METAINTERPRETERS FOR EXPERT SYSTEM CONSTRUCTION

LEON STERLING AND RANDALL D. BEER

▷ We discuss the use of metainterpreters for building expert systems in PROLOG. Three issues are covered. The first is a technique for mixing a metainterpreter into an object program imbuing it with the functionality specified by the metainterpreter. Mixing a metainterpreter into a PROLOG program consists of two steps: partially evaluating the metainterpreter with respect to the object program and pushing down the metaarguments of the metainterpreter into the object program. The second issue is a classification of metainterpreters into structural, contextual, and behavioral enhancements. Examples are given of useful enhancements for building expert systems. Finally, we discuss the combination of several metainterpreters and how a programming environment for building expert systems could be built based on the ideas in this paper. ◁

1. INTRODUCTION

Expert systems have been commonly decomposed into a knowledge base and inference engine. This decomposition is not entirely appropriate for expert systems written in PROLOG. Much of what an inference engine does is provided by PROLOG itself; knowledge bases are executable. However, simple knowledge bases written in PROLOG are generally insufficient as expert systems.

Three limitations of PROLOG for building expert systems are commonly cited. There is no explanation capability, no mechanism for reasoning with uncertainty, and an inflexible control regime. This paper discusses how metainterpreters can be used to overcome these limitations.

Clark and McCabe [7] suggested adding features such as explanations or uncertainty reasoning by modifying the PROLOG program constituting the knowledge

Address correspondence to Professor L. Sterling, Department of Computer Science, Case Institute of Technology, Case Western Reserve University, Cleveland, Ohio 44106.

Received 10 July 1987; accepted 11 October 1987.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/89/$3.50

base. Extra arguments are added to each predicate to maintain the necessary information—the uncertainty, for example, or a representation of the proof tree. This approach retains the efficiency and structure of the knowledge base, but sacrifices modularity and clarity. The entire knowledge base must be altered to incorporate a new feature. Furthermore, a purely declarative program is altered to include procedural concerns such as an uncertainty calculus. Blurring of tasks in this way is generally confusing.

An alternative approach for adding extra features is based on metainterpreters [24]. The knowledge base of the expert system is kept intact, and a metainterpreter is written to specify how to associate the feature with the knowledge base. Meta interpreters can also be used to express alternate control.

Solving a query with a metainterpreter rather than PROLOG is known in the folklore to cost an order of magnitude. Some confirming statistics are presented in [28] and [25]. One claim of this paper is that partial evaluation can be realistically used to solve this problem.

The partial evaluation of metainterpreters has been discussed by Gallagher [9] and Takeuchi and Furukawa [28]. It is a program transformation technique which can be used to specialize a metainterpreter to a given object program in such a way that the extra level of interpretation is removed while the functionality is retained. Controlling partial evaluation is difficult in general. Here, by restricting our task to removing the level of intepretation introduced by the metainterpreter, we avoid many of these difficulties.

An important pragmatic issue in the use of metainterpreters for building expert systems is their combination. For example, a user may wish to have both an explanation capability and an uncertainty calculated for her program where both capabilities are expressed as separate metainterpreters. Rather than writing an entirely new metainterpreter, however, she should have some way to simply integrate the functionality of the existing ones. In considering the ways in which various metainterpreters can be combined, we have found a classification of their enhancements to an object program into structural, contextual, and behavioral to be indispensible.

We advocate a particular methodology for building expert systems in PROLOG. The domain knowledge is cleanly separated from its use. The functionality of the system is incrementally developed by selecting from a library of existing metainterpreters, tailoring and combining them as required. Finally, the knowledge base is imbued with the functionality of the combined metainterpreter via partial evaluation, resulting in an efficient object level program.

This style of development, where the programmer assembles a system from modular pieces, is reminiscent of the use of inheritance in the object-oriented programming paradigm, especially the multiple inheritance of FLAVORS [6, 19, 30]. In FLAVORS, the functionality of some base flavor may be enhanced by selecting from a set of potential mix-ins. Consequently, we draw on the concepts and terminology of object-oriented programming in describing our work.

In this paper we concentrate on metainterpreters for PROLOG. The discussion, however, could be applied easily to metainterpreters for other logic programming languages such as FCP [17]. Most comments referring to PROLOG would equally apply to other languages.

This research continues the outline given in [24]. It is an application of general work in the logic programming community on metainterpreters [8, 11, 18, 21] and

partial evaluation [3, 13, 14, 28, 20] to expert system construction. Implicit in our research, but not addressed in this paper, are many interesting metalinguistic issues such as those raised in [4] and [29].

An overview of the paper is as follows. The next section introduces the notion of metainterpreters for PROLOG. Sections 3 and 4 discuss program transformation techniques which remove the extra level of interpretation necessitated by a metainterpreter. We focus particularly on the use of partial evaluation and demonstrate how the control of a general partial evaluator may be simplified in the context of removing a level of interpretation. Section 5 presents a classification of the possible ways in which a metainterpreter can enhance an object program. Section 6 considers the problem of combining the functionality of several metainterpreters into a single one and presents a program transformation technique which accomplishes the combination for two of the three classes of enhancements considered in Section 5. Finally, we conclude with a discussion of directions for future research.

2. METAINTERPRETERS AND FLAVORS

A *metainterpreter* for a language, sometimes called a metacircular interpreter [1], is an interpreter for a language written in that language. It is a special case of a metalevel interpreter where the object language and the metalanguage have been amalgamated. For a discussion of amalgamation in logic programming see [4].

Many different metainterpreters can be written for a given language. An important characteristic of a metainterpreter is the level of computational detail it makes accessible. This characteristic is called *granularity*. The granularity is *coarse* if there is little access, and *fine* if it is detailed.

Throughout the paper the predicate *solve* is used to denote a metainterpreter. The arity of *solve* changes, however, depending on how the metainterpreter has been enhanced. Conventions of Edinburgh PROLOG as laid out in [27] are used.

The simplest metainterpreter calls PROLOG directly. It is defined as

```
solve(Goal) :- Goal.
```

It corresponds to amalgamating object languages and metalanguages by making them identical. The granularity of this metainterpreter is very coarse, allowing little scope for enhancements other than alternative top level shells, as described in [27].

A far more useful metainterpreter is at the clause reduction level. The three clause metainterpreter for pure PROLOG given below is well known. It makes explicit the choice of clause being used to reduce a goal, and the choice of literal to reduce in the resolvent. Unification and backtracking are handled implicitly, relying upon the behavior of PROLOG:

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

Metainterpreters can be written at finer levels of granularity than the clause reduction level. For example, a metainterpreter can model backtracking by keeping explicit stacks, or can perform unification. In our experience, the metainterpreter at the clause reduction level has the granularity most suited for building expert systems.

Our underlying motivation for discussing metainterpreters is how they can be used to build complex programs for applications such as expert systems and program development environments. Many uses of inheritance in the object-oriented programming paradigm are essentially concerned with this issue [23]. Inheritance allows a new object to inherit features of an existing one, supporting reusable software components which minimize redundancy and encourage modularity. Multiple inheritance is an especially powerful form of inheritance in which a new object inherits features from several other objects. It has been most popularized by the FLAVORS system, an object-oriented extension to LISP found on many LISP machines [6, 19, 30]. Flavors allows the functionality of some base flavor to be modified or enhanced by combining it with *mix-in* flavors selected from a library of potential augmentations.

We claim that this style of development is also desirable for expert systems. Where possible, a programmer should be able to pick and choose from a library of stereotypic expert system features and integrate them into his knowledge base. Consequently, we draw heavily on the concepts and terminology of FLAVORS, speaking of "flavored" metainterpreters and of "mixing" a number of metainterpreters into a knowledge base. Rather than intending this as simply a verbal analogy, however, we believe that our use of metainterpreters shares many of the fundamental concerns of inheritance in object-oriented programming. Such basic questions as how one decomposes a given domain into a number of modular components, how one expresses these components so that a given subset of them may be integrated, and how one automates this integration are very much at the heart of our work.

It should be noted that our use of the concepts of object-oriented programming emphasizes different issues than previous work on the integration of logic programming and object-oriented programming. Such work has tended to emphasize the modeling of message passing and state [15, 10, 22, 31], while we have focused on the uses of inheritance to modularly develop complex software systems.

A *flavor* in the context of logic programming is a metainterpreter. Associated with each flavor is the relation it computes, called its *metagoal*, and the program used to compute it, called its *theory*. Flavors are represented accordingly as a ternary relation, *flavor*(*Name*, *Meta Goal*, *Theory*). For reasons that are probably obvious the above metainterpreter is the *vanilla* flavor. Its metagoal is *solve*(*Goal*), and its theory is the three clauses.

A flavor is *enhanced* to have greater functionality or extended performance. For example, the *prooftree* flavor shown in Figure 1 builds a proof tree of the proof of an object goal.

Throughout the remainder of this paper, we will make reference in our examples to an expert system for evaluating credit requests to a bank [2]. Two clauses from this expert system are shown in Figure 2.

```
flavor(prooftree, solve(Goal,Proof),
  [solve(true,nil),
    (solve((A,B),(ProofA and ProofB)) :-
      solve(A,ProofA), solve(B,ProofB)),
  (solve(A,(A if ProofB)) :-
    clause(A,B), solve(B,ProofB))]).
```

FIGURE 1. Enhanced flavor for building a proof tree.

```
credit(Client,Answer) :-
  ok_profile(Client),
  collateral_rating(Client,CRating),
  financial_rating(Client,FRating),
  bank_yield(Client,Yield),
  evaluate(profile(CRating,FRating,Yield),Answer).
financial_rating(Client,FRating) :-
  financial_factors(Factors),
  score(Factors,Client,0,Score),
  calibrate(Score,FRating).
```

FIGURE 2. Credit evaluation expert system fragment.

3. THE PARTIAL EVALUATION OF METAINTERPRETERS

3.1. Introduction

The use of partial evaluation to remove the extra level of interpretation necessitated by a metainterpreter was first discussed by Gallagher [9] and developed by Takeuchi and Furukawa [28]. Conceptually, the partial evaluation of a PROLOG program is simple. A more general program is specialized for a specific use based upon partial information concerning that use (such as partial instantiation of some of the arguments). A metainterpreter may be specialized to a given object program by combining the metainterpreter with the object program and partially evaluating the combined program with respect to a given object goal.

The program transformations which are the basis of partial evaluation use existing techniques. Komorowski [16] describes partial evaluation in PROLOG as consisting of three methods: pruning, forward data structure propagation, and backward data structure propagation. Pruning, the simplest, is essentially only choosing the clauses of a program actually used in a computation. Forward data structure propagation consists in propagating any partial argument instantiations of a goal to its subgoals. Backward data structure propagation involves replacing any calls to a deterministic goal with the body of its associated clause.

Our partial evaluator is similar to that of Takeuchi and Furukawa [28] in its use of the techniques of pruning, forward data structure propagation, and selective unfolding. The last is essentially a generalization of backward data structure propagation in which the requirement of determinism is dropped and a clause containing a goal to be unfolded is duplicated for each clause unifying with that goal.

The result of partially evaluating the *prooftree* flavor and the credit evaluation system with respect to the goal *credit*(*Client*, *Answer*) is shown in Figure 3.

3.2. Controlling Partial Evaluation

A general partial evaluator can be difficult to control, particularly with respect to the decision whether or not to unfold a given goal. In general, our partial evaluator takes advantage of the constraints imposed by the context of its intended use to simplify its control. The primary goal is the removal of the extra level of interpretation necessitated by a metainterpreter. Rather than giving the user the ability to

```
solve(credit(Client,Answer),
    credit(Client,Answer) if
      (Proof1 and Proof2 and Proof3 and Proof4 and Proof5)) :-
      solve(ok_profile(Client),Proof1),
      solve(collateral_rating(Client, CRating),Proof2),
      solve(financial_rating(Client,FRating),Proof3),
      solve(bank_yield(Client,Yield),Proof4),
      solve(evaluate(profile(CRating,FRating,Yield),Answer),Proof5).

solve(financial_rating(Client,FRating),
    financial_rating(Client,FRating) if
      (Proof1 and Proof2 and Proof3)) :-
solve(financial_factors(Factors),Proof1),
solve(score(Factors,Client,0,Score),Proof2),
solve(calibrate(Score,FRating),Proof3).
```

FIGURE 3. Partially evaluating the credit system and the *prooftree* flavor.

control the partial evaluator with explicit declarations, we prefer to give a set of general rules for removing the extra level of interpretation from the combination of object program and metainterpreter.

The top level relation for the partial evaluation is *peval*(*ObjGoal*, *Program*, *NewProgram*). This takes *Program*, which is the concatenation of the object program and the metainterpreter, and partially evaluates it with respect to *ObjGoal*. A new program is produced which is the specialized metainterpreter with the extra level of interpretation removed, but the metaargument structure in place. A compatibility issue immediately arises between the representation of object clauses and the format expected by a metainterpreter. We represent object clauses as *clause*/2 relations to be consistent with their representation in the database.

The first, and most important, control decision is whether or not to unfold a given goal. One extreme would be to unfold every goal, essentially converting the program to a set of ground facts. However, this is an exponential process and is usually not desirable for very large programs. Our partial evaluator provides a declaration, *should_unfold*/1, which specifies that a given goal should be unfolded. However, rather than intending that this declaration be provided by the user for each partial evaluation, we have provided a set of them which seems to be sufficient for removing the extra level of interpretation from the metainterpreters which we have so far considered. For example, metainterpreter calls with conjunctive object goals are unfolded. Figure 4 shows the *should_unfold*/1 declarations we have found sufficient to date.

A second control decision concerns the handling of infinite loops. A computation which terminates at run time can still cause an infinite loop during partial evaluation if an insufficient number of arguments are instantiated. Our system maintains a stack of pending goals, and detects an infinite loop whenever a goal to be partially evaluated is an instance of a goal on the stack. The detection of an infinite loop terminates partial evaluation and returns the goal itself.

Another important control issue is that of the evaluability of goals. The notion of an evaluable goal is useful primarily for simplifying a program resulting from partial

```
should_unfold(Goal) :-
    functor(Goal,solve,N), arg(1,Goal,(A,B)).
should_unfold(Goal) :-
    functor(Goal,solve,N), arg(1,Goal,true)
should_unfold(Goal) :-
    functor(Goal,solve,N), arg(1,Goal,A), system(A).
should_unfold(Goal) :-
    system(Goal).
```

FIGURE 4. Should-unfold declarations.

evaluation. For example, if enough arguments are instantiated in a call to *append/3*, then that call may be performed during partial evaluation rather than at runtime. However, it is an open research issue to determine whether or not an arbitrary goal is sufficiently instantiated to be completely evaluated. Further, while such evaluations can produce more efficient and aesthetic programs, they are not strictly necessary for removing the extra level of metainterpretation.

Because the goal evaluability issue is primarily a matter of aesthetics, we provide no general mechanism for declaring a given goal to be evaluable. However, our partial evaluator handles the evaluability of most system goals, evaluating for example *X is* $0+1$ and *clause*(*solve*(*credit*(*Client*, *Answer*), *Tree*), *Body*) without any explicit declarations from the user.

A final control issue to be addressed by a general partial evaluator is that of open programs. A program is *open* if some of the goals are left intentionally undefined at partial evaluation time. This can be useful, for example, if data for an expert system are to be provided at a later time.

We handle open programs by assuming that a goal which fails during partial evaluation time will also fail at run time, that is, we assume that a system to be partially evaluated is closed. This assumption is motivated by the fact that the partial evaluator is intended to be used to assemble a final, efficient version of a debugged system. However, we are currently examining ways to allow restricted kinds of open programs without requiring an explicit declaration.

To conclude this section, a comparison is given between our partial evaluator and the more general partial evaluator discussed by Takeuchi and Furukawa in [28]. Their approach on each of the four control issues is to provide explicit hooks in the form of user-specified declarations. We look at the nature of the declarations used for each control issue.

In [28], the default behavior is that every goal is unfolded unless a declaration, *inhibit_unfolding*(*Goal*), is present. The onus is on the user to know which goals may cause trouble. Our basic assumption is the opposite. No goal is unfolded unless there is a declaration *should_unfold*(*Goal*), and declarations sufficient to remove a level of interpretation are provided in the system as given in Figure 4.

Infinite loops are detected in [28] by maintaining a stack of pending goals and checking whether a goal being evaluated is an instance of a goal on this stack. Unlike our approach, there is the possibility to nonetheless continue partial evaluation. This is specified by the user with a declaration *expand_loop*(*Goal*). Being able to recursively expand goals is irrelevant for our system, since we are only unfolding one level of interpretation.

Another type of declaration is used in the partial evaluator of [28] to specify evaluability of goals, namely facts of the form *type*(*Goal*,*e*). A *type* fact is also used to handle open goals. The declaration *type*(*Goal*,*t*) terminates the partial evaluation of *Goal*. In our system, there is no classification of goals into types according to how they should be handled by a partial evaluator.

Allowing declarations provides greater flexibility, but also demands more of the user. Our system demonstrates that the necessary declarations can be specified ahead of time, in the context of removing one level of interpretation. The expert system builder need not be concerned with details of the partial evaluator.

4. PUSHING DOWN METAARGUMENTS

A metainterpreter can have two distinct effects upon a given object program. It can affect the structure of the proof for, and indeed the eventual success of, a given object goal, and it can add additional arguments to the proof. We consider each in turn with respect to what needs to be done to mix a flavor into an object program.

The *prooftree* flavor of Figure 1 faithfully reflects PROLOG's standard proof structure. However, more complex proof structures are possible, such as a planner which dynamically reorders conjunctive goals.

Other metainterpreters affect the provability of certain goals. The *askable* flavor, based on the relation *askable* which will be introduced in the next section, for example, prompts the user for a solution to a goal that would fail in the standard PROLOG model of computation. The effects of metainterpreters on the structure of the proof tree for a given object are made explicit by the process of partial evaluation.

The other effect that a metainterpreter can have on an object program is computing extra arguments while solving a given goal. Such arguments are called *metaarguments*. Two flavors which compute a single metaargument are *prooftree*, whose metaargument is the proof tree of the goal being solved, and *count*, whose metaargument is the number of reductions used in solving a goal. The argument-computing effects of a metainterpreter can be made explicit by "pushing down" metaarguments into the object program.

The top-level relation which we use for pushing down the extra arguments in a goal is *push_down_meta_args*(*MetaGoal*, *MetaProg*, *ObjGoal*, *ObjProg*). Input for *push_down_meta_args* is a metaprogram such as Figure 3 and its associated metagoal, in this case *solve*(*credit*(*Client*,*Answer*), *Tree*). The output is an object program and object goal which no longer contain any explicit mention of the metainterpreter but which have been augmented with any additional arguments that the flavor computed. For simplicity, our system assumes that the first argument to every metainterpreter is always the object goal. The result of pushing down the metaargument computed by *prooftree* into the credit evaluation expert system fragment is shown in Figure 5.

Partial evaluation followed by the pushing down of metaarguments makes entirely explicit both effects of a given metainterpreter on an object program. The result of this process is simply another object program, augmented by the functionality of the flavor.

```
credit(Client,Answer, credit(Client,Answer) if
    (Proof1 and Proof2 and Proof3 and Proof4 and Proof5)) :-
  ok_profile(Client,Proof1),
  collateral_rating(Client, CRating, Proof2),
  financial_rating(Client,FRating,Proof3),
  bank_yield(Client,Yield,Proof4),
  evaluate(profile(CRating,FRating,Yield),Answer,Proof5).

financial_rating(Client,FRating,
    financial_rating(Client,FRating) if
  (Proof1 and Proof2 and Proof3)) :-
  financial_factors(Factors,Proof1),
  score(Factors,Client,0,Score,Proof2),
  calibrate(Score,FRating,Proof3).
```

FIGURE 5. Expert system fragment with *prooftree* mixed in.

5. CLASSIFYING METAINTERPRETER ENHANCEMENTS

This section concentrates on how to enhance flavors, building complex flavors from simple ones. The examples are presented to demonstrate the ease of enhancing flavors in logic programming. More examples can be found in [27], with particular emphasis on the application of enhanced flavors for building expert system shells and program debuggers.

There are two ways of enhancing a flavor. Firstly arguments can be added to the *solve* predicate, and secondly its behavior can be changed through the addition, modification, or deletion of clauses or goals within a clause. These two forms of enhancements correspond to the two effects a metainterpreter can have on a given object program: adding extra arguments to the proof of a given goal and affecting the structure of the proof itself.

Enhancements due to additional arguments are further of two kinds. The extra arguments can be used as a structure to be computed while solving a goal, or can be used as a context representing information needed during the computation. We consider each of these three forms of enhancement.

Definition. A flavor $solve_M$ is a *structural enhancement* of a flavor $solve_N$ if (1) each argument of $solve_N$ corresponds to an argument of $solve_M$, (2) corresponding arguments behave identically on identical inputs, and (3) arguments of $solve_M$ not corresponding to arguments of $solve_N$ compute a structure as the goal is being solved.

To facilitate specifying the correspondence, arguments in enhanced flavors are in the same order as the simpler flavor. Additional arguments appear last. This will be true also for contextual enhancements to be introduced below.

The *prooftree* flavor shown in Figure 1 is a typical structural enhancement. A second example is the *count* flavor in Figure 6. Its metagoal *solve*(*Goal*, *N*) is true if *Goal* requires *N* reductions to be solved. It is clear that the behavior of count in

```
solve(true,0)
solve((A,B),N) :-
  solve(A,N1), solve(B,N2), N is N1 + N2.
solve(A,N) :-
  clause(A,B), solve(B,N1), N is N1 + 1.
```

FIGURE 6. Structurally enhanced flavor for counting goal reductions.

solving a goal is identical to *vanilla*. The structure computed is the number of goal reductions.

In general, structural enhancements can introduce arbitrary PROLOG code to maintain the extra arguments, for example the arithmetic additions in Figure 6.

Structural enhancements are normally used to instantiate their extra arguments. The *count* flavor, for example, would instantiate its second argument to the number of goal reductions used in solving a particular goal. Note that the base fact of *count*, *solve*(*true*,0), is ground. This is a typical feature of a structural enhancement.

The general metalevel predicate *demo*(*Goal*,*Proof*) described by Bowen and Kowalski in [4], where *Proof* is the proof of *Goal*, is a structural enhancement of the metalevel interpreter *demo*. In fact, all additional arguments computed by structural enhancements, such as the number of goal reductions, can be regarded as "alternative proofs."

Definition. A flavor $solve_M$ is a *contextual enhancement* of a flavor $solve_N$ if (1) each argument of $solve_N$ corresponds to an argument of $solve_M$, (2) corresponding arguments behave identically on identical inputs, and (3) arguments of $solve_M$ not corresponding to arguments of $solve_N$ carry a context as the goal is being solved.

A typical contextual enhancement of the vanilla flavor is given in Figure 7. The flavor is called *branch*, and its metagoal *solve*(*Goal*,*Context*) is true if *Goal* is solved; that is, *Context* has no declarative relationship with *Goal*. Its intended use is to convey the search tree further in the program. All the flavor does is insist that the associated search trees are logically consistent from one goal to another. The branch flavor can form the basis of a why not explanation facility [26], for example.

The additional arguments for contextual enhancements must be instantiated to an initial context; otherwise an error is likely. The initial context for the branch flavor is the empty list. If the inital context is not instantiated, *branch* will construct an incomplete list.

The third class of enhancements are called *behavioral enhancements*. A behaviorally enhanced flavor extends the computation performed by the flavor being

```
solve(true,Context).
solve((A,B),Context) :-
  solve(A,Context), solve(B,Context).
solve(A,Context) :-
  clause(A,B), solve (B,[A if B|Context]).
```

FIGURE 7. Contextually enhanced flavor for carrying the proof tree branch.

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
solve(A) :- system(A), A.
solve(A) :- askable(A), ask(A).
```

FIGURE 8. Behaviorally enhanced flavor for querying the user.

enhanced. The metagoal of a behaviorally enhanced flavor remains the same. Typically, extra clauses are included in the theory.

Definition. A flavor $solve_M$ is a *behavioral enhancement* of a flavor $solve_N$ if they produce different proofs for identical object goals. A metainterpreter can potentially be behaviorally enhanced by the addition, modification, or deletion of clauses or goals within a clause. Any modification to a metainterpreter which is not a structural or contextual enhancement is a behavioral enhancement. Behavioral modification by side effects, for example tracing a computation on the screen, we regard as a behavioral enhancement.

Figure 8 shows an example of a behavioral enhancement of *vanilla*. The metagoal of *askable* is the same as that of *vanilla*. Its theory, however, contains two additional clauses which handle system goals and ask the user if a given goal is true if it can be proven in no other way. Behavioral enhancement can also arise from the addition or modification of goals within a clause. In general, a behavioral enhancement is any change to a metainterpreter which affects its proof of an object goal.

The forms of enhancement are by no means exclusive. Flavors can be enhanced in several ways at once. For example, in the explanation shell for expert systems described in [26], the metainterpreter for the why not component exhibits all three enhancements. The argument constituting the contextual enhancement is a branch of the proof tree, the argument constituting the structural enhancement returns the proof tree, and the behavioral enhancement adds extra clauses so that the metainterpreter always succeeds even if PROLOG fails.

6. COMBINING METAINTERPRETERS

An important underlying principle of our methodology for building expert systems, and complex programs more generally, is that they should be constructed from simple building blocks. The individual units must be easy to combine and use. If metainterpreters are to be useful as modular components of expert systems, it must be easy to mix several metainterpreters and incorporate their joint effect in an object program.

This section discusses how to combine the effect of different metainterpreters when solving a given object goal. Two strategies for combination, nesting and separative combination, are presented. For each, a metalevel goal or program is given which specifies what is computed. Further strengths and weaknesses of each are discussed.

We suggest there are two major issues relevant for the combination of metainterpreters. The first is whether an efficient program can be derived which incorporates

the effects of the different flavors. More particularly in this paper, we consider whether the mixing technique of Sections 3 and 4 is applicable. The second issue is *commutativity*, that is, whether the combination strategy is sensitive to the order in which the metainterpreters are given.

A minor issue is handling name clashes. In this paper, all metainterpreters are referred to as *solve*, which would cause problems in most current PROLOG implementations. Each metainterpreter could be renamed, or a good module system used. The best solution is to use an extension of PROLOG such as metaPROLOG [5], where theories are treated as genuine objects and can be referred to and manipulated. Each metainterpreter would be a theory.

In this section all metainterpreters are assumed to be structural or contextual enhancements of a common metainterpreter, unless explicitly stated otherwise. Each enhancement has a single extra argument. For example, *prooftree*, *branch*, and *count* are all of this form. Restricting to enhancements with only one extra argument is in fact no loss of generality. Several extra arguments can be grouped into a single structured argument with a technique similar to the pushing down of metaarguments discussed in Section 4.

Combining behavioral enhancements is not discussed in this paper. Different ways of modifying control using behavioral enhancements are not obviously consonant. It is a research issue to find a useful way of specifying behavioral enhancements to allow them to be combined commutatively.

The first strategy for combining metainterpreters is called *nesting*. Suppose $solve_1$ and $solve_2$ are two metainterpreters. The effect of both $solve_1$ and $solve_2$ in solving a given object level query *Goal* is captured with the metalevel query $solve_1(solve_2(Goal, Arg1), Arg2)$.

It is straightforward to mix two metainterpreters into an object program to achieve the effect of a given nested query. The first metainterpreter is mixed into the object program, and the resultant program is the new object program into which the second metainterpreter is mixed. The process can be iterated to mix in several metainterpreters. A discussion of this approach is given in [25] together with some statistics on the speedup of solving the final object program in contrast with going through levels of interpretation.

The nesting strategy is not commutative, however. Consider mixing both *prooftree* and *branch* into an object program. If *prooftree* is mixed in after *branch*, the argument constituting the proof tree will contain references to the context that has been added, which would not be there if *prooftree* had been mixed in before *branch*. The problem is more noticeable with flavors with extra goals such as *count*. If *count* and *prooftree* are mixed, the proof tree may or may not include the arithmetic calculation, depending on the order of mixing in the metainterpreters.

The reason for the lack of commutativity of the nesting strategy is the loss of distinction between object level and metalevel entities. Goals in the unenhanced metainterpreter, called basic goals, and goals introduced to manipulate the extra structure are indistinguishable once a metainterpreter has been mixed into an object program. The partial evaluator cannot distinguish easily between object level and metalevel goals. Although this may be overcome with a more sophisticated partial evaluator taking account of appropriate metaknowledge denoting basic goals, it is preferable to find a simpler form of combination which is commutative.

```
solve(Goal,[Flavor|Flavors],[Result|Results]) :-
    flavor(Flavor,MetaGoal,Theory),
    amalgamated(Goal,MetaGoal,Result),
    solve_flavor(Theory,MetaGoal),
    solve(Goal,Flavors,Results).
solve(Goal,[ ],[ ]).
```

FIGURE 9. Solving goals separatively.

The second strategy is called *separative* combination. The extra argument of each metainterpreter is computed separately in the same context. Figure 9 contains a definition for a predicate *solve*(*Goal*, *Flavors*, *Results*), where *Goal* is the object goal, *Flavors* is a list of flavors to be mixed, and *Results* is a list of arguments computed by each flavor.

The goal is solved sequentially for each flavor. There are two points of interest in the code. Firstly, if the goal being solved is not ground initially, but is instantiated by the first flavor, then later flavors will solve the instantiated goal. This guarantees consistency. Secondly, there is no commitment to the representation of the flavor or the form of the result in the code.

The predicate *amalgamated*/3 is responsible for respecting representational details. More importantly, it communicates information between the object and metalevel goals. An appropriate definition is

```
amalgamated(Goal,solve(Goal,Result),Result).
```

The order of the flavors is unimportant for the separative strategy. They commute with respect to the program in Figure 9. In other words, the argument corresponding to each flavor will be the same no matter in which order it appears.

The code for *solve*/3 solves the object goal as many times as there are flavors. It is preferable to solve a given object goal only once. This can be achieved by generating a metainterpreter which combines the effects of the flavors. The top level code for combining metainterpreters is given in Figure 10. The basic relation is *combine*(*Flavors*, *NewFlavor*), where *Flavors* is a list of flavor names and *NewFlavor*

```
combine([Flavor|Flavors],flavor(comb,solve(Goal,Args),Meta )) :-
    flavor(Flavor,_,_),
    basic_flavor(Flavor,Vanilla),
    flavor(Vanilla,_,VanillaTheory),
    combine([Flavor|Flavors],Args,VanillaTheory,Meta ).
combine([Flavor|Flavors],[Arg|Args],InMeta,Meta ) :-
    flavor(Flavor,solve(Goal,Arg),Theory),
    combine_theories(Theory,In Meta,OutMeta),
    combine(Flavors,Args,OutMeta,Meta ).
combine([ ],[ ],Meta,Meta ) :-
    complete(Meta ).
```

FIGURE 10. Combining flavors separatively.

is a new flavor which represents the combination of all of them. The metagoal of the combined flavor is *solve*(*Goal*, *Args*), where *Args* is the list of arguments which are the enhancements of the flavors being combined.

The code for *combine_theories* is technical but straightforward and is omitted here. It is an example, similar to the pushing down of metaarguments, of a problem that has been sufficiently constrained that syntactic transformations suffice to solve it. The last clause in Figure 10 completes the incomplete structures generated by *combine_theories*.

7. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

The paper has discussed how different metainterpreters can be combined and mixed into a knowledge base. A classification of metainterpreters has been developed to specify the combination. A basic metainterpreter of a given granularity is identified. Three types of enhancements of the basic metainterpreter can then be described: structural enhancements, contextual enhancements, and behavioral enhancements.

A strategy for combining structural and contextual enhancements called separative combination is described. It is best implemented as follows. Given a collection of metainterpreters, combine them to produce a new metainterpreter, and then mix the new metainterpreter into the knowledge base. The mixing in is performed by a restricted form of partial evaluation augmented by an additional step of pushing down metaarguments.

Future work is possible for the three areas of classification, combining metainterpreters and mixing them into the knowledge base. The classification can be sharpened with respect to behavioral enhancements. Hopefully a clearer understanding of the types of behavioral enhancements will enable them to be freely combined with structural and contextual enhancements. For the partial evaluation, the assumptions about expert systems that are used could be more clearly stated, and it could be shown how they affect the partial evaluator.

The example flavors have all been small. We feel, nonetheless, that they are representative of what will be needed to develop large applications. PROLOG programs are concise. A large application should still be developed as concrete proof of the utility of these ideas.

Finally, the relationship to LISP flavors needs to be explored further. Both the separative and nesting strategies of mixing flavors treat the flavors as independent. It is both more interesting and more general purpose to allow interactions between flavors. Indeed, the bulk of work on flavors in LISP is in the development of a language for expressing interaction. The examples in logic programming that we have seen have not needed interaction. The appropriate language or form for expressing interactions for logic programming flavors thus remains essentially academic, but is a topic for research currently under investigation.

This work was partially supported by a core research grant from the Cleveland Advanced Manufacturing Program and the State of Ohio. Arun Lakhotia helped with running some test examples.

REFERENCES

1. Abelson, H. and Sussman, G. J., *The Structure and Interpretation of Computer Programs,* MIT Press, 1985.
2. Ben David, A. and Sterling, L. S., A Prototype Expert System for Credit Evaluation, in: L. F. Paul (ed.), *Artificial Intelligence in Economics and Management,* Elsevier North-Holland, 1986, pp. 121–128.
3. Bloch, C., Source-to-Source Transformations of Logic Programs, M.Sc. Thesis, CS84-22, Weizmann Inst. of Science, 1984.
4. Bowen, K. and Kowalski, R., Amalgamating Language and Meta-language, in: Clark and Tarnlund (eds.), *Logic Programming* Academic, 1982, pp. 153–172.
5. Bowen, K. and Weinberg, T., A meta-level extension of PROLOG, in: J. Cohen and J. Conery (eds.), *Proceedings of the 1985 Symposium on Logic Programming*, IEEE Computer Society Press, 1985, pp. 48–53.
6. Cannon, H. I., Flavors: A Non-Hierarchical Approach to Object-Centered Programming, unpublished paper, 1982.
7. Clark, K. L. and McCabe, F. G., PROLOG: A Language for Implementing Expert Systems, in: Hayes, Michie, and Pao (eds.), *Machine Intelligence 10* Ellis-Horwood, 1982, pp. 455–470.
8. Dincbas, M., Meta control of logic programs in METALOG, in: *Proceedings of FGCS '84,* Tokyo, Nov. 1984, pp. 361–370.
9. Gallagher, J., Transforming Logic Programs by Specializing Interpreters, in: *The Proceedings of the Seventh European Conference on AI*, Brighton Center, UK, 1986.
10. Gallaire, H., Merging Objects and Logic Programming: Relational Semantics, in: *Proceedings AAAI-86*, Philadelphia, 1986, pp. 754–758.
11. Gallaire, H. and Lasserre, C., Metalevel Control for Logic Programs, in: Clark and Tarnlund (eds.), *Logic Programming*, Academic, 1982, pp. 173–185.
12. Hammond, P., micro-PROLOG for expert systems, in: *micro-prolog: Programming in logic,* Prentice-Hall International, 1984.
13. Kahn, K. M., The Compilation of PROLOG Programs without the Use of a PROLOG Compiler, Tech. Report, UPMAIL, Uppsala Univ., Sweden, 1984.
14. Kahn, K. M., Partial Evaluation, Programming Methodology, and Artificial Intelligence, *AI Magazine*, Spring, 1984, pp. 53–57.
15. Kahn, K., Tribble, E. E., Miller, M. S., and Bobrow, D. G., Objects in Concurrent Logic Programming Languages, in: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications,* 1986, pp. 242–257.
16. Komorowski, H. J., A Specification of an Abstract PROLOG Machine and Its Application to Partial Evaluation, Linkoping Studies in Science and Technology Dissertations, No. 69, Software Systems Research Center, Linkoping Univ., Sweden.
17. Mierowsky, C., Taylor, S., Shapiro, E. Y., Levy, J., and Safra, S., The Design and Implementation of Flat Concurrent PROLOG, Tech. Report CS85-09, Dept. of Computer Science, Weizmann Inst. of Science, Rehovot, Israel, 1985.
18. Pereira, L., Logic Control with Logic, in: *Proceedings of the First International Logic Programming Conference,* Marseille, 1982, pp. 9–18.
19. Moon, D. A., Object-Oriented Programming with Flavors, in: *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* Portland, Ore., 1986, pp. 1–8.
20. Shapiro, E. Y. and Safra, M., Meta-interpreters for real, in: *Proceedings of IFIP-86,* 1986.
21. Shapiro, E. Y., Unpublished lecture notes on Concurrent PROLOG, 1985.
22. Shapiro, E. Y. and Takeuchi, A., Object Oriented Programming in Concurrent PROLOG, *New Generation Comput.* 1:25–48 (1983).

23. Stefik, M. and Bobrow, D. G., Object-Oriented Programming: Themes and Variations, *AI Magazine* 6(4):40–62 (Winter 1986).
24. Sterling, L. S., Meta interpreters for Expert Systems, CAISR TR 134–85, Case Western Reserve Univ., 1985.
25. Sterling, L. S. and Beer, R. D., Incremental Flavor-Mixing of Meta-interpreters for Expert System Construction, in: *Proceedings of the Third Logic Programming Symposium*, Salt Lake City, 1986, pp. 20–27.
26. Sterling, L. S. and Lalee, M., An Explanation Shell for Expert Systems, *Comput. Intelligence* (1986).
27. Sterling, L. S. and Shapiro, E. Y., *The Art of Prolog*, MIT Press, 1986.
28. Takeuchi, A. and Furukawa, K., Partial Evaluation of PROLOG programs and its Application to Metaprogramming, ICOT Tech. Report, 1985.
29. Weyhrauch, R. W., Prolegomena to a Theory of Mechanized Formal Reasoning, *Artificial Intelligence* **13**:133–170 (1980).
30. Weinreb, D. and Moon, D., Flavors: Message Passing in the Lisp Machine, MIT-AI Memo No. 602, 1980.
31. Zaniolo, C., Object-Oriented Programming in PROLOG, in: *Proceedings 1984 International Symposium on Logic Programming*, Atlantic City, 1984, pp. 265–270.