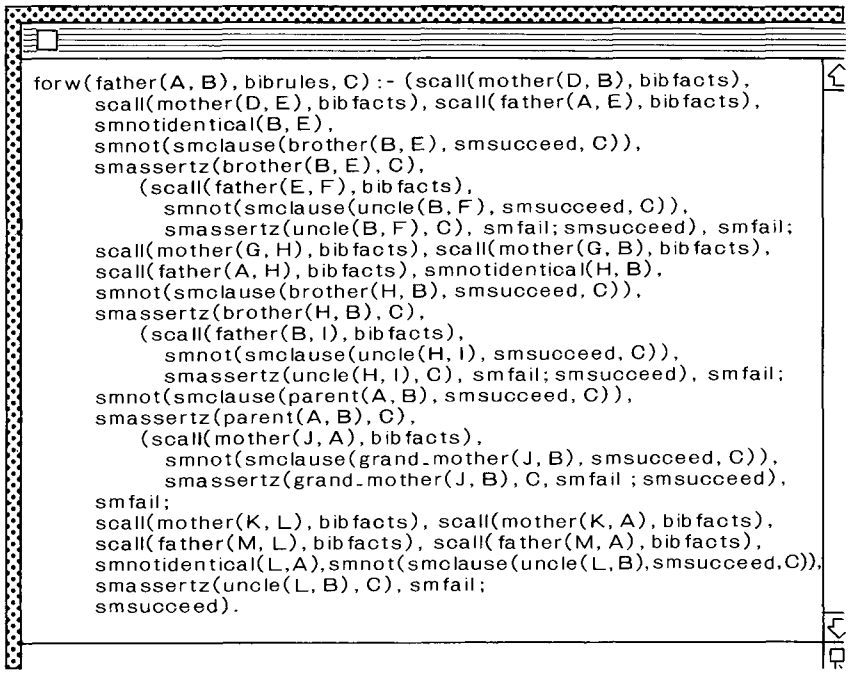

Fig. 6 The result of the partial evaluation of *forw*(*father*(*X*,*Y*), *bibrules*, *Z*).

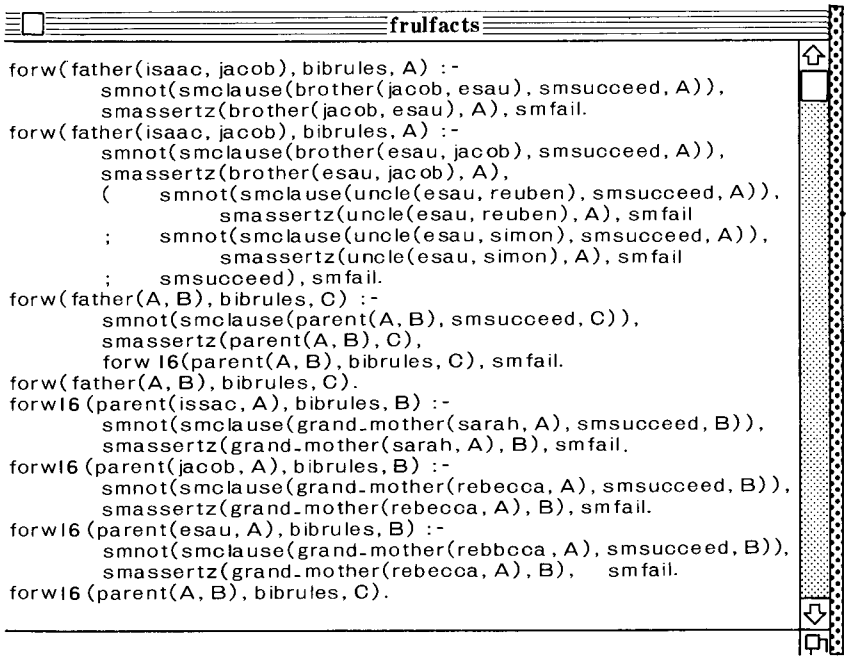

Fig. 7 The result of partial evaluation of *forw*(*father*(*X*, *Y*) *bibrules*, *Z*), with *bibfacts* visible.

other atoms in the body can be solved in *Th*, asserts the instantiated clause head in *ThRes* and recursively derives the consequences of the new assertion. *scall* is once again the query metainterpreter of Fig. 3.

For example, let us consider the theories shown in Fig. 2. By evaluating *forw*(*father*(*isaac*, *jacob*), *bibrules*, *brut*) we obtain the theory *brut* in Fig. 5.

The algorithm is very inefficient, because there is no fast method to access the relevant clauses. All the considerations already made about the partial evaluation of the metainterpreter *scall* apply to *forw* as well. In this case, however, the optimization on clause access is dramatic and allows to statically generate the "forward chaining" version of all the relevant clauses. As usual, the links between theories must explicitly be made visible to the partial evaluator. Let us consider the case where the link between *bibrules* and *bibfacts* is not visible. The theory *tat* in Fig. 6 contains the result of the partial evaluation of *forw*(*father*(*X*, *Y*), *bibrules*, *Z*).

Note that *forw* has been dramatically simplified and the only metacalls are those related to the invisible theory *bibfacts*. The result of the partial evaluation is therefore parametric with respect to the content of *bibfacts*, since we partially evaluate *forw* only with respect to the general knowledge contained in the theory *bibrules*.

If *bibfacts* is made visible to the partial evaluation of *forw*(*father*(*X*, *Y*), *bibrules*, *Z*), we obtain the more efficient version (not containing any invocation

of *scall*), contained in the theory *frulfacts* in Fig. 7. *frulfacts* now depends both on *bibrules* and on *bibfacts*, and must be updated whenever one of the two theories is updated.

Let us conclude with some experimental performance results. The query

forw(*father*(*isaac*, *jacob*), *bibrules*, *brut*),

if executed in *engine*, requires 2567 logical inferences. The same query requires 459 logical inferences in *tat* (partial evaluation with respect to *bibrules* only) and 22 logical inferences only in *frulfacts* (where partial evaluation considers *bibfacts* too).

§9 Conclusions

The partial evaluation system we have described has been tested on several rather complex examples and has always given satisfactory results. Our treatment of cuts and side-effects primitives is correct, while the multiple worlds feature turned out to be very useful and powerful. The gain in performance is usually impressive and the partial evaluator succeeded in the compilation of levels of interpretation in all our examples. We are currently performing some experiments with metainterpreters affecting the execution control (for example, a metainterpreter which executes in a coroutine style Prolog extended with a freeze operator).

The main problem with the current prototype is the efficiency of the compilation process. Optimizations on the basic algorithm have been studied and are currently under experimentation. Future developments include

- the self-application of the partial evaluator (with specific metainterpreters) to derive efficient compilers, as already shown in Refs. 13), 29) and 7). This would contribute to the performance of the compilation process;
- the study of techniques which allow to automatically mix the functionalities of different metainterpreters, as first suggested by Refs. 27) and 19). This is an important issue in the framework of our KBMS, where the user can freely define new languages (classes and inference engines), and could be relieved from the task of defining all the language related tools. Initial results in this direction can be found in Ref. 21.

References

1) Bowen, K. A. and Kowalski, R. A., "Amalgamating language and metalanguage in logic programming," in *Logic Programming* (K. L. Clark and S.-A. Tarnlund, eds., Academic Press, pp. 153-172, 1982.
2) Coscia, P. et al., "The Epsilon Knowledge Base Management System: architecture and Data Base access optimization," *Workshop on Logic Programming and Data Bases*, Venezia, Italy, December, 1986.

3) Coscia, P.,Franceschi, P., Levi, G., Sardu, G. and Torre, L., "Object level reflection of inference rules by partial evaluation," in *Meta-Level Architectures and Reflection* (D. Nardi and P. Maes, eds.), North-Holland, 1987.
4) Ershov, A., "On the partial evaluation principle," *Information Processing Letters, Vol. 6, No. 2*, pp. 38-41, 1977.
5) Ershov, A., "Mixed Computation : Potential applications and problems for study," *Theoretical Computer Science, 18*, pp. 41-67, 1982.
6) Fanti, L. and Zanobetti, S., *Proprietã della valutazione parziale in programmazione logica* (in Italian), Tesi di laurea, Università di Pisa, Italy, 1986.
7) Fujita, H. and Furukawa, K., "A self-applicable partial evaluator and its use in incremental compilation," in *Workshop on Partial Evaluation and Mixed Computation* (D. Bjørner, A. P. Ershov and N. D. Jones, eds.), Gl. Avernaes, Denmark, October, 1987. [to appear in *New Generation Computing*]
8) Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M. and Ueda, K., "Mandala : A logic based knowledge programming system," *International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, pp. 613-622, 1984.
9) Futamura Y., "Partial evaluation of computation process —an approach to a compiler-compiler," *Systems, Computers, Controls, Vol. 2, No. 5*, pp. 45-50, 1971.
10) Gallagher, J., "Transforming logic programs by specializing interpreters," *ECAI86. 7th European Conference on Artificial Intelligence*, Brighton, U. K., pp. 109-122, 1986.
11) Gallagher, J. and Codish, M., "Specialisation of Prolog programs using abstract interpretation," *Workshop on Partial Evaluation and Mixed Computation* (D. Bjørner, A. P. Ershov and N. D. Jones, eds.), Gl. Avernaes, Denmark, October, 1987. [to appear in *New Generation Computing*]
12) Ghelfo, S. and Levi, G., "A partial evaluator for metaprograms in a multiple theories logic language," *ESPRIT Project 530 (Epsilon) Report*, October, 1986.
13) Jones, N. D. Sestoft, P. and Sondergaard, H., "An experiment in partial evaluation: The generation of a compiler-compiler," in *Rewriting Techniques and Applications* (J.-P. Jouannaud, ed.), *Lecture Notes in Computer Science, Vol. 202*, Springer-Verlag, pp. 124-140, 1985.
14) Kahn, K. M., "A partial evaluator of Lisp written in a Prolog intended to be applied to the Prolog and itself which in turn is intended to be given to itself together with the Prolog to produce a Prolog compiler," *Technical Report, UPMAIL*, Uppsala University, Sweden, March, 1982.
15) Kahn, K. M. and Carlsson, M., "The compilation of Prolog programs without the use of Prolog compiler," *International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, pp. 348-355, 1984.
16) Kauffmann, H. and Grumbach, A., "Representing and manipulating knowledge within "worlds"," *Proc. First Int'l Conf. on Expert Data Base Systems* (L. Kershberg., ed), pp. 61-73, 1986.
17) Komorowski, H. J., "A specification of an abstract PROLOG machine and its application to partial evaluation," *Linkoping Studies in Science and Technology Dissertations, N. 69*, Linkoping University, Sweden, 1981.
18) Komorowski, H. J., "Partial evaluation as a means for inferencing data structures in an applicative language : A theory and implementation in the case of PROLOG," *Ninth ACM Symp. on Principles of Programming Languages*, Albuquerque, New Mexico, pp. 255-267, 1982.
19) Lakhotia, A. and Sterling, L., "Composing logic programs with clausal join," *Workshop on Partial Evaluation and Mixed Computation* (D. Bjørner, A. P. Ershov and N. D. Jones, eds.), Gl. Avernaes, Denmark, October, 1987. [to appear in *New Generation Computing*]

20) Levi, G., Modesti, M. and Kouloumdjian, J., "Status and evolution of the Epsilon prototype," *Proc. ESPRIT Technical Week*, Bruxelles, Belgium, 1987.
21) Levi, G. and Sardu, G., "Meta-level definition and compilation of inference engines in Epsilon," *ESPRIT Project Epsilon (P 530) deliverable 14a*, October, 1987.
22) O'Keefe, R. A., "On the treatment of cuts in Prolog source-level tools," *Proc. Int'l Symp. on Logic Programming*, Boston, Mass., 1985.
23) Safra, S., "Partial evaluation of Concurrent Prolog and its implications," *Master's Thesis*, Weizmann Institute of Science, Rehovot, Israel, July CS86-24, 1986.
24) Safra, S. and Shapiro, E., "Meta-interpreters for real," in *Information Processing 86* (H.-J. Kugler, ed.), Dublin, Ireland, North-Holland, pp. 271-278, 1986.
25) Shapiro, E., "Concurrent Prolog: A progress report," *Computer*, *Vol. 19*, *No. 8*, pp. 44-58, 1986.
26) Sterling, L., "Meta-interpreters for expert systems," *CAISR TR 134-85*, Case Western Reserve University, 1985.
27) Sterling, L. and Beer, R. D., "Incremental flavor-mixing of meta-interpreters for expert system construction," *Proc. 3rd Symp. on Logic Programming*, Salt Lake City, Utah, pp. 20-27, 1986.
28) Takeuchi, A. and Furukawa, K., "Partial evaluation of PROLOG programs and its application to metaprogramming," *ICOT Tech. Report*, 1985.
29) Takeuchi, A. and Furukawa, K., "Partial evaluation of PROLOG programs and its application to metaprogramming," in *Information Processing 86* (H.-J. Kugler, ed.), Dublin, Ireland, North-Holland, pp. 415-420, 1986.
30) Takeuchi, A., "Affinity between metainterpreters and partial evaluation," in *Information Processing 86* (H.-J. Kugler, ed), Dublin, Ireland, North-Holland, pp. 279-282, 1986.
31) Venken, R., "A PROLOG meta-interpreter for partial evaluation and its application to source-to-source transformation and query optimization, in *ECAI-84: Advances in Artificial Intelligence* (T. O'Shea, ed.), Pisa, Italy, North-Holland, pp. 91-100, 1984.
32) Venken, R. and Demoen, B., "A partial evaluation system for PROLOG: Theoretical and practical considerations," in *Workshop on Partial Evaluation and Mixed Computation* (D. Bjørner, A. P. Ershov and N. D. Jones, eds.), Gl. Avernaes, Denmark, October, 1987. [to appear in *New Generation Computing*]