University of Alberta

GÖDEL: AN INTERACTIVE INCREMENTAL LOGIC PROGRAMMING ENVIRONMENT

by

Daniel Lanovaz
Duane Szafron

Technical Report TR 90-17
May 1990

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Gödel: An Interactive Incremental Logic Programming Environment

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

SUMMARY

This paper describes Gödel, an interactive incremental logic programming environment. Gödel supports programming in "standard" Prolog. However, Gödel uses its environment to superimpose a module structure and an inheritance typing system which can be used to support modern software engineering strategies. In addition, Gödel presents the user with a uniform user interface for browsing, editing, executing and debugging which simplifies context switches between these activities and reports errors as they occur. The most important and novel aspects of Gödel are its introduction of software engineering support without changing the language, its incremental translation of clauses to a persistent clause base as the clauses are entered, its efficient object-oriented inference engine written in Smalltalk-80 and its primitive clauses that provide a seamless interface between Prolog and Smalltalk-80.

KEY WORDS: Programming Environment, Logic Programming, Prolog, Smalltalk-80, Interactive, Incremental

INTRODUCTION

Development of logic programming interpreters and compilers has been under way for many years. There are five important research components to this development: execution speed, portability, language features, software engineering support and user interfaces.

Goals of portability and efficiency pose formidable problems in Prolog interpreter designs. Pioneering work by Warren in the design of the "W*arren Abstract Machine"* (WAM)[1] proved that Prolog could be executed in time and space equivalent to other symbolic languages. However, further efficiency gains are necessary for Prolog to become the implementation language of the Japanese initiated Fifth Generation Computing Systems (FGCS) project[2]. To increase efficiency, VLSI technology can be used to implement high level computer architectures for logic programming[3,4].

Generally, it appears that standard sequential Prolog implementations have become as efficient as possible. Therefore, to increase performance further, a movement from sequential to parallel implementations is gaining momentum. These implementations take advantage of the inherit parallelism available in logic languages. Although such work is interesting, in this paper we wish to deviate from the issue of fast portable logic inference engines and look in more detail at the way in which a programming environment may be used to superimpose new language features on Prolog, enhance Prolog software engineering support and provide a consistent user interface during development.

In contrast to developing faster Prolog interpreters, environment research is intended to improve programmer productivity. Increased computing power in the form of high performance workstations equipped with bit mapped displays and pointing devices has directed programming environment research towards graphical user-interactive techniques. Graphical workstations have shown the usefulness of graphical human-computer interfaces in increasing developer productivity. The result is the integration of programming languages and the computer's graphical environment exemplified in, among others, the Smalltalk-80[5] , Mesa[6], and Interlisp-D[7] systems.

However, very little work has been done in studying the application of programming environments to logic programming. This paper describes **Gödel**: a **G**raphically **O**riented **D**evelopment **E**nvironment for **L**ogic programming[8]. Gödel is a marriage between a conventional batch oriented logic programming language (Prolog) and a user-interactive development environment using an object-oriented design strategy. Since Gödel was developed to experiment with ways of making a more logical, appealing and useable environment, we swayed from the compelling force to extend or modify the Prolog language directly, or increase the interpreter's execution speed. We took the approach that if Prolog is a language to be taken seriously, it must be complemented with a powerful and useable environment.

In this paper, we describe each aspect of our environment including the clause database, modulizer, typer, editor, debugger and the interpreter. We argue that in many cases a programming environment can be used to augment Prolog's language features to provide better software engineering support without changing the language. We also point out those situations where language changes would be necessary. Finally, we also argue that the object-oriented design methodology used in both the graphical interface and the inference engine provides a suitable foundation for a logic programming environment's architecture.

The Architecture

A software development environment is a collection of integrated tools used to design, code, and maintain a software system. Software development environments can be characterized in two ways: the scope of the individual components and the integration (communication) between components. When the scope is limited to software coding, execution and debugging, the environment is often called a programming environment. Gödel can be characterized as a programming environment with highly coupled components. This section is a description of Gödel's architecture. That is, this section names the components of Gödel and describes how they communicate.

The Design Methodology

The architecture of a software system is based on the methodology used to design it and the programming language used to implement it. For example, the architecture of a software system which is based on an object-oriented design and implemented in Smalltalk-80 will be quite different than a software system based on transactions and implemented in C.

There are two basic approaches to designing and implementing a programming environment. One approach is to design and implement the environment using the paradigm and language that the environment is supposed to support. The InterLisp and Smalltalk-80 programming environments are examples of this approach. The major advantage of this approach is that extensions and modifications to the environment are often as easy as application programming.

However, designing and implementing a Prolog environment in Prolog has two major disadvantages:

1) Prolog is a relatively new and evolving programming language and logic programming is a relatively new design paradigm. Much needs to be learned about the suitability of logic programming for certain features which are fundamental in programming environments. For example, very little work has been done on designing and implementing graphical interactive interfaces.

2) In general, Prolog programs do not execute as quickly as programs written in more conventional languages. Although the lack of speed might only be an inconvenience when it comes to the implementation of the inference engine, it would be disastrous for implementing the graphical user interface.

The second approach is to design and implement the programming environment using a different paradigm. The most obvious approach is to follow the lead of existing Prolog interpreters and use a procedural design implemented in C. However, we have chosen another option. Gödel has been designed using the object-oriented design methodology and implemented in Smalltalk-80. There are several apriori advantages to this approach:

1) Object-oriented programming is designed for modular, reusable, modifiable code. During experimentation, the ability to make changes and enhancements to the source through class hierarchy changes is an asset.

2) The object-oriented methodology is well suited for graphical, window oriented applications[9, 10, 11]. This allows for experimentation with the interaction between graphics and logic programming.

3) It is possible that the object-oriented design of a Prolog inference engine may provide new insights into implementation strategies.

4) Object-oriented languages provide inheritance of class definitions. Similarly, taxonomic hierarchies have been used in knowledge representation which is an important application of logic programming. As in LOGIN[12], this approach may provide an alternate basis for incorporating inheritance into logic programming.

The two major disadvantages of our approach are:

1) In the long term, logic programming environments should be written in logic programming languages for consistency and extensibility. The object-oriented approach does not directly contribute to this goal.

2) Object-oriented interpreter implementations may be slower than conventional procedural designs.

We believe that the advantages of an object-oriented design outweigh the disadvantages at this time.

The Components

The architectures of existing Prolog interpreters consist of three components: a clause editor, an interpreter/compiler and a clause database. This architecture is standard among all logic programming systems, but interaction between components has generally been kept to a minimum. For example, in the Unix environment, the clause editor is one of the available editors such as Vi or Emacs. The interpreter is a completely separate software component that transfers clauses from an edited file (and/or from the user) into the clause database, accepts user queries, and outputs results. During the debugging phase, if an error is found in a clause description, the developer must exit the interpreter, enter the editor to make changes, re-enter the interpreter to reload the clause definitions into the database, and lastly repeat the query and continue debugging. The number of task context switches is numerous, and any reduction would decrease development overhead.

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 database so that a uniform view of the system persists throughout these three activities. Our design centers around the concept of a clause database (persistent heap) that is accessible by all environment components, not just the interpreter. The architecture of Gödel is represented by figure 1.

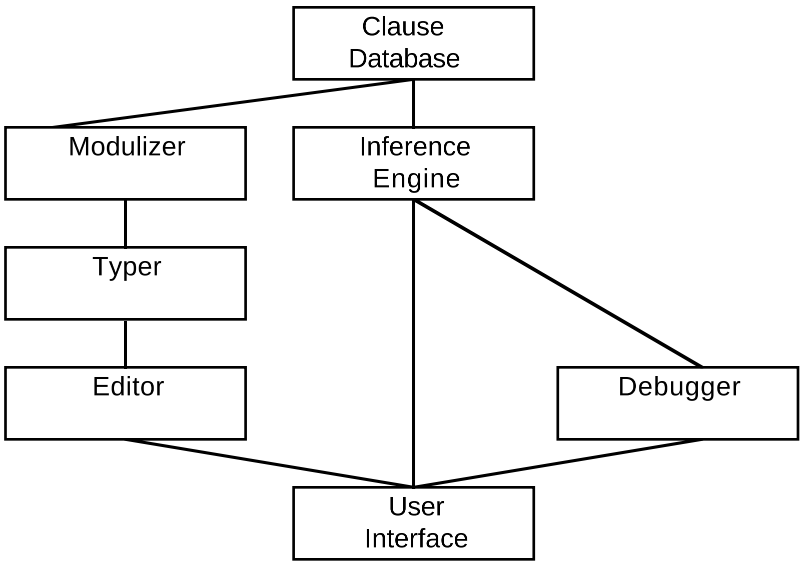


Figure 1: The Architecture of Gödel.

Gödel provides five services:

1) Gödel provides graphical interactive incremental syntax directed editing with static semantic checking.

2) Gödel supports modulization.

3) Gödel supports typing.

4) Gödel supports graphical source level debugging.

5) Gödel's inference engine provides access to the full power of the underlying Smalltalk-80 programming language.

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 re-consult text files when source changes are made because modifications affect the shared repository. For Prolog, the ability to have multiple queries in various stages of execution allows a more natural communication pattern between user and environment. This means that while there is one Clause Database, Modulizer and Typer in Gödel, there can in fact be multiple Editors, Debuggers and Inference Engines active at the same time.

THE CLAUSE DATABASE

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

1) Separate programs and projects can re-use components.

2) There is no need to re-compile 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 database can be organized to quickly answer questions like: "Find all locations where <*anObject*> is used?" to make development easier.

Since Gödel is implemented in Smalltalk-80 each clause is an instance of a class and common state and behavior are abstracted using a class hierarchy. The atoms and their components (predicates, terms, variables, constants etc.) are also represented by objects. In addition, Gödel supports primitive clauses. A primitive clause is a clause whose body is replaced by source code of a different language. This language is usually the implementation language of the Prolog interpreter - in this case Smalltalk-80. A more detailed description of the clause database appears elsewhere[13], but figure 2 shows the representation of the standard clause: $p(X) \Leftarrow q(X) \wedge r(X)$.

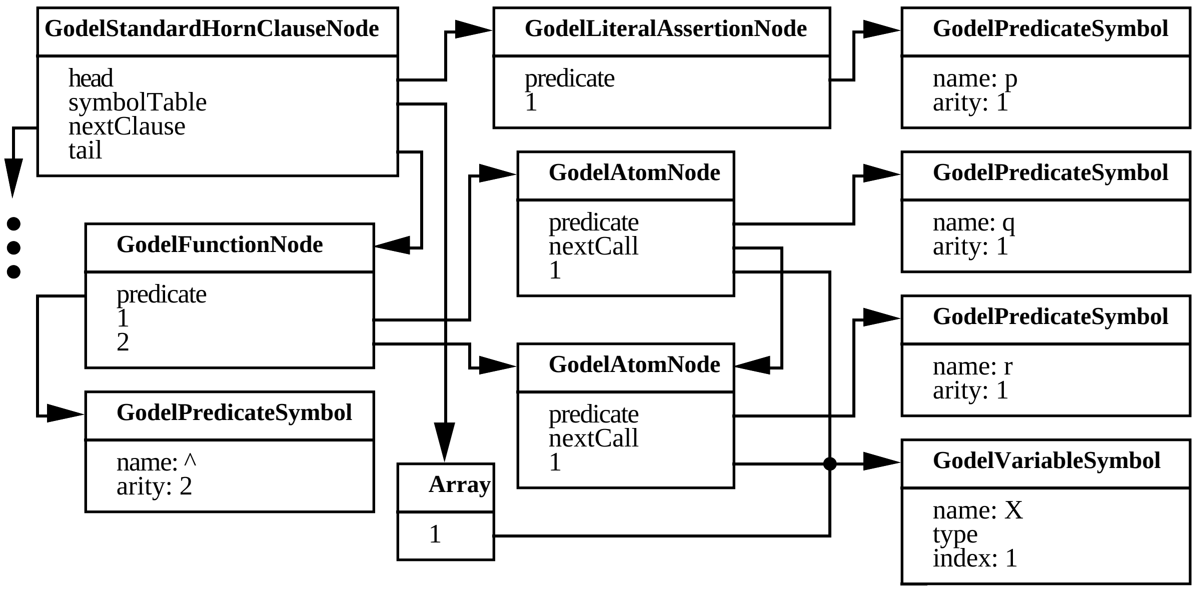


Figure 2: Intermediate code for a standard clause

Currently, Gödel transforms input clauses into an intermediate form 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 language at the virtual machine level. 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 .

The clause database is modified through the modulizer and typer. We believe that both modulization and typing are necessary if Prolog is to be usable for large software engineering projects. However, it is highly advantageous to provide a system where both enhancements are unobtrusive and introduced in such a way that conventional Prolog programs can be run with minor or no modifications!

THE MODULIZER

Modulization is an abstraction technique in which programs are decomposed into many independent modules which communicate using small, well-defined interfaces. The advantages of modulization are well known[14]. For example, modules provide scoping limitations so that names used locally in a module can be used independently in other modules. They limit the propagation of code changes since changes to the implementation of a module do not affect any other modules. They also provide portability and a good mechanism for isolating errors.

Modulization by Environment

Gödel introduces modulization to Prolog by superimposing a module structure on the language without modifying the language itself. Introducing modulization through the environment has three distinct advantages over creating a new language or modifying Prolog. They are:

1) Since Prolog is widely used, there is a large body of existing code. Since code is often re-used, a new language would lose this library of existing code.

2) The use of modulization in Prolog is a relatively new concept and is not well understood. It will probably take some time before one modulization technique is recognized as superior. Thus, it is easier to experiment with an environment.

3) There is no common syntax for Prolog, but a single environment can be configured to generate code for any dialect.

Existing module systems for Prolog are mainly concerned with the division of the predicate name space. The problem is that these systems are based on the syntactic view of modules (i.e. the compilers symbol table) instead of the view that modules have a connection with the language's semantic theory. Therefore, modulization should be geared towards constructing logic programs, and must diverge from the standard view used by languages like Ada or Modula-2. The approach taken in Gödel is loosely based on O'Keefe's[15] algebraic formalization of a module system where various operators are used to construct programs out of pieces. These operators allow varying degrees of communication between modules.

Module Communication

To define module communication in a Prolog programming environment it is first necessary to understand the concept of clause extension. Gödel's interactive environment relies heavily on incremental translation. This affects not only the design of the clause database and inference engine, but also the interaction between a developer's actions and Prolog source code. We define incremental translation in Prolog as the act of *extending* the clauses in a predicate or procedure. For example, if a procedure consists of the clause $p(X) \Leftarrow p(1) \wedge p(2)$ and the assertion *p(1),* then adding the assertion *p(2)* extends the procedure *p* to include this new clause.

In Gödel, there are three distinct ways to combine modules: union, open inclusion and closed inclusion (see reference 8 for a formal definition of these operators using the formalism of O'Keefe). Module combination is always performed from within one module by specifying the name of the other module to be used in the operation. Let M_1 represent the current module for the operation and M_2 denote the other module involved in the operation.

A *union* operation on module M_1 using module M_2 adds all clause definitions from M_2 into M_1 and adds all clause definitions from M_1 into M_2. In addition, any clause definitions subsequently added to either module are added to both modules. This operation has a counterpart in conventional Prolog systems where M_1 and M_2 are simply two disk files containing clauses. In such a Prolog, union is obtained by performing two operations: *consult*(M_1) and then *consult*(M_2). Of course, any subsequent extensions made by the user affect only the current database and do not get written to the disk files.

A *closed inclusion* operation on module M_1 using module M_2 adds all public clauses (each clause in a module is declared to be public or private) from M_2 into M_1. Note that all procedures in M_1 and M_2 must have different names. If the same procedure name is used in M_1 and M_2, they are considered different procedures and Gödel will prompt the user to change the name of one of the procedures during the open inclusion operation. After closed inclusion, none of the included procedures can be extended in M_1. This is similar to an import declaration in procedural languages where M_1 wants to call procedure P in M_2, but the implementation details of P are hidden. Closed inclusion mimics the use of an abstract data type module. For example, assume we have a module *Set* containing the type Set, and various predicates that operate on sets like *createSet/1*, *addToSet/3*, and *removeFromSet/3*. A module with imports the *Set* module using closed inclusion cannot add definitions to the Set predicate or know the internal representation of a Set (is it a list, or a functor such as *set(List)*?). Closed inclusion enforces this information hiding.

An *open inclusion* operation on module M_1 using module M_2 adds all public clauses from M_2 into M_1. As with closed inclusion, all procedures in M_1 and M_2 must have different names during the inclusion operation. However, after open inclusion, any clauses subsequently added to M_1 will extend the definitions of M_2. In contrast to closed inclusion, open inclusion permits the user to extend procedures. For example, in the current module, the user may wish to add an additional *addToSet/3* predicate to the *Set* module which checks for a specific condition.

The operations of union, closed and open inclusion do not affect the semantics of Prolog, but only constrain the visibility of predicates during program development. For this reason, the gluing operations have a clear semantics, making their implementation straightforward. While other modulization systems for Prolog may be able to model these operations, Gödel's interactive error detection and correction makes it easy to use these operations. For example, if closed inclusion is used to import a procedure, *P*, from module M_2 into module M_1 and the user tries to extend *P* in module M_1, Gödel immediately generates a semantic error notifying the user of the illegal action.

Modulization and Non-logical Predicates

Non-logical predicates interact with a Prolog modulization system in a different manner than conventional predicates. For example, the predicates: *assert*, *retract*, *prove*, *bagOf*, *setOf* and *not* all require special consideration.

In procedural languages, module independence is ensured in two ways. Modules communicate using the protocol defined by their imports and exports so that internal objects of a module may not be accessed. In addition, the contents of each module (code and declarations) are static. That is, no code or declarations are added at run time. In Prolog a module is dynamic in that new clauses can be asserted and old ones can be retracted. This means that a further constraint must be added to a modulization mechanism to ensure module independence. That is, calls to the predicate *assert* must result in the new clause being added to the local clause-base of the module in which the call appears. In addition, calls to *retract* must be constrained to remove clauses only from the local clause-base.

Metapredicates take on a variety of forms. Some Prologs allow variables to appear as literals in the antecedent of a clause while others do not allow the promotion of logical variables to literals. In the latter case, the system uses a *call/1* or *prove/1* predicate. In the former case, the logical variable is interpreted as if it where the single parameter to the call or prove predicate. The semantics of these predicates are that the predicate's single parameter is interpreted as though it was inserted directly as a goal in the current refutation.

In a modulized Prolog system, the single parameter *prove* predicate is insufficient. Any call to a predicate must also specify which module the predicate should be proven in. In Gödel, the predicate *prove(ACall)* defaults to proving *ACall* in the module where the call predicate originated. However, Gödel also supports the predicate *prove(AModule, ACall)* which is true if *AModule* can prove *ACall*. If the predicate *prove/2* is used, the name bound to *AModule* must be visible within the calling module. In this way, the prove predicate is restricted since it cannot call predicates from arbitrary modules. If a call to a non-visible predicate is attempted, a run-time error occurs.

Second order predicates such as *bagOf* and *setOf* are used to find multiple solutions to a query. Their usual parameter sequence is a list of variables, a list of goals, and a list to collect each instance of the first parameter when the list of goals is satisfied. For example, *setOf([X, Y], pred(X, Y), AList)* is true when *AList* is a list containing each instance of *X* and *Y* such that *pred(X, Y)* is true. Conventional implementations of all solutions predicates use the prove predicate. In a modulized Prolog system which supports only closed inclusion (strict imports and exports), there are situations where *setOf* may fail. For example, suppose *setOf* is defined in module M_2, and it is imported into module M_1 using closed inclusion. Suppose that M_1 calls *setOf* with its second parameter bound to a predicate *P* in M_1. M_1 can see the definition of *setOf* and when *setOf* executes it tries to prove *P*. However, *P* is not visible in M_2. The union operation defined earlier remedies this problem by allowing the definition of *setOf* to be made available to M_1 and all definitions in M_1 to be made available to *setOf*.

Gödel uses the conventional definition for the *not* predicate. Since *not* uses the prove predicate, the same restrictions apply to it as to the all solutions predicates. To prove the goal *not(P)*, the predicate *P* must be visible from the module containing the *not* predicate. Therefore, any module wishing to use the *not* predicate must union or openly include the module containing *not*.

THE TYPER

Many non-procedural languages like Smalltalk, Lisp and Prolog do not support compile-time type checking. Parameters to methods (Smalltalk), arguments to functions (Lisp) or terms in predicates (Prolog) which are not of the correct form are only detected at run-time. In large software systems, such argument mismatches may cause unexpected errors to occur in sections of code only executed in unusual situations, long after product release.

For example, consider an operator *add*, which adds an object to a *Set*. Assume a logic error has resulted in the invocation of this operator in a situation where a constant is used as an argument instead of a *Set*. In Smalltalk, an argument mismatch error leads to an object receiving a message that is not in its class's method dictionary. For example, the Smalltalk expression *$a add: 1* results in a notifier window which states that an instance of class *Character ($a)* does not understand the message *add:*. In Lisp, if a parameter is mismatched, the error can be reported when it causes an error in a built-in function (whether it is reported depends on the dialect of Lisp). For example, *(add 'a 1)* could lead to an error when in the implementation of *add*, the interpreter checks if the atom *1* is in the atom *'a*, which should have been a list. In Prolog, the situation is worse. A run-time error will not occur since an argument mismatch will result in a unification failure and the interpreter will proceed. For example, *add(1, a, NewSet)* will simply fail when the terms *inList(1, a)* and *notInList(1, a)* in its definition fail to unify. The user is never warned of the type mismatch.

We are not claiming that Prolog is more susceptible to argument mismatch errors, only that it is more sensitive to them, since such errors are harder to detect. For this reason, we believe that it is essential to introduce some method of detecting argument mismatch errors if large software systems are to be constructed in Prolog.

Implicit or Explicit Typing

Several proposals exist for adding typing to logic programming (and Prolog). Types can be regarded as sets of terms generated by a regular tree grammar which describes the objects that a Prolog predicate manipulates[16]. In this way, no explicit type declarations are required.

Alternately, an explicit typing scheme can be used in which type definitions are predicates evaluated at compile time[17]. In such a scheme, predicates, functors and constants would be typed. Note that explicit variable type declarations are not necessary in a Prolog environment since variable types can be inferred when the predicates which use those variables are entered into the environment. For example, given the predicate declaration *isEmpty(List)*, the environment can infer that the type of the argument variable, *L*, in a term *isEmpty(L)*, is *List*. This means that a user need only declare types for functors, predicates, and constants.

We have used explicit typing in Gödel for three reasons:

1) Explicit type definitions can be used by programmers as an aid in understanding program structure and behavior.

2) Explicit type definitions can be used in compile time analysis to improve compiled code and execution speeds.

3) Explicit type definitions are useful in debugging so that failures due to the wrong type of argument, can be reported using more descriptive error messages.

An obvious disadvantage to the explicit typing approach is the need to specify types, especially during the prototyping stage of software development. To eliminate this problem, Gödel introduces the concept of pseudo-strong typing. Gödel supports two types of software; production software in which type declarations must be in place before clauses which use them and prototype software in which typing is optional.

For example, a prototype project may be developed without typing concerns. When the developer is convinced that the design is appropriate, typing constraints are introduced for predicates, constants, and functions. The developer is notified interactively about type inconsistencies within the code, allowing either reformulation of typing constraints, or reformulation of the otherwise ill-formed clauses. Although other Prolog systems have static type checkers, the ability to interactively make a transition from untyped to well-typed source code alleviates the burden of the constant mode switches required during the edit/compile loop of static type checkers.

Declarations

Constants, predicates and functions are typed by using declaration statements. In addition, a fourth kind of declaration is useful for declaring type symbols themselves. These type symbols can then be used in other declaration statements. By pre-defining type symbols, the problem of referencing an undeclared type during interactive editing is alleviated.

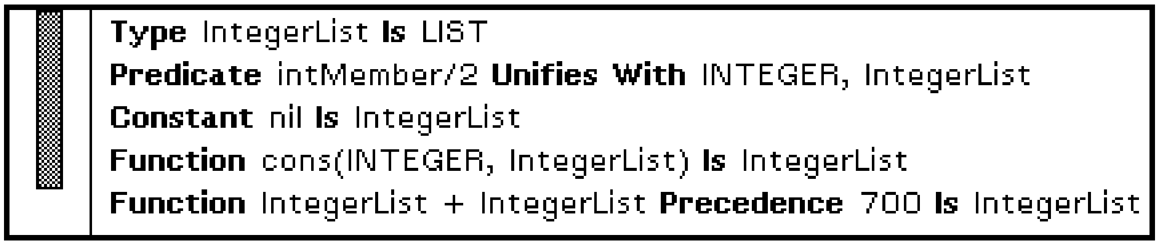
Each Gödel constant declaration consists of a constant name and a type. Each type declaration consists of a name and the name of another type which it refines. Gödel has several built-in types: *INTEGER*, *CARDINAL*, *CHAR*, *REAL*, *STRING*, *CARDINAL*, *LIST* and *OBJECT*. The *OBJECT* type is used as a default type and is compatible with all other types. The *LIST* type is used for lists and the [head|tail] notation is supported for objects of this type.

Each predicate declaration consists of a predicate name, and a list of type names. The type names constrain the types of the terms which can appear in the predicate. The arity is simply calculated as the length of the type list. For example, the declaration of a predicate which determines whether an integer is an element of an integer list could have the name, *intMember* and a list of the two type names: *INTEGER* and *integerList*, where *integerList* has been declared to be a refinement of type *LIST*.

Gödel also supports operators. Operators are only a notational convenience and are used to express terms where the functor is an infix, prefix, or postfix operator. To disambiguate the parsing of arguments between the three kinds of functors, precedence numbers and associativity are assigned to each operator. In Gödel, the declaration of such an operator is similar to the declaration of a normal predicate except that the predicate name is replaced by the operator symbol, a precedence number is included and a notation is used to specify the associativity of the operands.

Function declarations follow the same format as predicate declarations. Again there are two types: conventional functor declarations that define the functor's name and argument types plus the functor's return type, and function declarations that define operators where a precedence number and operand associativities are included.

The format of declarations and the way they are entered is discussed in the User Interface section of this paper, but figure 6 shows an example of some declaration statements. Note that these declarations are not entered as text so there is no need to enter the arity of predicates or formatting words like: "UNIFIES WITH" or "PREDICATE".



Type IntegerList **Is** LIST
Predicate intMember/2 **Unifies With** INTEGER, IntegerList
Constant nil **Is** IntegerList
Function cons(INTEGER, IntegerList) **Is** IntegerList
Function IntegerList + IntegerList **Precedence** 700 **Is** IntegerList

Figure 6: Some Type Declarations

Every declared object is an instance of the class *GuideDeclarationSymbol*. GUIDE[18] (General User Interactive Design Environment) is a programming environment whose basic structure was used in the implementation of Gödel. Subclassing is done to capture common behavior and state of all declarations. In addition, each symbol maintains a list of other symbols on which it depends. For example, a variable is a dependent of its type. These dependencies are used by the editor for static semantic checking.

Type Extensions

To provide more generic predicates, parameterized polymorphic typing is used (see reference 17). For example, the predicate *add/3* (where the 3 refers to the arity of the predicate) can be defined as having three parameters of types: *a*, *List of a*, and *List of a*, where *a* is a parameter which has a type as a value. This way, *add/3* can be used to insert a *Color* into a *List of Color*, and return the new *List of Color* or it can be used to insert an *Animal* into a *List of Animal*, and return the new *List of Animal*. The predicates which use type parameters are said to have polymorphic types.

The current version of Gödel does not support polymorphic types, but its addition is straightforward. The user would first be required to declare a parameterized type:

ListOfElement(Element), and then the function add/3 could be declared as function with the type list: *(Element, ListOfElement, ListOfElement)*.

Gödel supports generic predicates through the use of external declarations. For example, consider a sort procedure that uses a comparison predicate to determine the sorting order. Gödel's approach to incremental compilation and static semantic checking requires all predicates to be pre-declared. To make a generic procedure for sorting, we need to introduce an external declaration to declare a comparison predicate. Its purpose is merely a placeholder until a module gluing operation merges the formal definition to an actual definition. In a module using the sort predicate, the compare predicate could be defined in that implementing module.

Overloading is where a functor or constant symbol can have more than one type. For example the constant *nil* could be overloaded to be an *integerList* and a *characterList*. In Gödel, overloading of functor and constant symbols in the same module is not permitted. It is our belief that multiply defined symbols usually indicate that a program is not properly modulized. However, Gödel could be extended to allow multiply defined symbols in the same module. The type inference system would then check each occurrence of a symbol when determining correct typing.

Prolog's operational use of first order terms makes them behave as record structures or type constructors (see reference 12). A term, such as *person(X, Y, Z)*, can be interpreted as a three field record where the programmer assigns meanings to each field (for instance name, age and date of birth). The notion of a functor as a record construct can be used to add further declarations to Gödel.

For example, suppose we define an abstract data type for a *person*. Each person knows his name, age, and date of birth. The name is a string, the age an integer, and the date of birth an integer triple consisting of month, day and year. In Prolog, this record could be defined by a functor *person* of arity three. In Gödel, we introduce a record declaration:

```
FUNCTION person IS RECORD
    name          : STRING
    age           : INTEGER
    dateOfBirth   : Date
ENDRECORD OF Person;
```

This declaration does not change the semantics of Prolog. It only clarifies the programmers intended interpretation. The extra field declarations are not needed, but can be used not only add documentation to programs, but to produce more readable error messages. For example, if the functor *person* was used incorrectly in a clause, a message like "The name of a Person must be of type STRING" could be reported. Alternately, if a field was omitted the message could read "A Person consists of a name, age and dateOfBirth. Only two fields are present".

Another kind of declaration, the subrange, could be added to the declaration list and used to catch compile-time errors and improve program documentation. As with the record declaration, we do not wish a subrange to change the semantics of Prolog. An example of subrange use would be in the Date declaration referred to earlier:

```
TYPE Month IS SUBRANGE [1..12];
TYPE Day IS SUBRANGE [1..31];
FUNCTION date IS RECORD
    day     : Day
    month   : Month
    year    : INTEGER
END RECORD OF Date
```

Although we have chosen not to change the semantics of Prolog in Gödel, it would in fact be useful to make a change for sub-ranges. If the subrange type could be extended to the inference engine, run-time errors could be detected if a variable of a subrange type was instantiated to an object (number) outside the defined range. We have not implemented this, but future extensions to Gödel could include such operations.

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. Adding type inheritance 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 displayed considerable speed-up in execution time was a Prolog solution to Schubert's Steamroller problem[19]. A speed-up factor of ten 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.

THE USER INTERFACE

Graphical user interfaces have led to new ideas about the use of graphics for representing, constructing and editing programs in programming environments. The graphical representation of programs can influence language directed editing, approaches to incremental compilation, and debugging techniques. We have extended these ideas from the procedural programming domain and will show their usefulness in a logic programming environment. Many ideas developed here originate from environments like PECAN[20], Magpie[21], and GUIDE(see reference 18).

Gödel's interface was designed with three concepts in mind:

1) A consistent, single interface model should be available to the user. The ability to move between components without mental context switches or remembering component dependent commands is vital in designing an effective user interface.

2) Incomplete and incorrect programs must be representable. We take an approach in which the environment ensures static semantic correctness by using dialog windows that house information that is either waiting initial entry into the system or is semantically incorrect[22].

3) A graphical interface may affect the run-time environment. We wish to use graphical techniques to enhance the debugging of logic programs.

Basic Interface Objects

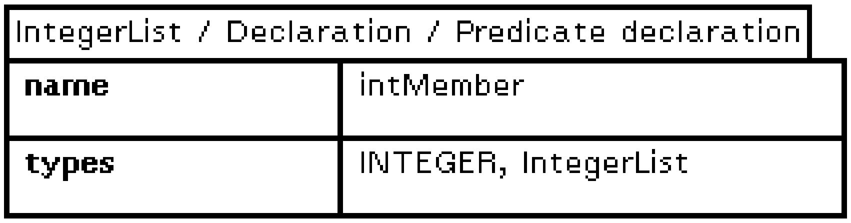
The developer interacts with Gödel using a three button mouse and a set of windows. Windows contain rectangular panes representing views onto aspects of the environment. These views include clause definitions, type definitions, module structures, and run-time traces. Windows can overlap each other, so navigation or context changes occur by simply moving the mouse from one window to another. The active window is a window with special status since no other window may overlap it. It is denoted by a highlighted title bar. A window is made active by using the mouse to point to it and pressing the mouse button. That same button selects character positions or structures within a pane of the window. Figures 7, 8 and 9 are examples of windows.

Commands are entered using pop-up menus. Each pane in a window has a menu associated with it. The menus that appear are context sensitive in that their contents are partially determined by the current state of the pane and the types of structures visible or selected.

A structure in a pane is selected by moving the cursor over the structure and pressing a mouse button. Selected structures are denoted by highlighting them with a contrasting background. Several structures may be selected by moving the cursor while a mouse button is pressed. Special actions may be performed on a structure by double clicking a mouse button while the structure is selected. The action taken depends on the current window and the selected structure, but usually an edit (dialog) box is opened on that structure.

Dialogs are a class of windows used to convey or retrieve specific information, and are classified into two categories: modal and modeless[23]. Modal dialogs (also called alerts) do not allow the user to switch contexts, but enforce explicit acknowledgement of the window's contents (message). In contrast, modeless windows allow the user to switch between contexts without being constrained to one "mode" of operation. Modal dialogs usually contain error information, or ask for important information that is required before execution can continue. On the other hand, modeless dialogs contain information that does not have to been immediately entered into the system (i.e. information that is not required by another object).

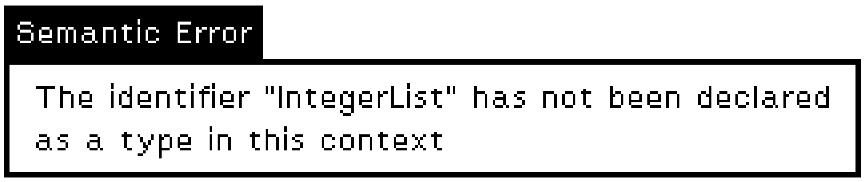
In Gödel, dialogs hold textual information that has yet to be accepted into the system or that was excised because of semantic inconsistencies. A dialog has a title tab that displays the module name and the type of object it represents as well as window panes containing text and associated labels. Figure 7 shows a modeless dialog box used to enter a declaration into the module called *IntegerList*. If the syntax and static semantics of the information is correct, then this information will be transferred to the clause base when the user selects the accept command from a pop-up menu. If it is not, then the information will remain in the dialog box until the user corrects the errors and re-accepts it. Semantic consistency of the clause base is maintained at all times by encapsulating semantically incorrect objects in dialogs.


IntegerList / Declaration / Predicate declaration

name	intMember
types	INTEGER, IntegerList

Figure 7: A dialog window for entry of a predicate declaration in the module IntegerList.

Gödel uses modal dialog boxes to convey error messages to the user. Figure 8 shows an alert that reports a static semantic error. This alert is modal and the user must acknowledge it by selecting close from a pop-up menu, before any other actions can be performed.


Semantic Error

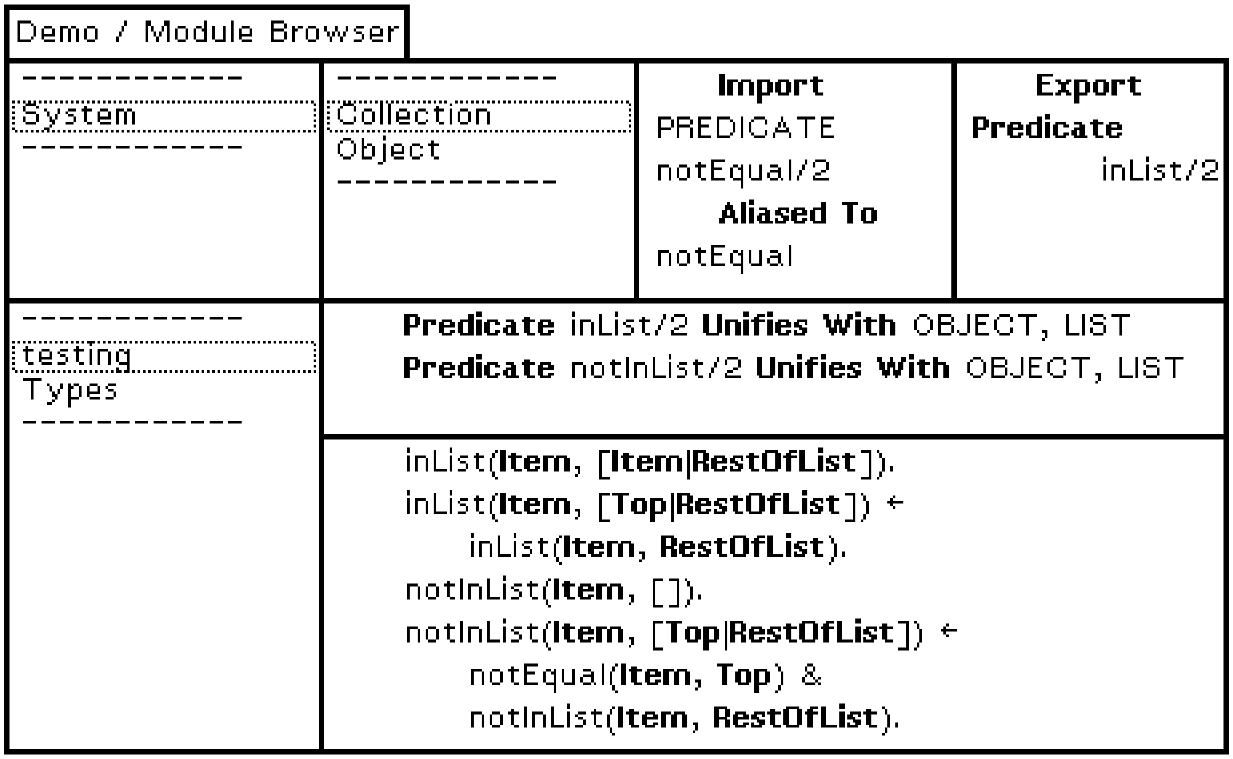
The identifier "IntegerList" has not been declared as a type in this context

Figure 8: An alert window for an undeclared type.

Browsers

Gödel provides browsers to navigate the module hierarchy, to navigate refutation trees and to access and modify program objects. The *module browser* contains panes to view and edit: modules, categories of modules, clauses and declarations within modules, and the gluing constructs between modules. The *debugger* browser displays the computation state of a query and is the interface to Gödel's debugger. The last browser is called a *workspace*. It displays the history of Prolog queries and their variable bindings. Each module owns its own instance of a workspace.

The module browser contains six panes as shown in figure 9. The top left pane contains a list of categories, where each category has an associated list of modules. Each module in the system is assigned a category so it can be located more efficiently. The module pane contains a list of modules owned by the currently selected module category. By selecting a module within the module pane, the remaining five panes update their views to display information particular to that selected module. Two panes display information associated with module gluing operations. The first pane contains a list of modules or objects from modules which are used by the selected module either by union or inclusion operations. The second pane displays a list of declarations visible to external modules.



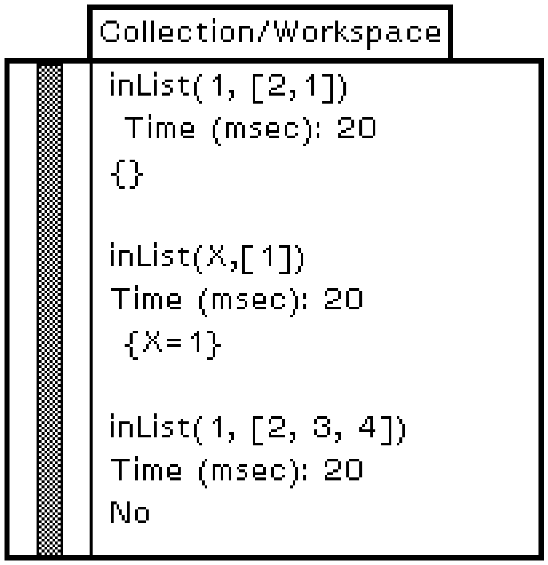
Predicate inList/2 **Unifies With** OBJECT, LIST
Predicate notInList/2 **Unifies With** OBJECT, LIST

```
inList(Item, [Item|RestOfList]).
inList(Item, [Top|RestOfList]) ←
    inList(Item, RestOfList).
notInList(Item, []).
notInList(Item, [Top|RestOfList]) ←
    notEqual(Item, Top) &
    notInList(Item, RestOfList).
```

Figure 9: A module browser

The other three panes are associated with program clauses for the selected module. They are the declaration protocol pane, the declaration pane, and the clause pane. The declaration protocol pane is similar to the module category pane. It is used to classify declarations. This allows efficient access to particular declarations. The declaration pane lists all declarations for the selected module while the clause pane contains all of the clause definitions for the module. The general design of the clause browser is to encapsulate in one window all information necessary to find and update program source efficiently.

Each module 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. A workspace for module *Collection* is shown in figure 10.



```
inList(1, [2,1])
 Time (msec): 20
{}

inList(X,[1])
Time (msec): 20
 {X=1}

inList(1, [2, 3, 4])
Time (msec): 20
No
```

Figure 10: A module workspace

Debugger browsers are described in the debugger section of this paper.

THE EDITOR

The design philosophy of Gödel's editor is to detect and correct errors as early in the development cycle as possible. The editor incrementally compiles all new input and performs both syntactic and static semantic analysis on this input. The editor provides user feedback as changes are made to clauses, without having to recompile the entire module (or file) where those clauses reside. It is our belief that this early error detection can increase developer productivity.

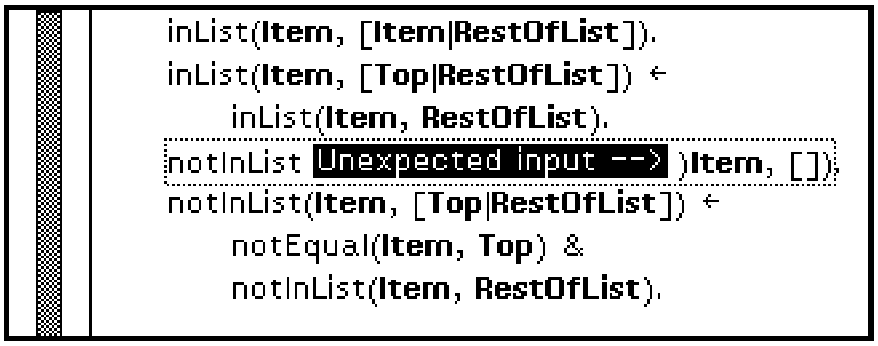
In-Line Dialogs

The goal of Gödel's language directed editor is to keep the clause database semantically consistent at all times. When syntactic and semantic errors occur during the incremental compilation of Prolog clauses, they cannot be acknowledged for later modification, but must be refused acceptance into the clause database. This means that the user must re-submit the incorrect information after taking corrective measures. Where is this information stored in the interim?

The solution we use is a variant of the one which has been implemented in GUIDE (see reference 22). The user is required to enter code and data through special dialog windows. As correct information is entered it is transferred from the dialog to another window which displays the program. In Gödel, the correct information is viewed in the clause pane of the module browser. This scheme is useful way of dealing with incorrect programs. However, in Gödel we have modified this approach to make entry more rapid.

Gödel supports a new kind of dialog called an *in-line dialog*. Instead of creating a separate text entry window, the in-line dialog is embedded within the clause pane. The in-line dialog consists of the clause being edited. The user initially selects an *insertion point*. in a clause. If the insertion point lies between existing clauses, then a new clause is created and contained in the in-line dialog. All text in the in-line dialog is enclosed in an outlining box so the extent of the in-line dialog is clear. If the insertion point is within an existing clause, then that clause is contained in the in-line dialog. When the user finishes typing and chooses accept from the pop-up menu, the clause is checked for syntactic and static semantic errors.

If a syntax error exists then the error message is inserted into the in-line dialog at the point of the error as shown in figure 11. If a static semantic error exists then an alert box is displayed similar to the one shown in figure 8, and the text in the input dialog causing this error is highlighted to give a visual indication of the error's position. If no errors exist then the clause is transferred to the clause data-base and the outline disappears to reflect its change in status.



```
inList(Item, [Item|RestOfList]).
inList(Item, [Top|RestOfList]) ←
    inList(Item, RestOfList).
notInList Unexpected input --> )Item, []).
notInList(Item, [Top|RestOfList]) ←
    notEqual(Item, Top) &
    notInList(Item, RestOfList).
```

Figure 11: A syntax error in an in-line dialog

Superficially, this process resembles the editing of methods in Smalltalk-80. However, there is a major difference. In Gödel, at any time, the user is editing only a single clause in the clause pane, not the entire contents. When the user chooses accept from the pop-up menu, the only code which is translated is the single clause in the in-line dialog. The other clauses in the pane are not affected. That is, this translation is incremental by individual clauses This would be analogous to incremental compilation of Smalltalk-80 code by individual statements!

The advantage of this approach is the ability to use the dialog entry paradigm while retaining the more familiar text entry format of conventional editors. Also, the context switch from editing in the clause pane to editing in a separate dialog window is alleviated.

It is straightforward to perform static semantic checking when declarations and clauses are added. However, static semantic errors can also be introduced by deleting structures. Consider the deletion of a type declaration, where the type is used in many procedure declarations. In Gödel, symbol dependencies (as mentioned in the typer section of this paper) are used to find all procedures dependent on the type to be deleted. Gödel then indicates to the user that the deletion of the type would cause a certain number of errors and asks for confirmation for the deletion, as shown in figure 12. If the user confirms the deletion, then all incorrect structures are excised from the code and stored in dialog boxes. If the user subsequently changes the program to make these structures correct, they can be re-accepted into the program without re-typing them.

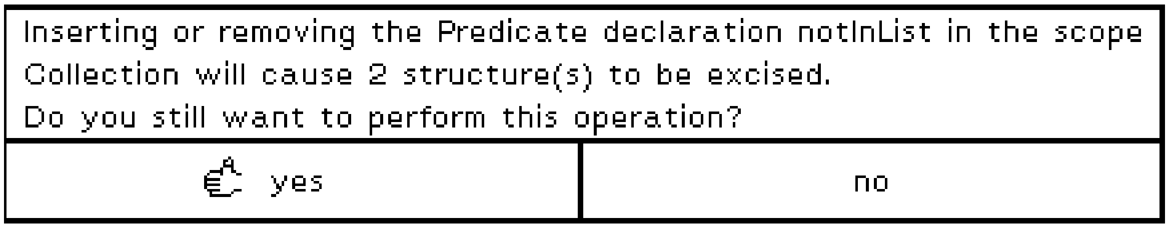

Inserting or removing the Predicate declaration notInList in the scope Collection will cause 2 structure(s) to be excised.
Do you still want to perform this operation?

Figure 12: A confirmation box for a deletion that causes semantic errors

Alternate Clause Representations

Language directed editing fits naturally into an interpreted language like Prolog because all program clauses must be compiled into an intermediate form for fast execution. By editing the compiled forms of clauses instead of their textual representation, incremental compilation of Prolog programs is straightforward. Not only does incremental compilation benefit from syntax directed editing, but so does the graphical representation of Prolog programs.

Since the editor knows about (and can manipulate) the syntax of clauses, it can automatically format source by making its own syntactic transformations. Note however that we are not claiming our scheme for clausal representation is more readable than another scheme, for this is a matter of taste. What we are trying to demonstrate is that the editor can be used to alleviate the burden of formatting source code, simultaneously allowing the addition of special syntactic constructs to the language.

A prime example of syntactic constructs is the if-then-else structure incorporated into most Prolog languages. If-then-else is used as a soft cut, eliminating the explicit use of the cut/0 predicate by the programmer. If-then-else is sometimes implemented directly into the inference engine, or can be defined using cut/0 and prove/1 as:

```
ifThenElse(X, Y, Z) ⇐              if X then Y else Z ⇐
    prove(X) ∧                     prove(X) ∧
    cut ∧                          cut ∧
    prove(Y).                      prove(Y)
ifThenElse(X, Y, Z) ⇐              if X then Y else Z ⇐
    prove(Z).                      prove(Z).
```

Historically, Prologs used a syntax like (<cond> -> <thenBody> ; <elseBody>) to represent the if-then-else. An example of where an if-then-else would be used is in a deterministic membership predicate which is true if its first argument is a member of the collection represented by its second argument. The implementor assumes the first argument is always ground and wants no backtrack points to remain once membership is satisfied (for efficiency sake). This behavior can be implemented in at least three ways:

```
/* conventional definition for member */
member(Item, [Item | RestOfList]) ⇐ cut.
member(Item, [FirstItem | RestOfList) ⇐
    notEqual(Item, FirstItem) ∧
    member(Item, RestOfList).

/* conventional (i.e. CProlog) syntax for member using if-then-else */
member(Item, [FirstItem | RestOfList]) ⇐
    ( Item = FirstItem ->
        true ;
        member(Item, RestOfList) ).

/* Gödel syntax */
member(Item, [FirstItem | RestOfList]) ⇐
    IF Item = FirstItem THEN
        true
    ELSE
        member(Item, RestOfList)
    ENDIF
```

We claim that the last version is more readable, and fits naturally into the language directed editing paradigm that Gödel supports. The editor stores the if-then-else in clausal form which can be proved via the defining if-then-else clauses:

```
member(Item, [FirstItem | RestOfList]) ⇐
    ifThenElse(=(Item, [FirstItem|RestOfList]),
            true,
            member(Item, RestOfList)).
```

However, the clause is displayed for editing in its expanded form. This idea can also be extended to constructs such a repeat/fail loops.

The debate on the use of if-then-else and other procedural notations has raised some concerns over their usefulness, or for that matter, their appropriateness. The use of if-then-else is geared towards an operational rather than declarative reading of clauses, results in writing complex clauses, and needs explicit equality calls to instantiate output variables[24]. For these reasons, our ideas of using syntax directed editing of these constructs should only be interpreted as follows: the use of procedural constructs are neither beneficial, hindering, or mandatory. Only that, if they are used, they should be treated procedurally.

THE DEBUGGER

There are two separate approaches to debugging Prolog programs: procedural and declarative (or algorithmic). The procedural (or trace) method closely follows the execution order of the inference engine. In contrast, the declarative (or diagnostic) method (see reference 24) searches through a program's computation tree and asks the user to verify predicates based on their currently instantiated variables[25]. From this information, erroneous program clauses can be uncovered.

In Gödel, we have chosen to design and implement a graphical procedural debugger rather than a declarative debugger for three reasons:

1) The procedural method is easy to incorporate into the interpreter. This allows quick prototyping to determine the feasibility of our ideas.

2) It is not clear if the declarative form of debugging is suitable for large programs since the debugging process is controlled by the system.

3) Because of the complexity of declarative debugging algorithms, it may prove advantageous to implement them as meta-interpreters in Prolog itself.

Future extensions to Gödel could experiment with incorporating declarative algorithms into the inference engine. This would provide a comparative basis for the two approaches.

The Model

There are two commonly used models for procedural debugging: the tree model and the box model. These models are designed so a programmer can easily trace and understand the complex execution pattern of his program. The tree model relies on the refutation tree formed by the depth first left to right computation of pure Prolog. The tree model makes it is easy to see how a goal is divided into subgoals.

The box model represents the flow of control through each subgoal. This model is simple, elegant, easy to implement, and is used by numerous Prolog implementations. It consists of four components, or corners of the box: **Call**, **Fail**, **Redo** and **Exit**. Most box model debuggers are designed for a line-oriented display device. However, Gödel uses a bit-mapped graphical display. We have transformed the box model to utilize the workstation's graphical capabilities.

Gödel's debugging model is a variant of the box model. The debugging window displays the four actions of *call*, *exit*, *fail*, and *redo* in a pop-up menu. The *step* command is used to proceed one unification at a time. It allows the user to enter into a subtree. The *call* command is used to execute an entire goal, skipping the subtree rooted at that goal. The *fail* command (not yet implemented) causes the current goal to fail. The *redo* command (not yet implemented) advances to the last backtrack point. An additional command, *proceed*, is used to continue until a breakpoint is reached or the refutation completes.

The Interface

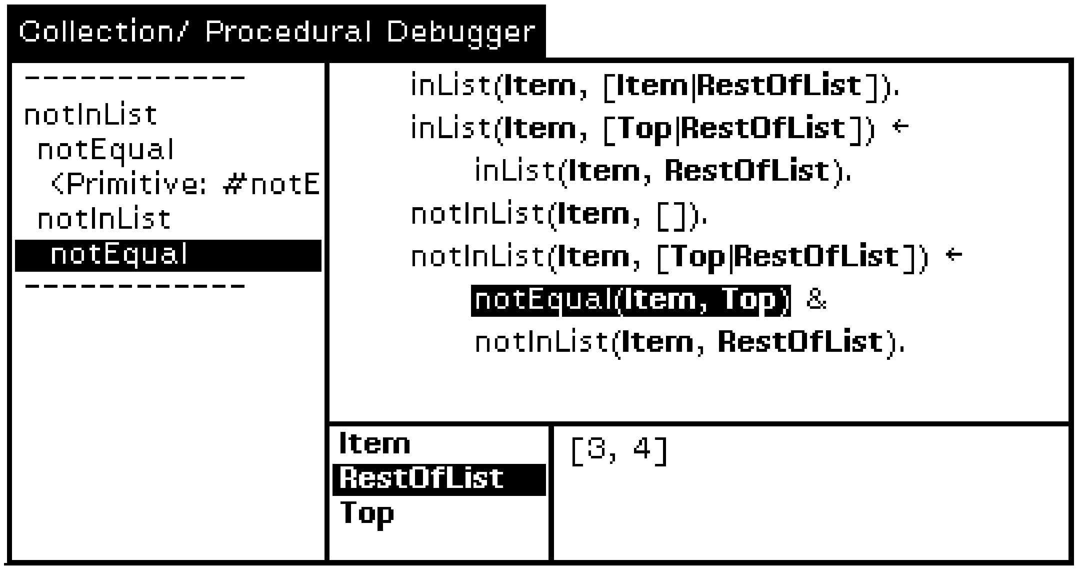
Gödel's procedural debugger consists of four components: a run-time stack pane, a code pane, a variable list pane, and a variable value pane. The run-time stack pane represents the goals and their associated subgoals produced during a refutation. This pane is the history of the current refutation. By selecting elements from this list, the user can peek into the refutation's history to view clauses and their variable bindings.

The code pane is the link between the interpreter's stack frames and the user's source code. It contains actual source code instead of some intermediate representation of the program. The debugger highlights the currently selected goal to make it easier for the user to make the connection between the source and a point in the refutation tree.

The variable list is the link between the source code's variable names and their bindings. This list consists of variables *as they appear* in the user's source. This eliminates the need for cryptic variable identifiers like _123 when browsing variable bindings.

The variable value pane is used for displaying the bindings of the variables. However, it can also be used to change these bindings interactively.

Figure 13 is an example debug window that provides a visual representation of the execution state. The query **notInList(5, [1, 2, 3, 4])** was issued. We have stepped to the point where a comparison (notEqual) is being performed.



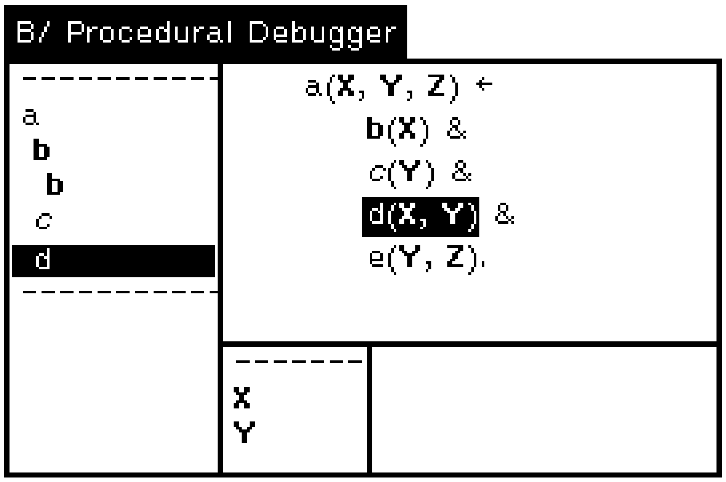
```
Collection/ Procedural Debugger

------------
notInList
 notEqual
  <Primitive: #notE
 notInList
  notEqual
------------

    inList(Item, [Item|RestOfList]).
    inList(Item, [Top|RestOfList]) ←
        inList(Item, RestOfList).
    notInList(Item, []).
    notInList(Item, [Top|RestOfList]) ←
        notEqual(Item, Top) &
        notInList(Item, RestOfList).

Item
RestOfList
Top

[3, 4]
```

Figure 13: The debug window

This graphical user interface makes it easier to visualize the execution of a Prolog program and can be further expanded if the interpreter is generalized. For example, although Gödel does not currently support delayed or parallel execution, its graphical interface could be used to display such results. Consider the case where the interpreter supports delayed execution of the goal:

a(X,Y,Z) ⇐ b(X) & c(Y) & d(X,Y) & e(Y,Z).

where b has succeeded, c is delayed until Y becomes bound, d is about to be executed and e is waiting for execution. Gödel could use multiple fonts to represent this situation as shown in figure 14, where completed goals are displayed in bold font, delayed goals are displayed in italics font and goals awaiting execution are displayed in normal font.



```
B/ Procedural Debugger

-----------
a
 b
  b
 c
 d
-----------

    a(X, Y, Z) ←
        b(X) &
        c(Y) &
        d(X, Y) &
        e(Y, Z).

-------
X
Y
```

Figure 14: Using fonts for program visualization

Breakpoints

In a trace and spy Prolog debugger, it is necessary to set breakpoints in a variety of locations. The user may wish to halt execution whenever a successful unification occurs with any clause head within a procedure. The user may also wish to set breakpoints between or directly on any atom within a particular clause. Inserting breakpoints between atoms is analogous to adding a primitive system predicate breakPoint/0 whose body notifies the interpreter that a breakpoint was reached. This method, though, requires the user to assume there is a procedural and predefined ordering of goals within the body of a clause, an unrealistic assumption in a true logic programming language.

We argue that breakpoints set on atoms must specify one of two conditions: break before attempting to solve the goal, or after the goal is solved (denoted *breakBefore* and *breakAfter*). Gödel will stop execution and display the current state of execution in the debug window when encountering a breakpoint. Breakpoints are set by selecting an atom within a clause and selecting the "add breakpoint" item from the clause menu. Clauses with breakpoints are displayed with a different font to give a visual clue of their presence. For example, in the clause grandfather(X, Y) $\Leftarrow$ father(X, Y) $\wedge$ father(Y, Z), if the user sets a *breakBefore* on father(X, Y), the bindings for X and Y can be examined but Y will still be unbound. If a breakAfter breakpoint is set on father(X, Y), Y will be bound.

THE INTERPRETER

Gödel's object-oriented inference engine (interpreter) is described in reference 13. The object-oriented interpreter allows for the distribution of code among the objects to be executed, in this case, clauses. The code to interpret clauses is associated with each clause object (or parse node). This is in contrast to interpreters using procedural designs where there exists a central procedure that, given a clause base and a query, defines the actions to perform the refutation.

In Gödel, there is no central interpreting procedure. Each object in the system knows what to do when receiving a message to execute itself in a given context. The defining context for Prolog interpretation is represented by the class *GödelInterpreter*. A GödelInterpreter is an abstract data type containing methods that set and return the values of its instance variables. It also contains methods to answer queries from the debugging window about the interpreter's current state. Overall, a *GödelInterpreter* is the encapsulation of state variables. By having several instances of a *GödelInterpreter*, several queries can be at various stages of execution.

Gödel also takes an object-oriented approach to unification. There are three advantages to this approach:

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 utilized. 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.

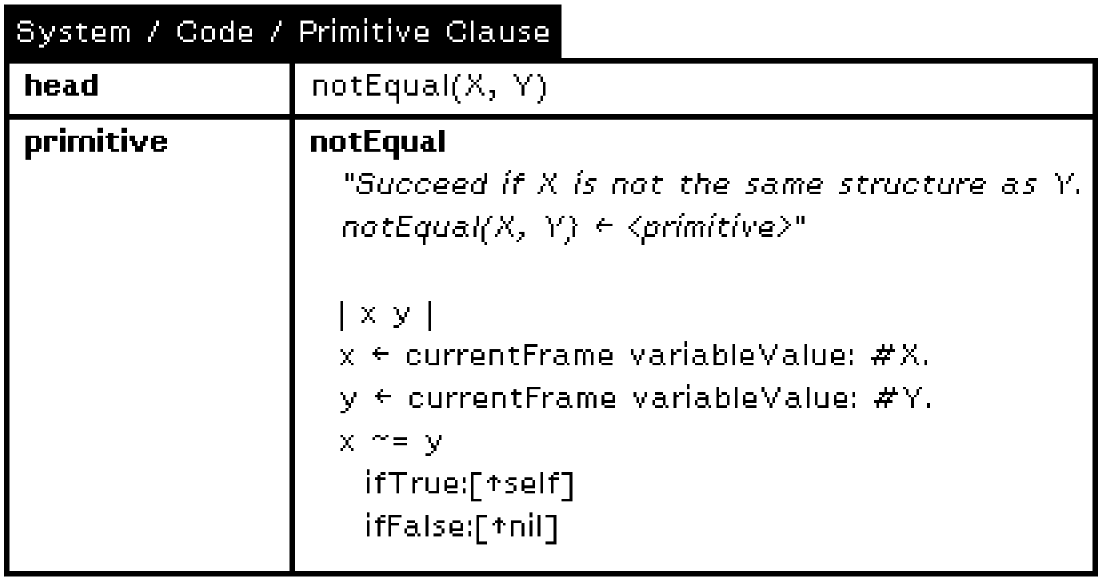
For example, Smalltalk's *Collection* classes encapsulate sequences of objects. It is possible to write a predicate *addToList(AnObject, AList)* where *AList* is an instance of a *Collection* instead of a Prolog list structure. Similarly, the class *View* can be used to create windows that Prolog code 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.

Each clause class can respond to the message *#stepIn: anInterpreter*. The single argument, *anInterpreter*, is an instance of a *GödelInterpreter*. When a 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 grandfather and father predicates as defined previously, along with the query *grandfather(X, Y)*. The interpreter's *currentCall* is the *GödelAtomNode*, *grandfather(X, Y)*. The message *#stepIn* is sent to the node *grandfather(X, Y)*. A *GödelAtomNode's* behavior upon receiving this message is to search 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 body 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 15 shows an example of a primitive clause in Gödel.



System / Code / Primitive Clause

head	notEqual(X, Y)
primitive	see below

```
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 15: 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 which is the first argument to the value which 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.

Although adding primitive clauses to Prolog applications detracts from the elegance of having one implementation language, it can be offset by the gains in efficiency.

CONCLUSION

Gödel is an interactive incremental programming environment for Prolog. It has a uniform and powerful graphical user interface that provides immediate error feedback during development. It superimposes modulization and typing schemes onto Prolog to make it more attractive for developing larger production applications. It provides a seamless interface to the object-oriented programming paradigm through primitive clauses that have the power of Smalltalk-80.

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. D.H.D. Warren, "An Abstract Prolog Instruction Set.", *SRI Technical Note 309*, Menlo Park, California, 1983.
2. "Outline of Research and Development Plans for FGCS", ICOT Research Center, Minato-ku, Tokyo, May 1982.
3. J.W. Mills "A high performance LOW RISC machine for logic programming", *Proceedings of the IEEE 1986 3rd International Symposium on Logic Programming*, 1986.
4. E. Tick and D.H.D. Warren, "Towards a Pipelined Prolog Processor" *Proceedings of the IEEE 1984 International Symposium on Logic Programming*, 1984.
5. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.
6. R.E. Sweet, "The Mesa Programming Environment", *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, 1985.
7. W. Teitelman and L. Masinter "The Interlisp Programming Environment", in *Interactive Programming Environments*, Barstow D.R., Shrobe H.E. and E. Sandwell (eds.), McGraw-Hill, 1984, pp.83-96.

8. D. Lanovaz "Godel: A Prototype Prolog Programming Environment", Master of Science Thesis, Univerity of Alberta, Edmonton, (1988).

9. P.S. Van der Meulen "INSIST: Interactive Simulation in Smalltalk", *OOPSLA Conference Proceedings*, 1987.

10. J. Laursen, R. Atkinson "Apus: A Smalltalk Production System", *OOPSLA Conference Proceedings*, 1987.

11. N. Grossman and R.K. Ege "Logical Composition of Object-Oriented Interfaces", *OOPSLA Conference Proceedings*, 1987.

12. H. Ait-Kaci and R. Nasr "LOGIN: A Logic Programming Language with Built-in Inheritance", *Journal of Logic Programming* **3**, 1986.

13. D. Lanovaz and D. Szafron, "An Object-Oriented Inference Engine for Prolog", *Technical Report 90-18*, Department of Computing Science, Univerity of Alberta, Edmonton, May 1990.

14. D.A. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules, *Comm. ACM*, 15(12), December 1972.

15. R.A. O'Keefe, "Towards an Algebra for Constructing Logic Programs", *Proceedings of the IEEE Symposium on Logic Programming*, 1985.

16. P. Mishra, "Towards a theory of types in Prolog", *Proc. IEEE International Symp. on Logic Programming*, Atlantic City, 1984.

17. A. Mycroft and R. O'Keefe, "A Polymorphic Type System For Prolog", *Artificial Intelligence*, 23, 1984.

18. D. Szafron and B. Wilkerson, "GUIDE: Preliminary User's Manual", *Technical Report TR86-05*, Department of Computing Science, Univerity of Alberta, Edmonton, March 1986.

19. D. Szafron and F.J. Pelletier "Some Notes on Prolog Technology Theorem Proving", *Technical Report TR89-10*, Department of Computing Science, Univerity of Alberta, Edmonton, March (1989).

20. S.P. Reiss, "Graphical Program Development with PECAN Program Development Systems", *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.

21. N.M. Delisle, D.E. Menicosy and M.D. Schwartz, "Viewing a Programming Environment as a Single Tool", *ACM*, 1984.

22. D. Szafron and B. Wilkerson "Some Effects of Graphical User Interfaces on Programming Environments", *CIPS Congress 86*, Vancouver, 1986.

23. *Inside Macintosh*, Vol. I, II, III [1985], Adison-Wesley, Inc., Reading, second printing, 1985.

24. E. Shaprio, *Algorithmic Program Debugging*, The MIT Press, 1982.

25. L.M. Pereira "Rational Debugging in Logic Programming", *Lecutre Notes in Computer Science*, No. 225, Goos G. and J. Hartmanis (eds.), 1986.