An Object-Oriented Inference Engine for PROLOG

Daniel Lanovaz
Duane Szafron
Department of Computing Science, University of Alberta, Edmonton, AB, T6G 2H1 CANADA

This article describes an object-oriented inference engine for PROLOG. The inference engine is part of the Graphically Oriented Development Environment for Logic (Gödel) programming. Gödel incrementally translates source clauses to a persistent clause base in which each clause is an object. The inference engine is a distributed one in which each clause object knows how to unify and execute itself. This means that Gödel supports multiple queries at various points of execution. That is, although multiple queries cannot actually execute concurrently, they can be suspended at any point in their execution and reactivated at any time. This is a major advantage during debugging and exploratory programming. In addition, the inference engine supports primitive clauses written in the implementation language, SMALLTALK-80. This provides a simple interface between PROLOG clauses and SMALLTALK objects that puts the entire SMALLTALK-80 class hierarchy in the hands of PROLOG programmers. Despite these advantages, an object-oriented interface engine has one drawback. In its most general form, the object granularity is too small, so that too many objects must be created and destroyed during program execution. This reduces execution speed. For this reason, several optimizations have been performed in the implementation to consolidate objects to a more reasonable size. With these optimizations, Gödel's interpreter has execution speeds that are still significantly slower than compiled PROLOG but acceptable for exploration, development, and debugging. In exchange, this approach provides integration with the object-oriented paradigm, including all the power that comes from object-oriented programming environments, including browsers, incremental translation, inspectors, debuggers, and multiple query execution, as well as the ability to easily experiment with both the environment and the PROLOG language itself.

KEY WORDS: Logic Programming, Prolog, Smalltalk-80, Object-Oriented, Inference Engine

INTRODUCTION

The basic interpreting algorithm for PROLOG was invented by Colmerauer and Roussel. It was a slow and memory-intensive ALOGL-W implementation which was later improved and rewritten in FORTRAN by Battani and Meloni [1]. Since then, many articles have described various forms of procedural PROLOG implementations and more specifically, interpreting

This is a preprint of a copyrighted article that appeared in the J. Systems Software, Vol. 19, No. 1, September 1992, pp.13-25.

algorithms. As far as object-oriented approaches are concerned, we are aware of two other object-oriented approaches besides the one described here. SMALLTALK/V contains a very simple PROLOG interpreter in its distribution [2], and Levy and Horspool [3] have developed an object-oriented PROLOG-to-C++ translator.

We begin by giving a general description of the main components needed for interpreting PROLOG programs and outlining several key optimizations that have emerged during the evolution of these algorithms. This will provide the background necessary to understand the object-oriented inference engine, its advantages and its disadvantages. Not all PROLOG interpreters are implemented in the manner described in this section; however, there are certain similarities among implementations. It is our goal to summarize these similarities in order to paint a clear picture of the interpretation process. For a more detailed account of PROLOG interpretation and compilation, see Warren [4] or Hogger [5].

The variables and terms in the head of a clause are subject to unification during the execution of PROLOG programs. Unification enables procedures (collections of clauses of which the heads have the same predicate and arity) to provide input and output parameters for answer extraction. It also provides clause selection capabilities based on a powerful pattern matching mechanism. The rules for unifying two objects are:

1. If both objects are atomic (numbers, strings, etc), they are compared for equality
2. If one object is an unbound variable, it is changed to reference the other object.
3. If both objects are unbound variables, the younger one references the older one.
4. If either object is a bound variable, it is dereferenced repeatedly.
5. If both objects are compound, the functors are compared, and each of the terms are unified recursively (for lists, the head and tail are unified).

If any of these tests fail, the unification is said to fail, otherwise a most general unifier (MGU) is returned consisting of a set of variables and the objects with which they were unified.

The workhorse of an interpreter is the resolution inferencing mechanism that acts on the clause database. PROLOG 's inference engine is significantly less complicated than a general resolution theorem prover because of the strict use of Horn clauses and the execution order. Because of this, a description of the interpreting algorithm is straightforward. Given a query to a PROLOG system, the PROLOG inference engine finds a refutation using the steps of Algorithm 1.

Algorithm 1: Pseudo interpreter

1. Try a goal: Search the clause data base for a matching clause head (procedure).

1.1 If found: the goal is unified with the head of a clause and recursively each goal in the body is tried (left to right). If all the body's goals succeed, the goal has succeeded. This position in the data base is noted.

1.2 Not found: the search failed, so backtracking to a previous goal with untried candidates for that procedure is done.

2. Re-try a goal: reset any variables set by unification and search the clause database from the previously noted position.

For each procedure call, an interpreter's stack contains a frame consisting of three parts: a binding environment for variables, an invariant control component, and, in the case of a non-deterministic call (a procedure with more than one untried candidate clause), a nondeterministic control component.

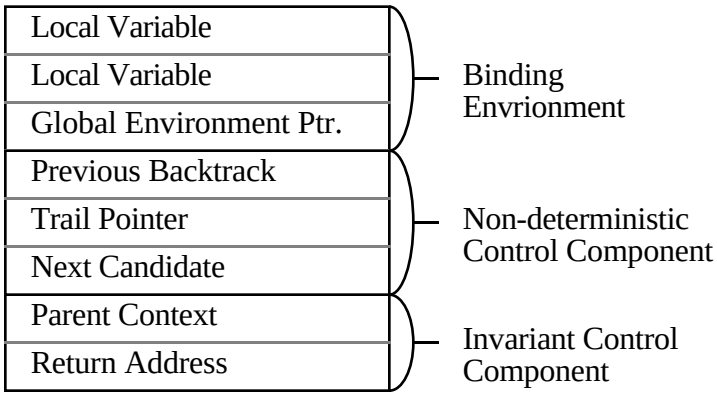

Figure 1 A procedural interpreter context

When a frame is created for a procedure call, a binding environment is created for all variables appearing in the clause containing that procedure. The binding environment has two parts, a local part which is usually part of the frame itself and a global part which is not part of the frame. The global part is for variables which may be referenced after the frame is deleted.

For example, given the clause p(X) <- q(X,Y), r(Y, Z), when the procedure p(X) is called, a context will be created which contains the binding environment for the variables X, Y and Z. These variables are said to be introduced in that context. Given two variables, A and B, A is said to be more recent than B if the context in which A was introduced was created before the context in which B was introduced. Given two unbound variables to be unified, the value pointer of the most recent variable is always bound to the less recent variable so that value pointers always point upwards in the stack. This prevents dangling pointers both during optimization and during backtracking.

In interpreters which use structure copying for term representation, each variable in a binding environment is simply a pointer to a structure or constant. In interpreters that use structure sharing

for term representation, each variable is represented by a molecule which consists of two pointers. The first pointer is a value pointer which points to the value to which the variable is bound (either to another molecule or to a structure in the code). In the case where the value is a structure in the code, the second pointer is an environment pointer which points to the binding environment of the variables appearing in the structure. In the case where the value is not a structured term, the second pointer is not used. For example, if X is bound to the structured term f(Y,Z), then the value pointer of the molecule of X would point to the code for f(Y,Z) and the environment pointer would point to the binding environment for Y and Z.

In fact, the global environment pointer in Figure 1 is only needed when structure sharing is used. When structure copying is used, a copy stack is used to store explicit copies of terms, and the global stack is not required. Some believe that structure copying gives a better locality of reference while structure sharing has the potential of severe thrashing in a paged virtual memory system. However, although Bruynooghe [6] and Mellish [7] have compared the relative merits of each of these approaches and no convincing results have emerged.

The invariant control component contains a pointer to the parent frame and a return address. For example, consider the clauses:

```
p(X) <- q(X), r(X), s(X).
r(X) <- t(X).
```

When a frame, F', is created for the call to r(X), its parent frame will be the frame, F, which was created when p(X) was called. Notice that several other frames were placed on the stack between the frames F and F', since a frame was created for the call q(X) and this call may have resulted in other frames being created as well. The return address for frame F' is the call s(X) since if the call r(X) is successful then s(X) will be the next call executed.

In addition to frames on the run-time stack and global variables in a global stack or heap, a variable trail (V-Trail) stack is also required. A V-Trail stack is used to record the variables introduced before the most recent backtrack frame, but bound in the most recent backtrack frame or in a frame after it. When a backtrack occurs, the V-trail is used to undo the bindings of these variables. For example, consider the clauses:

```
p(X,Y) <- q(X), r(X).
q(a).
q(b).
r(b).
```

When the call q(X) is made, X is bound to the constant a in a frame, Q. However, Q is non-deterministic so it becomes the most recent backtrack frame. Since X is introduced in a frame before Q, the variable X is trailed on Q's portion of the V-trail. Later, when the call r(X) fails and

a backtrack is made to frame Q, the binding of X is removed since it is on Q's portion of the V-trail.

Optimizations to reduce memory requirements include last-call optimization, deterministic call optimization (DCO), and deterministic/non-deterministic frame distinctions. Last call optimization is performed when trying to prove the last goal in a clause. If that goal is deterministic (only has one candidate clause), then after unification the newly created node can be copied over the deterministic node on top of the execution stack. The elimination of the extra frame transforms an otherwise recursive call into a memory-efficient iterative loop. In some situations this saves considerable amounts of memory.

A deterministic computation is a situation in which a PROLOG goal has only one possible unifying clause head. Here the interpreter can perform deterministic call optimization, releasing stack frame storage at computation completion. However, this can only be done if there are no backtrack points (alternate clauses) between the start and finish of the computation.

Another common optimization for PROLOG interpreters is indexing. When the candidate clauses for a goal are being determined by unifying the goal with each clause head in the candidate list, extra information must be recorded if future candidate clauses exist. Naive interpreters do not check to see if the subsequent clauses are permissible future candidates. Indexing determines this by matching the first term of the goal with the first term of each remaining clause head in the candidate list. If future candidates are not possible, there is no need to create a backtrack point. This reduces memory consumption and execution time.

AN OVERVIEW OF GÖDEL

The object-oriented inference engine described in this paper is part of Gödel [8, 9] (Graphically Oriented Development Environment for Logic programming). To understand the interpreter, it is first necessary to understand the overall structure and clause representation in Gödel. From the user's point of view, PROLOG program development can be decomposed into clause editing, interpretation, and debugging. Gödel's architecture is designed so that a highly integrated set of operators (tools) acts on a clause data base so that a uniform view of the system persists throughout these three activities. Our design centers around the concept of a clause data base (persistent heap) that is accessible by all environment components, not just the inference engine. The architecture of Gödel is represented in Figure 2.

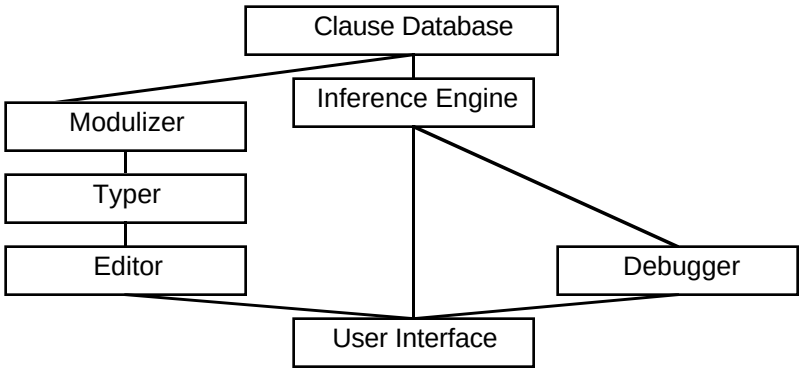

Figure 2: The Architecture of Gödel.

Since we have at our disposal a central repository of program clauses, there is no reason to restrict the number of queries that can be executed at one time. One window may contain a refutation at midpoint, while a second window may contain a completed query. Also, there is no need to reconsult text files when source changes are made because modifications affect the shared repository. For PROLOG , the ability to have multiple queries executing concurrently allows a more natural communication pattern between user and environment. This means that while there is one clause data base, modulizer and typer in Gödel, there can in fact be multiple editors, debuggers and inference engines active at the same time. Although Gödel contains a modulization and typing system, they are not directly related to the discussions in this paper.

In Gödel, access to the inference engine is through a workspace. Since Gödel superimposes a modulization structure on PROLOG , each module has its own workspace. A workspace is a conventional text edit window. Commands are entered, executed, and the results from the execution are displayed within this window. This replaces the usual screen prompt mode of conventional interpreters. In addition, Gödel provides access to its inference engine though its debugger. Figure 3 shows a debugging browser. Additional resources are required in the inference engine to support the debugging features.

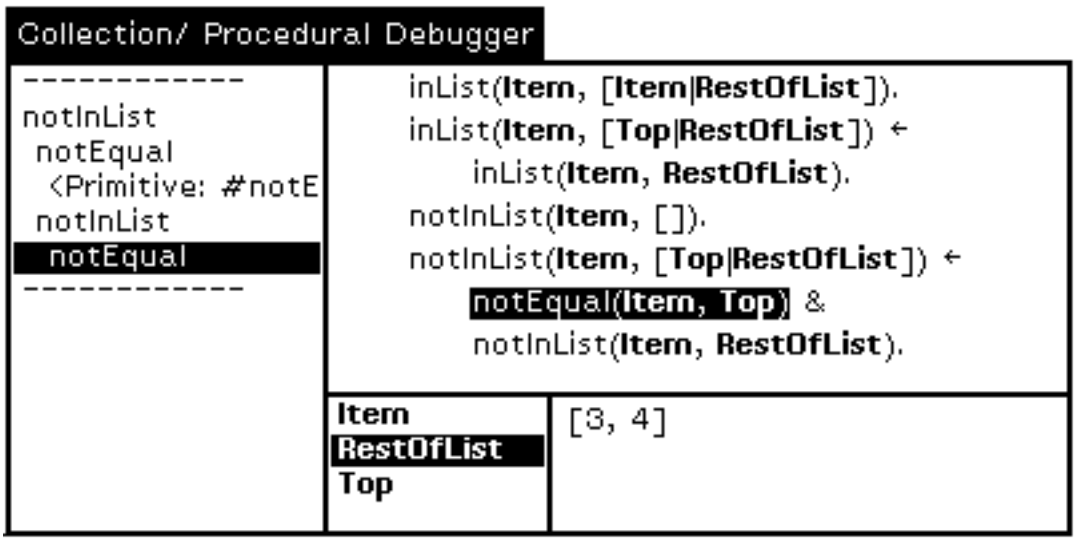

Figure 3: The debugger window

THE CLAUSE DATA BASE

Conventional PROLOG systems maintained the clause data base in a set of files, consulting each file as clauses were needed. Our approach is to view the clause data base as an entity to aid in the development of multiple programs. That is, the data base is shared among programs. There are three major advantages to a central view of persistent clauses:

1) Separate programs and projects can reuse components.

2) There is no need to recompile clauses before they are used. This affects the design of the editor and inference engine, since they not only operate on a clause base that is continually evolving, but they also manipulate compiled clauses instead of the clauses' textual counterparts.

3) A persistent clause data base can be organized to quickly answer questions such as "Find all locations where *<anObject>* is used?" to make development easier.

To understand how the clause data base is represented, it is necessary to understand how the components of clauses are represented.

Atom Representation

In PROLOG, there are two kinds of atomic formula: atoms representing the heads of clauses and atoms appearing in the bodies of clauses. We represent these two kinds of atoms by two separate classes: *GodelLiteralAssertionNode* and *GodelAtomNode* respectively, as shown in Figure 4. Since these atoms share common behavior, we also use a common abstract superclass, *GodelLiteralNode* to implement this common behavior. Each kind of atom includes a predicate symbol and a list of terms which are represented in SMALLTALK by indexed instance variables.

In addition, *GodelAtomNodes* include a reference to the next *GodelAtomNode* in the clause body. In addition to other behavior, each *GodelLiteralNode* contains the behavior necessary to unify itself with another object and to execute itself.

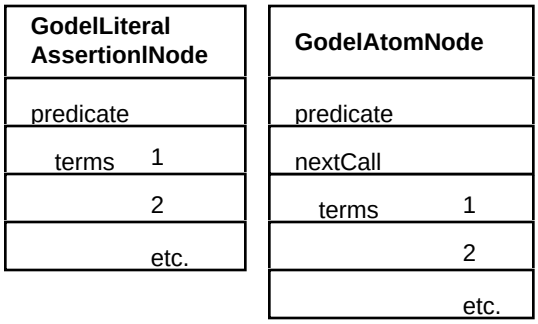

Figure 4: Atom representation

Each term in the term list of an atom is either a function application, a constant or a variable and each of these has its own representation as a SMALLTALK-80 class. Notice that all of these classes are called Nodes. This is because they are actually nodes in a parse tree that are created by a PROLOG parser controlled by the editor. In fact, these nodes form an intermediate representation of clauses, since our final goal is to produce bytecode sequences (similar to SMALLTALK bytecodes) that represent Warren Abstract Machine (WAM) instructions. However, Gödel does not currently support the final translation to bytecodes.

In addition to the components shown in the figures, each object described in this article has other components which are used for editing, displaying and updating them. However, since these components are not germane to this article they have been excluded to simplify the presentation.

Clause Representation

PROLOG clauses can be grouped into four categories based on run-time behavior: standard clauses, primitive clauses, assertions and queries. A standard clause contains a head and a body and is represented by an instance of the SMALLTALK class, *GodelHornClauseNode*. If its head can unify with the current goal atom, it must execute each goal in its body.

A primitive clause is a clause whose body contains source code from a different language and is represented by an instance of the class *GodelPrimitveHornClauseNode*. The different language is usually the implementation language of the PROLOG interpreter, in this case SMALLTALK-80. An assertion is a clause with no body that denotes a fact in the clause data base and is represented by an instance of the SMALLTALK class, *GodelAssertionNode*. Once the unification succeeds, the next goal in the unifying procedure must be attempted. A query contains a body, but no head. Queries are not stored permanently in the clause data base so they will not be discussed here.

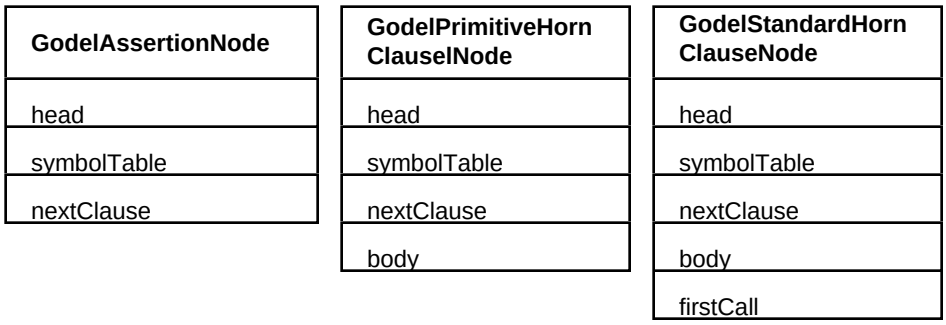

Figure 5: Clause representation

Figure 5 shows the three SMALLTALK classes that are used to represent clauses. In addition, two abstract superclasses are used for abstracting common behavior. *GodelClauseNode* is a superclass of all clause classes and *GodelHornClauseNode* is a superclass of the two non-assertion clause classes. Each clause has a head which is an instance of class *GodelLiteralAssertionNode,* a symbol table which contains all PROLOG variables introduced in the clause and a reference to the next clause in the data base. Assertions require no further information. Nonassertions have a body. In the case of a primitive clause, the body is an instance of the class *GodelPrimitiveNode* which contains the name of a SMALLTALK message. In the case of a standard clause, the body is an instance of the class *GodelFunctionNode* (with predicate $\wedge$) containing the atoms of the body. Standard clauses also cache a direct reference to their first atom for efficiency reasons.

For example, the standard clause:

$p(X,Y) \Leftarrow q(X) \wedge r(Y,X)$

would be represented by the objects shown in Figure 6. Note that Gödel introduces a typing system so that each variable symbol references a type object. However, in the interests of brevity we will not discuss Gödel's typing system here.

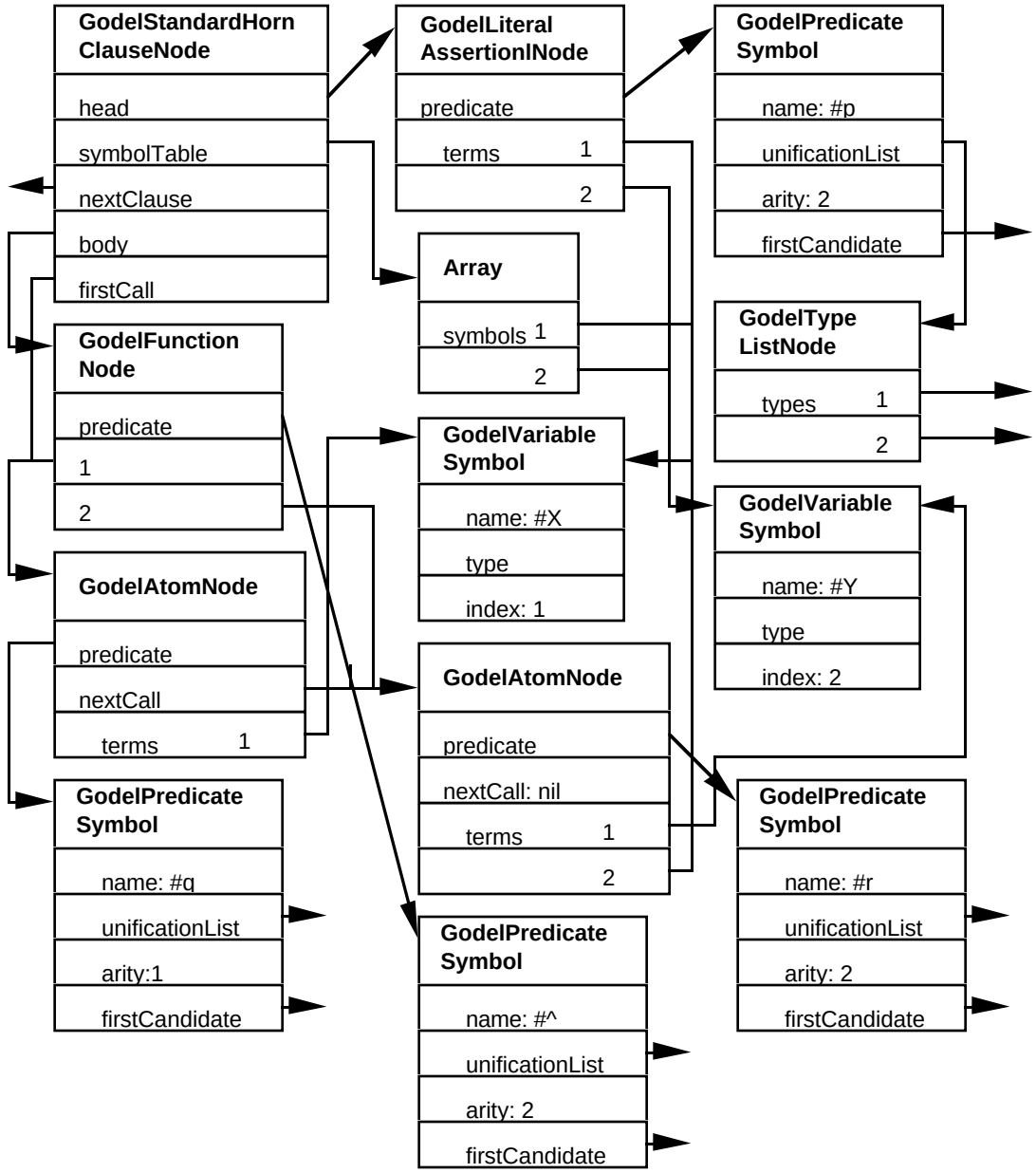

Figure 6: Representation of a standard clause

Optimizations

Several optimizations can be made to the intermediate code. For example, the class *GodelAtomicLiteralNode* could be eliminated. This class is used to encapsulate non-Gödel (i.e. SMALLTALK-80) objects. However, if we wish to completely integrate SMALLTALK-80 with Gödel, we should remove all distinctions between Gödel objects and SMALLTALK-80 objects. If this change is made then unification methods (and some printing methods) would have to be moved to the class *Object*. However, one major benefit would be a clean unification algorithm for arbitrary objects (i.e. a general object pattern-matching algorithm).

Compilation to Byte Codes

Currently, Gödel transforms input clauses into its object representation and interprets them. Compiling the Gödel language into WAM byte-codes would provide two advantages—increased performance and the ability to integrate the SMALLTALK-80 and PROLOG languages at the virtual machine level. Figure 7 shows the similarity between Gödel compiled clauses and SMALLTALK-80 compiled methods. Much more work needs to be done to cleanly integrate the two virtual machines. However, public domain PROLOG WAM compilers are available that can be used in the development of the Gödel compiler.

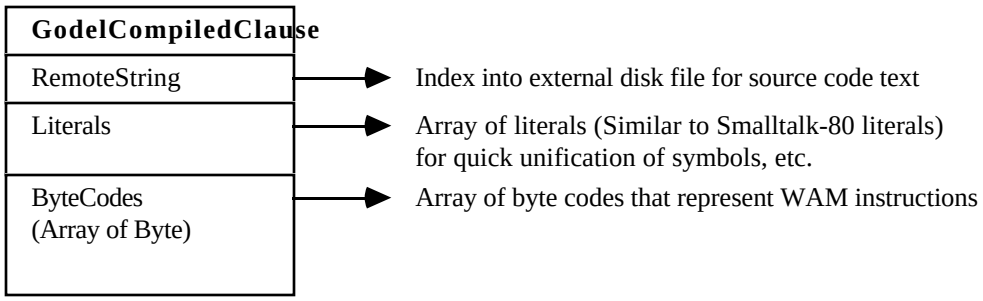

Figure 7: A comparison of Gödel compiled clauses and Smalltalk compiled methods

INTERPRETER OBJECTS

In Gödel, several refutations can exist at the same time in different windows. Typically, the stacks of standard PROLOG interpreters are allocated fixed amounts of storage. When allocated storage is exhausted, a stack overflow error is issued. Therefore, to have more than one refutation executing at a time, each refutation must be allocated an address space and a set of stacks. On a virtual memory system, it is possible to allocate different stacks for each refutation is possible, but preallocation of stacks is inefficient.

To support multiple queries, Gödel adopts the memory management strategy of SMALLTALK-80 so that each interpreter object is allocated space dynamically. Each query interpretation is initiated and supervised by an instance of the class *GodelInterpreter*. That is, multiple queries are simply interpreted by multiple instances of the interpreter class. However, unlike procedural interpreters which implement a central algorithm for interpreting clauses, Gödel distributes the interpretation algorithm among the clauses to be executed and other supporting objects which it uses.

Three classes and their subclasses are used to define the supporting objects used in Gödel's interpreter. The classes are *GodelParseNode, GodelBindingEnvironment* and *GodelFrame*. The *GodelParseNode* class and its subclasses are used to represent clauses and the objects they contain as described in the previous section.

A *GodelBindingEnvironment* is an object that encapsulates a list of molecules. Molecules are binding-context pairs representing run-time variable bindings; they are needed for Gödel's structure sharing term representation. A binding environment responds to messages that retrieve or set a variable's binding. Some PROLOG interpreters categorize variables as local and global. A local variable is allocated space on the run-time stack, while a global variable is allocated space on the global stack. In this way, more variable space can be retrieved during a success exit. At the present time, Gödel makes no such distinction. All variables are allocated in the binding environment local to each stack frame. However, it is straightforward to incorporate this space-saving mechanism. A *GodelBindingEnvironment* is implemented as a subclass of *Array* and each of its indexed instance variables references a molecule.

Gödel divides frames into three distinct classes: *GodelFrame, GodelDeterministicFrame,* and *GodelNonDeterministicFrame.* A *GodelDeterministicFrame* is added to the run-time stack when a goal successfully unifies with the head of a deterministic clause (no other choice points exist). A *GodelNonDeterministicFrame* is created when a goal unifies with the head of a clause and other candidates exist. In this case, backtrack information must be stored to remember those candidates. The superclass, *GodelFrame,* is used only by the procedural debugger. Instances of *GodelFrame* act as placeholders for the representation of the run-time stack within the debugger window.

Figure 8 shows the structure of objects from these three classes. In fact, with the exception of information required for the debugger and the variable *level,* it should not be surprising that the frames contain exactly the same information as for procedural interpreters. These frames serve the same purpose as those in a procedural debugger.

The only difference between Gödel and a procedural interpreter with respect to the frames is that in Gödel they are manipulated as distinct objects with an independent protocol as opposed to being implicit parts of a run-time stack directly manipulated by the interpreter. The variable, *level,*

is bound to an integer that defines the nesting level of the current frame. In a stack based PROLOG interpreter, absolute frame addresses are compared to determine if one frame is lower in the stack than another. To allow for an object-based memory architecture that allows multiple executing queries, each frame must contain a level (or frame depth). Level comparisons occur when determining which variables to push onto the trail stack (i.e. variables whose bindings need resetting during a backtrack operation). The advantage of this approach is that the interpreter can continue executing until all memory is exhausted. Also, this allows the memory manager to compact memory by moving frame objects. An obvious disadvantage is the extra word of memory needed for each frame object. Another disadvantage will be discussed in the Object Granularity section of this article.

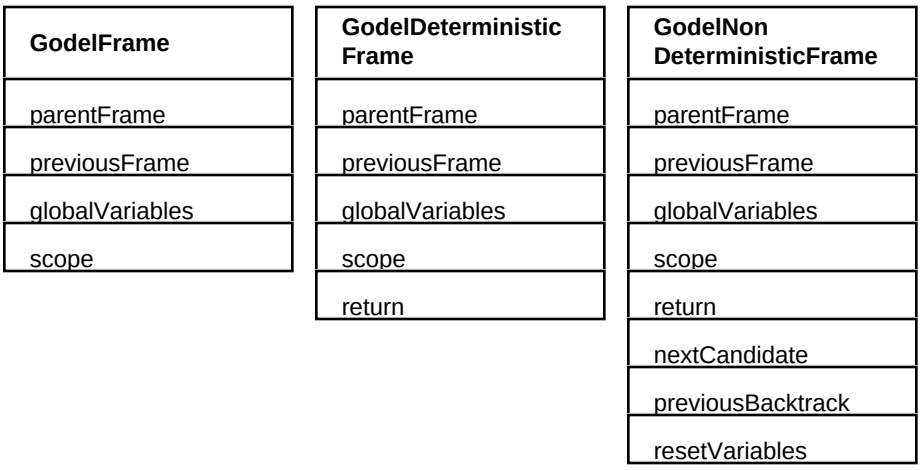

Figure 8: Representation of Gödel frames

The parent frame is the frame of the goal that will be reactivated when the body of the current goal is satisfied. It corresponds to the parent context in the procedural interpreter of Figure 1. The return is the *GodelAtomNode*, which should be called if the current goal is satisfied. It corresponds to the return address of the procedural interpreter. *GlobalVariables* represents a *GodelBindingEnvironment* containing all of the variables for the stack frame. It corresponds to the global environment pointer of the procedural interpreter. The fields *nextCandidate*, *previousBacktrack* and *resetVariables* are also similar to the equivalent variables in the procedural interpreter.

The Gödel frame fields *previousFrame* and *scope* have no counterparts in the procedural interpreter since they are not used by the Gödel interpreter. The previous frame is the frame for the previous atom in the current goal and is used only by the debugger to display the run-time stack. The scope of a frame is the *GodelAtomNode* that created the frame. It is used by the debugger for clause highlighting.

For example, if the active frame is for the call r(Y,X) in the clause:

$p(X,Y) \Leftarrow q(X) \wedge r(Y,X)$

then the parent frame would be the frame for the call p(X,Y) and the previous call would be the frame for the call q(X). In addition, if p(X,Y) was called from the clause:

$u(X,Y) \Leftarrow p(X,Y) \wedge v(Y)$

then return would be the *GodelAtomNode* for v(Y).

As stated earlier, each query interpretation is initiated and supervised by an instance of the class *GodelInterpreter*. Each instance of *GodelInterpreter* contains four instance variables: *currentFrame, currentCall, nextCandidate,* and *mostRecentBacktrack*. The variable *currentFrame* is bound to the instance of class *GodelFrame* which represents the top frame of the execution stack. The variable *currentCall* is bound to the instance of *GodelAtomNode* that is the currently executing goal in the refutation. The variable *nextCandidate* is bound to the instance of *GodelClauseNode* representing the next clause in the procedure whose head predicate matches the current goal. The variable *mostRecentBacktrack* is bound to the instance of *GodelFrame* which is the most recent nondeterministic stack frame.

The variable*CurrentCall* is the pivot point when searching for possible unifying clause heads. The list of clauses that can possibly unify with the current goal (the candidate list) is searched sequentially until a clause head successfully unifies with *currentCall*. At this point, the variable *nextCandidate* is bound to the clause immediately following the matched clause. If there is a next candidate, *mostRecentBacktrack* is set to *currentFrame* before it is updated to the newly created stack frame for the new call. This enables the interpreter to return to this state during backtracking.

UNIFICATION

The unification process described earlier finds the most general unifier (MGU) between two lists of terms. For example, the term *f(g(X), X)* unifies with *f(Y, 1)* producing the MGU *{Y/g(X), X/1}*. In Gödel, each term is represented by a distinct object. Given the representation of variables in a clause as molecules in a binding environment, each term object must respond to the message *#in: Frame unifierFor: anObject in: anObjectsFrame*, where *aFrame* is the message receiver's environment, and *anObjectsFrame* is the environment *anObject* is defined in.

For example, consider the assertion *father(denis, daniel)*, along with the query *father(X, Y)*. When executing this query, the interpreter creates an environment frame f_1. The interpreter then pushes a temporary frame f_2 onto the execution stack and tries to unify *father(X, Y)* with the candidate clause *father(denis, daniel)*. To do this, the interpreter sends the message *[in: f1 unifierFor: [father(denis,daniel)] in: f2]* to the atom *[father(X, Y)]*. If the unification fails the

object *False* is returned; otherwise, *True* is returned and a global unifier (an instance of class Unifier) is set to the collection of bound variables. In this case, *{X/denis, Y/daniel}*.Note that Gödel does not currently perform an occurs check. For example, some of the unification methods are:

```
<GodelClauseNode> in: aFrame unifierFor: anObject in: anObjectsFrame
    "Answer true if my head can unify with anObject, otherwise answer false."

    ^head in: aFrame unifierFor: anObject in: anObjectsFrame

<GodelAtomNode> in: aFrame unifierFor: anObject in: anObjectsFrame
    "Answer true if I can unify with anObject, otherwise answer false.
     I can unify with anObject if we have the same predicate name and
    if our terms can unify"

    (anObject isKindOf: GodelAtomNode)
        ifTrue:
            [(predicate unifiesWith: anObject)
                ifTrue:
                    [termList isNil
                        ifTrue: [^true].
                    ^termList
                            in: aFrame
                            unifierFor: anObject terms
                            in: anObjectsFrame]].
    ^false
```

There are three advantages to the object-oriented approach to unification:

1. The distribution of code among term objects supports the abstraction of each object's behavior (a constant object unifies with another object differently then a variable would).
2. New object types can be investigated without changing the existing interpreter. For example, if a new object (like an array) is added to the language, a unification method that defines how the array unifies with other objects is simply added to the class that describes the behavior of the new object.
3. The existing SMALLTALK object pool can be used. SMALLTALK is equipped with many pre-defined classes. The classes are structured in a tree rooted at the class *Object*. By adding unification methods to the class *Object* and its subclasses, each SMALLTALK object can respond to unification messages from PROLOG structures. Also, since we have defined primitive clauses, whose bodies are SMALLTALK-80 source, the user can write clauses that manipulate SMALLTALK objects.

For example, SMALLTALK's *Collection* classes encapsulate sequences of objects. It is possible to write a predicate *addToList(AnObject, AList)* where *AList* is an instance of a *Collection* instead of a PROLOG list structure. Similarly, the class *View* can be used to create windows that PROLOG code writes (or draws) in, providing a graphical interface for the PROLOG language. Overall, the combination of SMALLTALK classes and primitive clauses enables Gödel to inherit a sophisticated and powerful environment.

One reason for implementing Gödel in SMALLTALK-80 was to provide a development regime that permitted experimenting with environment and interpreter changes. As an example, we added type inheritance into Gödel's unification mechanism. This allowed us to experiment with modifications to both Gödel's interpreter and the PROLOG language.

The addition of type inheritance was chosen for one major reason. Since we have type definitions of the form **TYPE** A **IS** B and **TYPE** B **IS** C, it seemed natural that we modify the unification algorithm so that objects of type A can unify with objects of type C. The change to the interpreter was simple. Each variable instance maintains an extra type field. When a variable unifies with an object, the two types are coerced to the least upper bound of those types. The variable's type is then set to this upper bound.

Adding type inheritance to Gödel required subtle changes to the parser. To declare the type of a variable we extended the syntax of a variable in the terms of a Horn clause so that variables can be followed by an identifier representing their type. For example, if we have a clause *father(X, Y)* such that the variable *X* is of type *Man* and the variable *Y* is of type *Person*, we can express this fact by *father(X : Man, Y : Person)*.

One example of type inheritance that considerably reduced execution time was a PROLOG solution to Schubert's Steamroller problem [10]. A speed-up factor of 10 in execution time (for this particular problem) resulted from the addition of inheritance to the unification algorithm. We will not go into further detail on the benefits of type inheritance in PROLOG, as this has been discussed elsewhere [11].

CLAUSE EXECUTION

Recall that PROLOG clauses are represented by instances of the *classGodelClauseNode* and its subclasses. By separating PROLOG language constructs into classes that model the construct's behavior, we were able to distribute the inference engine among these classes.

Each clause class can respond to the message *#stepIn: anInterpreter*. The single argument, *anInterpreter*, is an instance of a *GodelInterpreter*. When a Horn clause receives the *#stepIn:* message, it performs all necessary operations to execute itself in the context determined by the single argument, *anInterpreter*. The value it returns indicates whether it failed, succeeded, or succeeded and results are to be displayed.

For example, suppose we have the standard definitions of the predicates *grandfather* and *father*, along with the query *grandfather(X, Y)*. The interpreter's *currentCall* is the *GodelAtomNode*, *grandfather(X, Y)*. The message *#stepIn* is sent to the node *grandfather(X, Y)*. Upon receiving this message, a *GodelAtomNode*'s searches for a procedure whose predicate and arity matches its predicate and arity. When a clause is found, that clause's head must be unified

with the current goal atom. If this succeeds, the interpreter updates the run-time stack and sets the *currentCall* to be the new clause's first call (the first atom in its body). The *#stepIn:* message is again sent to the new *currentCall*, *father(X, Y)*. This process continues until the refutation's success or failure.

Primitive clauses are special clauses whose bodies are not a conjunct of atoms, but code written in a different language (usually the interpreter's implementation language). In Gödel's case, this is SMALLTALK-80 code. Figure 9 shows an example of a primitive clause in Gödel.

System / Code / Primitive Clause

head	notEqual(X, Y)
primitive	**notEqual** *"Succeed if X is not the same structure as Y.* *notEqual(X, Y) ← <primitive>"* | x y | x ← currentFrame variableValue: #X. y ← currentFrame variableValue: #Y. x ~= y ifTrue:[↑self] ifFalse:[↑nil]

Figure 9: A primitive clause

Associated with primitive clauses are a collection of methods used to retrieve and set the values of the primitive clause's instance variables. For example, the *notEqual* primitive sends the message *#variableValue:* with argument *#X* to the *currentFrame* of the interpreter to retrieve the value of the variable *X*. Similarly, the message *#variable:value:* binds the variable that is the first argument to the value that is the second argument. These messages, along with the specification of PROLOG term structures as distinct classes with well-defined message interfaces, provides the developer with the necessary tools for manipulating arbitrary PROLOG terms.

There are two advantages to readily accessible and modifiable primitive clause code:

1) existing primitive methods can be specifically tailored by Gödel's users;
2) by using SMALLTALK-80's incremental compilation facility, primitive clauses can be freely added to a PROLOG application.

OBJECT GRANULARITY

The usefulness of a logic programming environment is limited unless multiple independent queries are allowed to coexist. As described previously, Gödel permits many refutations to exist simultaneously, each refutation being allocated its own address space (or stack set). To achieve this, Gödel adopts the memory management strategy of SMALLTALK-80 whereby each interpreter object is allocated space dynamically. However, there are certain problems with creating an efficient implementation for these dynamically allocated objects from within the SMALLTALK-80 environment itself. This section briefly describes the problems associated with creating dynamic stack structures. Emphasis is placed on the tradeoffs between object size, object numbers, and execution efficiency.

All components of the SMALLTALK-80 system are represented by objects: numbers, strings, queues, dictionaries, text editors, programs, computational processes, and many other entities. However, a fundamental design problem emerges regarding Gödel's run-time stack structures. At what scope (or granularity) should inference engine (stack) objects be created so as to gain the greatest advantage in terms of object pointer allocation, execution efficiency, and exploratory programming flexibility? We claim that the consistent SMALLTALK-80 object-based approach to information representation provides the greatest flexibility in terms of exploratory programming, but the conventional machine word-oriented stack structure provides the best avenue for execution efficiency. Therefore, the problem lies in finding a way to combine the preallocated machine-word stack structure with dynamically allocated stack objects.

Three implementations were considered for Gödel's run-time stacks: a machine-word/object stack, a segmented stack, and a linked object stack. The machine-word stack is equivalent to a contiguously preallocated array of machine words. Collections of logically related word-size objects are pushed and popped from the stack. The onus is on the stack user to logically define object boundaries. Two disadvantages to this approach are that the logically connected machine words cannot be manipulated as a single object and the stacks are fixed in size during initialization. However, three advantages to this approach are a significant decrease in the number of object pointers the SMALLTALK-80 system must allocate for each logical object placed on the stack, several optimization techniques such as overlapping object boundaries may be more easily implemented, the ability to randomly access any stack location.

For example, recall Figure 1. It displays the individual components of a typical Gödel non-deterministic frame. Each frame contains the following fields: a local variable list, a global environment pointer, a nondeterministic control component, and an invariant control component. These fields are placed on the machine-word stack by the interpreting algorithm when creating a nondeterministic frame. The following code shows a simplified implementation of this operation:

```
localStack push: parentContext; push: returnAddress;
      push: globalEnvironmentPtr; push: previousBacktrack;
      push: trailPointer; push: nextCandidate;
      push: localVariable1; push: localVariable2; ... push: localVariableN
```

In contrast, an object stack is an implementation that defines the logical object boundaries in the Smalltalk-80 system by the class of the objects. Each object placed on the stack is an instance of this defining class and therefore can only be manipulated as a logical group. The advantage of this approach is that system code (such as the debugger) can more easily manipulate the stacked data. A disadvantage is the increased number of objects and object pointers that are created (see figure 10). The following code is a simplified implementation of pushing a nondeterministic frame object onto the stack:

```
aNonDetFrame := NonDeterministicFrame new: lengthLocalVars.
      aNonDetFrame parent: parentContext; return: returnAddress;
      globalVariables: globalEnvironmentPtr;
      previousBacktrack:  previousBacktrack;
      trailPointer: trailPointer; nextCandidate: nextCandidate.
localStack push: nonDeterministicFrame
```

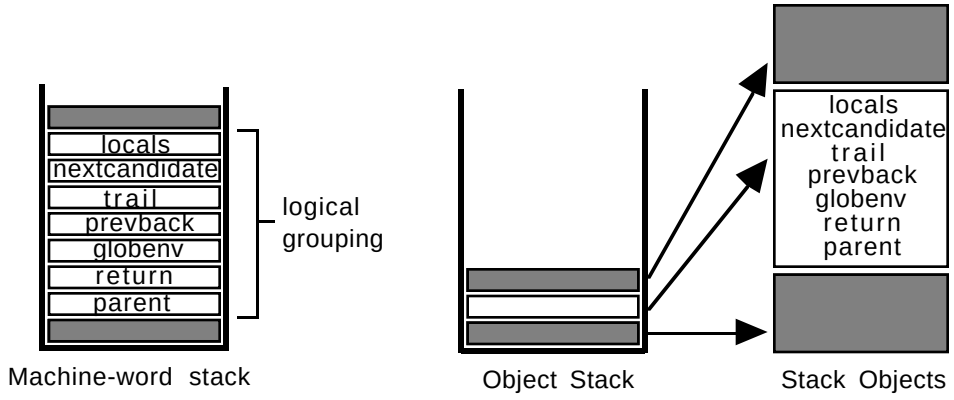

Figure 10: Machine-word/Object stack structures.

As we mentioned previously, Gödel permits multiple execution threads to exist simultaneously. This reveals two serious problems in the previous stack implementation: the requirement that a suitably sized contiguous stack segment be preallocated, and the inability for the stack to expand/shrink past its originally allocated size. An alternative implementation is a segmented stack. This stack organization is shown in Figure 11. The stack is not contiguous in memory, but consists of several segments linked together to form a sequential "super-segment". The topmost segment is owned by the stack object itself. Each segment contains four instance variables. They are used to determine segment address ranges. The following algorithm is used to push/pop information:

1. When an object is pushed onto the stack, if the top segment overflows, a new one is allocated.

2. When an object is popped, if the top segment underflows, it is released (and subsequently garbage collected). If the *previousSegment* of the top segment is null, an error occurs.

The main drawback to this approach is the multiple pointer indirection required for each stack access.

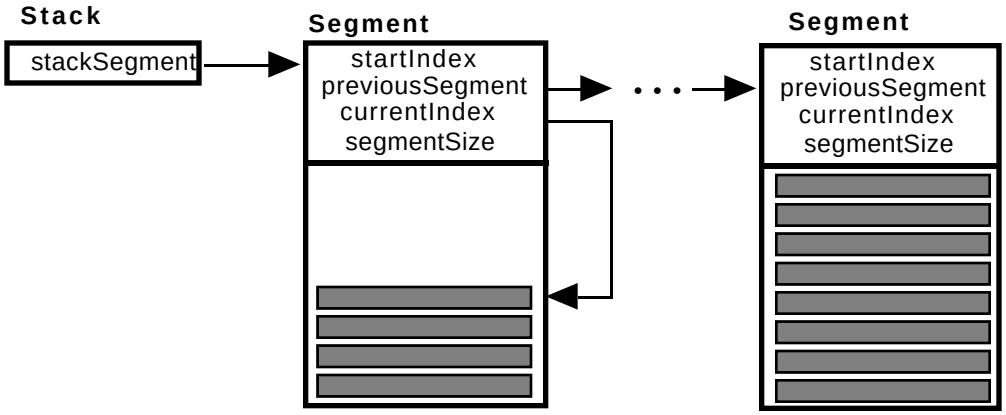

Figure 11: Segmented Stack

The final approach, and the one implemented in the current Gödel system, is to dynamically link stack objects (see Figure 12). Each stack object contains a *nextObject* instance variable in addition to application-dependent information. As the stack grows, new stack objects are allocated and linked within the stack.

Unfortunately, due to Gödel's heavy use of the run-time stack, it was found that a considerable amount of time was being spent within the SMALLTALK-80 memory allocator and garbage collector. This reason for this is as follows: as a refutation proceeds, nondeterministic and deterministic frames are allocated and pushed on the stack. Each time a subgoal fails, the interpreter must backtrack, freeing all frame objects from the stack top to the most recent backtrack point. This process requires a considerable number of objects to be allocated and subsequently freed for garbage collection.

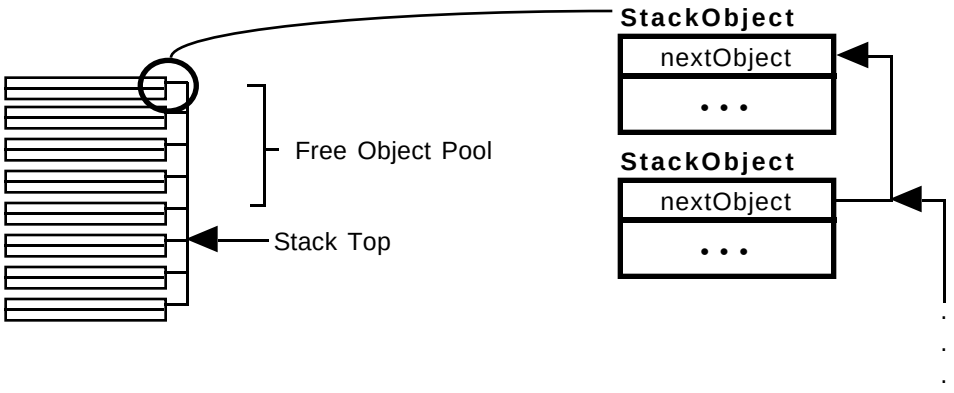

Figure 12: Linked Stack With Free Object Pool

Since object allocation and garbage collection are relatively expensive operations, the following optimizations were performed:

1. For each query, preallocate an object pool for each stack object class. In this case, pre-allocate an object pool for deterministic frames, non-deterministic frames, and binding environments.
2. When allocating a new stack object, extract an unused object from the object pool, if there are no more objects in the pool, allocate more.
3. When freeing a stack object, do not release it for garbage collection, return it to the free object pool. Only after the average stack usage size has stabilized are the extra stack objects (from the stack top to the object pool top; see Figure 12) freed for garbage collection.

The above implementation is necessary because access to the SMALLTALK-80 virtual machine is restricted. Ideally, each active process (query) would be associated with a set of stacks. As with the SMALLTALK-80 virtual machine, these stack objects would be allocated in a special stack zone. Also, each stack object would come in two formats: standard object format and frame format. The frame format would be a highly optimized version of the standard frame object format. If an object tries to send a message to a frame format object (for example, the debugger), it would be converted to standard object format and moved from the stack zone to object memory transparent to the user [12].

In conclusion, several tradeoffs were considered when implementing Godel's run-time stack structures: object count, object size, execution efficiency, and exploratory programming flexibility. It was discovered that object pooling reduced the amount of memory allocation and garbage collection, subsequently decreasing execution time. Unfortunately, the linked object stack introduced memory overheads.

In the end, the execution speed of Gödel is about 40% of the speed of the procedural stack-based Waterloo UNIX PROLOG [13] on small benchmarking programs [14] such as concat, naive reverse, queens, etc.

CONCLUSION

We have described an object-oriented inferencing engine for PROLOG in which each clause object knows how to unify and execute itself. We have shown how this distribution makes it easy to support multiple queries in various states of execution and indicated how to extend PROLOG to include new objects. For example, arrays may be added by defining unification methods in an *Array* class.

In addition, we have shown how to support primitive clauses written in the implementation language, SMALLTALK-80, thus creating a simple interface between PROLOG clauses and SMALLTALK objects. This interface puts the entire SMALLTALK-80 class hierarchy in the hands of PROLOG programmers. Finally we have described the major efficiency problem with object-oriented interpreters—object granularity—and we have provided a solution to this problem.

ACKNOWLEDGEMENT

This research was supported in part by research grants #OGP8191 and #INF36861 from the National Sciences and Engineering Research Council of Canada.

REFERENCES

[1] G. Battani and H. Meloni, Interpreteur du language de programmation PROLOG, Research Report, Artificial Intelligence Group. University of Aix-Marseille, Luminy, France 1973.

[2] *Smalltalk/VPM*, Digitalk, Los Angeles U.S.A., 1989.

[3] M. Levy and R.N. Horspool, Translation of Prolog to C++, Internal Report, University of Victoria, Victoria, B.C., Canada, June 1990.

[4] D.H.D. Warren, Implementing PROLOG-Compiling Predicate lLgic pPograms, Research Reports Nos. 39 and 40, Department of Artificial Intelligence. University of Edinburgh, Scotland, 1977.

[5] C.J. Hogger *Introduction to Logic Programming*, Academic Press, New York, 1984.

[6] M. Bruynooghe, The Memory Management of Prolog Implementations, in *Logic Programming*: (K.L. Clark, and S.A. Tarnlund, eds.). Academic Press, New York, 1982.

[7] C. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, in *Logic Programming*: (K.L. Clark, and S.A. Tarnlund, eds.). Academic Press, New York, 1982.

[8] D. Lanovaz, Gödel: A Prototype Prolog Programming Environment, Master of Science Thesis, University of Alberta, Edmonton, Alberta, Canada 1988.

[9] D. Lanovaz and D. Szafron, Gödel: An Interactive Incremental Logic Programming Environment, *Technical Report 90-17*, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, May 1990.

[10] D. Szafron and F.J. Pelletier, Some Notes on Prolog Technology Theorem Proving, Technical Report TR89-10, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, March 1989.

[11] A. Mycroft and R. O'Keefe, A Polymorphic Type System For Prolog, *Artificial Intelligence* 23, 295-307, (1984).

[12] *Objectworks Advanced User's Guide*, ParcPlace Systems, Mountain View, California, 1989, pp. 6.4 - 6.7.

[13] M. Cheng, Design and Implementation of the Waterloo Unix Prolog Environment, Master's Thesis, University of Waterloo, Waterloo, Ontario, Canada, 1984.

[14] T. Dobry, A High Performance Architecture for Prolog, Technical Report UCB/CSD 87/352, Computer Science Division, University of California, Berkeley, California, 1987.