

# Bristol Gödel Users' Manual

(Release 1.4, May 1994)

Antony F. Bowers and Jiwei Wang

Department of Computer Science  
University of Bristol  
University Walk  
Bristol BS8 1TR  
UK

email: {bowers, jiwei}@compsci.bristol.ac.uk

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Using the Gödel system</b>	<b>4</b>
2.1	Basic system commands . . . . .	4
2.2	Gödel utilities . . . . .	5
2.3	Debugging facilities . . . . .	6
2.4	Interface to shell . . . . .	7
2.5	Executing queries . . . . .	7
<b>3</b>	<b>Use of the Gödel tracer</b>	<b>9</b>
<b>4</b>	<b>Computational model</b>	<b>11</b>
4.1	Lloyd-Topor transformation . . . . .	11
4.2	Safe negation . . . . .	12
4.3	Computation rule . . . . .	12
4.4	Constraint solving . . . . .	13
4.5	Set unification . . . . .	14
<b>5</b>	<b>Interface to Prolog and C</b>	<b>15</b>
<b>6</b>	<b>Some hints for good programming in Gödel</b>	<b>17</b>
<b>7</b>	<b>Facilities absent from this release</b>	<b>20</b>
<b>8</b>	<b>Known bugs and limitations</b>	<b>21</b>
<b>9</b>	<b>Acknowledgements</b>	<b>23</b>
<b>10</b>	<b>References</b>	<b>24</b>

# 1 Introduction

*Bristol Gödel* attempts to be a reasonably faithful implementation of the Gödel language as described in *The Gödel Programming Language* by P.M. Hill and J.W. Lloyd, MIT Press 1994. However, due to the limitations of the underlying execution model, not all the features of the Gödel language are supported. This document explains the implementation of the Gödel language. It also serves as a supplement to *The Gödel Programming Language* concentrating on the practical use of the Gödel language.

The *Bristol Gödel* system is implemented in SICStus Prolog, and SICStus Prolog is also the target language of the Gödel compiler. There were two reasons for this decision. Firstly, Prolog is widely considered suitable for fast prototyping, and our own experience confirms this. It took about 1 man year to ship the first version of Bristol Gödel. Given that the Gödel language was constantly changing and evolving, this development time is rather short. Owing to recent improvements in Prolog implementation, the prototype system runs at a reasonable speed. With some optional checking disabled, the Gödel parser compiles at about 120 lines per sec. on a SPARC 2. Secondly, the semantic gap between Gödel and Prolog is much smaller than that between Gödel and C or WAM code. By compiling Gödel programs to Prolog, the compilation process is greatly simplified. However, it should be pointed out that Gödel's *commit* pruning operator could not be implemented efficiently in its full generality without modifying the underlying Prolog engine, so the implementation has some gaps, but these won't affect most users of the language.

Without any delay declarations, negations or IF-THEN-ELSE constructs, Gödel programs run at the same speed as equivalent SICStus programs (about 600K LIPS on a SPARC 2). A carefully written Gödel program (with all features) should run at about half of the speed of SICStus.

Please send bug reports to [goedel@compsci.bristol.ac.uk](mailto:goedel@compsci.bristol.ac.uk).

All suggestions for improvement are welcome.

## 2 Using the Gödel system

To use the Gödel system, you need to run the executable file called ‘goedel’. (Instructions for installing the Gödel system can be found in the README file in the top level directory of this release.) If you do not have SICStus Prolog, your Gödel system will be based on the SICStus runtime system. There are a few differences between this *Runtime Gödel* and Gödel based on full SICStus Prolog, and these will be indicated where appropriate. Having entered the Gödel system, you will see a prompt ‘[] <-’ at which you may type commands or Gödel queries. Commands always begin with a semi-colon (;) character to distinguish them from queries. All commands and Gödel queries end with ‘.’ followed by a carriage return (CR).

The commands available in the Gödel system are presented below. Some commands take one or two parameters. The parameters are either a `BigName` (a sequence of alphanumeric characters starting with a capital letter, e.g. `Module`) or a `String` (any sequence of characters enclosed in double quotes, e.g. `"File"`). The syntax of each command is given in brackets after its name. The commands are divided into four groups: basic commands and commands for compiling and loading Gödel programs; utilities for performing other operations on Gödel files; debugging utilities, and an interface to Unix.

### 2.1 Basic system commands

**Help** (;h.) Gives rudimentary help information and lists the abbreviations that may be used.

**Quit** (;quit.) This command returns you to the Unix shell.

**Compile** (;c `Module`.) This compiles the given module. In compiling a module, the Gödel system looks for files whose name consists of the module name and the extension ‘.loc’ or ‘.exp’. Error and warning messages are printed out if there are any problems with syntax. Programs that generate only warnings may still be correct, but we strongly advise you never to ignore warning messages.

There may be several error messages for a single error, due to Gödel’s overloading mechanism. Not all the error messages are relevant, but it’s advisable to read through them all.

If there is no error in the program, three files will be generated. The names of the files consist of the module name and extensions ‘.lng’, ‘.pl’ and ‘.ql’. (If you are using *Runtime Gödel*, only the ‘.lng’ and ‘.pl’ files are generated. In this case, the ‘.pl’ file is not compatible with that generated by a normal Gödel system.) The ‘.lng’ file contains the language of the module. The ‘.pl’ file is the compiled Gödel code in Prolog; and the ‘.ql’ file is the ‘.pl’ file compiled by SICStus.

If the given module imports some other modules, and those modules have already been compiled, the ‘;c’ command will load their ‘.lng’ files instead of parsing those modules.

**Make** (`;m Module.`) This compiles the given module, and also compiles all the user modules it depends upon, whether or not these have already been compiled.

**Load** (`;l Module.`) This loads the compiled module and the user modules it depends upon from their `.lng` and `.ql` files (`.pl` files in the case of Runtime Gödel). Absence of any of these files causes the load command to abort. After loading a Gödel program, the system prompt changes to `[Module] <-`, where `Module` is the name of the main module in the program.

**Make and Load** (`;ml Module.`) This makes and loads the module into the system. Any error causes the `;ml` command to abort.

**Check Syntax** (`;cs Module.`) This command checks the syntax of the given module but does not create any files. It is useful to determine the syntactical correctness of an incomplete program.

**Save States** (`;save File Goal.`) This instructs Gödel to save the current state (the loaded Gödel program) into the executable file `File`. When `File` is run, Gödel will start the execution of `Goal`. This command is not available in the Runtime Gödel.

**Switch On Checking** (`;checking.`) This switches on various checks. At the moment, three kinds of check are supported, and they are all enabled by default when the Gödelsystem starts up. The first is the singleton variable check which finds any singleton variables in a clause. Note that the effects of disjunctions and IF-THEN-ELSE constructs are not considered in the check. The second is the quantified variable check. This identifies quantified variables which have not been used in quantified subformula. The third is the floundering check. It is a primitive (but effective) way to single out variables which will definitely cause floundering under the current computation model.

**Switch Off Checking** (`;nochecking.`) This switches off the checks turned on by the `;checking` command. Using this command improves compilation speed, but it is not recommended until the program is known to have passed all the checks in a previous compilation.

## 2.2 Gödel utilities

**Program Compile** (`;pc Module.`) This creates the ground representation of the specified program whose main module is `Module` in a file whose name consists of the main module name and `.prm` extension. If there is any syntax error in the specified module and its dependent user modules, the ground representation of the module is not created.

**Program Decompile** (`;pd "File".`) This takes the ground representation of a program from the file whose name consists of the module name and `.prm` extension, and generates source files for all the component user modules of this program. Where

‘.exp’ and ‘.loc’ files already exist for one of these modules, they will be renamed with ‘.exp.old’ and ‘.loc.old’ extensions respectively, so that the existing source is not immediately destroyed.

**Script Compile** (;sc Script.) This compiles a generated script in the file whose name consists of the script name and ‘.scr’ extension and produces ‘.lng’, ‘.pl’ and ‘.ql’ files so as to be loaded for execution. If you are using Runtime Gödel, the ‘.ql’ file is not created.

**Flock Compile** (;fc "File" "Flock".) This takes a flock and returns a Gödel internal representation of the flock in a file whose name consists of the flock name and ‘.flk’ extension. If there is an existing flock of the same name, the existing flock will be overwritten. If there is any syntax error in the file of units, the ‘;fc’ commands aborts.

**Flock Decompile** (;fd "Flock" "File".) This takes a Gödel flock and returns a file of units. The flock should be in a file with the name consisting of the flock name and ‘.flk’ extension. If there is an existing file of the same name, the existing file will be overwritten.

**Canonicalise Prolog** (;cp "File1" "File2".) This utility converts Prolog clauses or terms in File1 into their canonical form in File2. Variable names are not preserved.

**Decanonicalise Prolog** (;dp "File1" "File2".) This converts Prolog clauses or terms in canonical form in File1 into standard form in File2.

## 2.3 Debugging facilities

**Debug Compile** (;dc Module.) This has the same effect as the ‘;compile’ command except that it also incorporates debugging information into the compiled code for the given module. Debug compiled modules run much more slowly.

**Trace** (;trace.) Switches on the tracer.

**Notrace** (;notrace.) Switches off the tracer.

**Spy** (;spy Predicate.) Sets spy points at all predicates or propositions with declared name Predicate, regardless of their arity and the module in which they are declared.

**Nospy** (;nospy Predicate.) Removes spy points at all predicates or propositions with declared name Predicate, regardless of their arity and the module in which they are declared.

**Nospyall** (;nospyall.) Removes all spy points.

**Type** (;t Symbol.) Displays the declarations of all the constants, functions, propositions and predicates in the goal language of the loaded program that have the declared name Symbol.

## 2.4 Interface to shell

**cd** (;cd "Directory".) Changes to the specified directory. The ‘Directory’ argument must be a string. If it is omitted, ‘;cd’ changes to the user’s home directory.

**ls** (;ls "Directory".) Lists the content of a directory. The ‘Directory’ argument must be a string. If it is omitted, ‘;ls’ lists the current directory.

**more** (;more "File".) Invokes the Unix ‘more’ command for the specified file.

**Shell** (;shell "Command".) Executes a shell command of the underlying operating system. If ‘Command’ is omitted, this enters a shell.

**pwd** (;pwd.) Displays the current directory.

## 2.5 Executing queries

A Gödel query should be a goal in the goal language of the program whose main module is the current module. A full stop ‘.’ should follow every query. The Gödel system attempts to solve the query and return answer bindings for the free variables appearing in it. If the query finitely fails, ‘No’ is printed. If it can be solved with the empty answer substitution, ‘Yes’ is printed. Otherwise, the answer substitution appears in the form:

x = Term,

for each free variable in the query. The system then prompts with a ‘?’. Typing in ‘;’ after the question mark causes the system to search for another solution. A carriage return (CR) returns you to the main prompt.

An example session of using the Gödel system follows.

```
% goedel
Goedel 1.3
Type ;h. for help.
[] <- ;l Lists.
[Lists] <- Append(x, y, [1, 3]).

x = [],
y = [1,3] ? ;

x = [1],
y = [3] ? ;

x = [1,3],
```

```

y = [] ? ;
No
[Lists] <- ;l Integers.
[Integers] <- x^2 + y^2 = z^2 & 0<x<50 & 0<y<50 & 0<z.

x = 3,
y = 4,
z = 5 ? ;

x = 4,
y = 3,
z = 5 ? ;

x = 5,
y = 12,
z = 13 ?
Yes
[Integers] <- ;ml EightQueens. % the EightQueens program
                                % can be found in the Goedel
                                % release package under
                                % directory /programs/demo/

Reading file "EightQueens.loc" ...
Parsing module "EightQueens" ...
Compiling module "EightQueens" ...
Module "EightQueens" compiled.
Loading module "EightQueens" ...
[EightQueens] <- Queen(x).

x = [1,5,8,6,3,7,2,4] ? ;

x = [1,6,8,3,7,4,2,5] ? ;

x = [1,7,4,6,8,2,5,3] ?
Yes
[EightQueens] <- ;q.
%
```



### 3 Use of the Gödel tracer

There are two modes of tracing in the Gödel system, namely, *trace* and *spy*. Trace starts tracing from the query, while spy only triggers the tracing at specified predicates. Trace and spy work only with debug-compiled modules (see **Debug Compile** in the previous section). It is recommended that tracing is done on a few particular modules which possibly contain bugs. Only these modules are debug compiled to avoid irrelevant tracing information.

When the tracer is invoked, execution of the `call`, `exit`, `fail` and `redo` ports of each debug-compiled procedure will be displayed. Each `call` has an unique index and each `exit`, `fail` and `redo` has an index corresponding to their original `call`. At each entry, you are prompted with ? at which the following commands can be used.

```
a - abort:      abort current query.
c - continue:   continue execution without trace info.
f - free:       let the tracer run free and print out all the trace info.
h - help:       print this message.
l - leap:       leap to the next spy point.
n - next:       next goal, the same as a carriage return.
r - redo:       reenter the failed goal, valid only in "fail" entries.
s - skip:       skip the trace info for executing the current goal,
                valid only in "call" entries.
1-9:           set term display depth in the tracer.
```

Suspended and awakened goals are also printed out under entries `suspend` and `awaken`, respectively. In a similar way to `call`, each `suspend` also has an unique index. Each `awaken` has an index corresponding to its `suspend`.

In this release, the tracer does not mask off terms defined in system modules as abstract data types. In a future release, a more sophisticated debugging tool will provide facilities for properly displaying terms in an abstract data type.

The Gödel tracer displays variables in the format `_N`. Variables in a suspended goal are renamed when the goal is awoken. Module prefixes are attached to the predicate with a `:`, e.g. `Lists:Append([1,2], [3,4], _12345)`. Syntactic sugars are sometimes displayed in the sugared form, e.g. `lists` and `sets`, and sometimes in the original form, e.g. the `Interval/3` function in the `Integers` module. Constraints are replaced by a variable and cannot be seen in the trace. This can be seen in the following example.

```
[Integers] <- ;trace.
Tracing on
[Integers] <- x^2 + y^2 = z^2 & 1 < x < 9 & 1 < y < 9.
  0 call: _34205=_33909 ?
  0 exit: _34710=_34710 ?
```

```
1 call: Integers:Interval(2,_34104,8) ?
1 exit: Integers:Interval(2,2,8) ?
2 call: Integers:Interval(2,_34008,8) ?
2 fail: Integers:Interval(2,_34008,8) ?
1 redo: Integers:Interval(2,2,8) ?
1 exit: Integers:Interval(2,3,8) ?
3 call: Integers:Interval(2,_34008,8) ?
3 exit: Integers:Interval(2,4,8) ?
```

```
x = 3,
y = 4,
z = 5 ?
```

Users of the Gödel tracer need to be aware that Gödel programs are converted into normal form using the Lloyd-Topor transformations (see Section 4.1 and [Lloyd 87] p. 113) and it is the normal forms which are traced.

Commits are invisible in the Gödel tracer. Furthermore, commits are ‘switched off’ when solving a negative goal (to be elaborated in the next section). The tracer does not give any indication about commits being switched off.

The **redo** command in the Gödel tracer can only be used on failed goals, due to implementation difficulties. Most predicates are intended to succeed, so the unexpected failure of a goal is a common indication of a programming error. When this happens, the **redo** command can be used to investigate the error.

The Gödel tracer differs from the Prolog tracer in that it does not have ‘break’. It is probably not necessary to have ‘break’ in the Gödel system, because one cannot modify the running program from the break level.

## 4 Computational model

The accompanying implementation of the Gödel language maps Gödel programs into Prolog. This has allowed us to build prototypes of the language quickly by using existing logic programming techniques. However, the computational model of the Gödel system is limited by the existing implementation of Prolog.

This implementation of the Gödel system uses a variant of SLDNF-resolution as its procedural semantics. In this section, we shall try to explain this computational model.

### 4.1 Lloyd-Topor transformation

Let a program statement be  $\text{Head} \leftarrow \text{Body}$ , and a goal be  $\leftarrow \text{Goal}$ . Gödel allows arbitrary first-order formulas in  $\text{Body}$  and  $\text{Goal}$ . In order to execute Gödel programs on an SLDNF-resolution based system, Gödel programs have to be transformed using the Lloyd-Topor transformation [Lloyd 87, p113]. The standard Lloyd-Topor transformation can introduce duplicated computation by creating new clauses. For instance, when

$$\begin{array}{l} \text{is replaced by} \quad A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (V \vee W) \wedge W_{i+1} \wedge \dots \wedge W_m \\ \text{and} \quad \quad \quad A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \dots \wedge W_m \\ \quad \quad \quad \quad A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \dots \wedge W_m \end{array}$$

the formula  $W_1 \wedge \dots \wedge W_{i-1}$  might have to be computed twice in the transformed program. Because  $W_1 \setminus W_2$  and  $\sim \text{SOME} [x_1, \dots, x_n] W$  are handled by the underlying system, we can skip steps in the Lloyd-Topor transformation which create new clauses. The result is a transformation syntactically different from that in [Lloyd 87], but semantically the same. The important difference is that no new clauses are generated in the transformation.

1. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (V \vee \neg W) \wedge W_{i+1} \wedge \dots \wedge W_m$
2. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (V \leftrightarrow W) \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (V \rightarrow W) \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \dots \wedge W_m$
3. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \exists x_1 \dots \exists x_n W \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \dots \wedge W_m$
4. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \forall x_1 \dots \forall x_n W \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg \exists x_1 \dots \exists x_n \neg W \wedge W_{i+1} \wedge \dots \wedge W_m$
5. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg(V \wedge W) \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge (\neg V \vee \neg W) \wedge W_{i+1} \wedge \dots \wedge W_m$

6. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg(V \vee W) \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg V \wedge \neg W \wedge W_{i+1} \wedge \dots \wedge W_m$
7. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg(V \leftarrow W) \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg V \wedge W \wedge W_{i+1} \wedge \dots \wedge W_m$
8. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg\neg W \wedge W_{i+1} \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \dots \wedge W_m$
9. Replace  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \neg\forall x_1 \dots \forall x_n W \wedge \dots \wedge W_m$   
by  $A \leftarrow W_1 \wedge \dots \wedge W_{i-1} \wedge \exists x_1 \dots \exists x_n \neg W \wedge \dots \wedge W_m$

## 4.2 Safe negation

The safeness condition is imposed in the Gödel system. In other words, negated subformulas of a goal are delayed until they contain no free variables. Furthermore, to ensure the soundness of the negation as failure rule, commits are disabled inside the execution of a negated goal.

These two restrictions are also applied to the execution of the condition in IF-THEN-ELSE constructs. This is because IF Cond THEN ThenPart ELSE ElsePart is semantically equivalent to Cond & ThenPart  $\setminus$  /  $\sim$ Cond & ElsePart.

## 4.3 Computation rule

The computation rule in the Gödel system, called the *faithful computation rule*, generally selects the leftmost unsuspended goal, subject to the following caveats:

- constraints may be solved in any order that the system finds convenient;
- in some circumstances the order in which suspended goals are woken is undefined (due to a limitation of SICStus Prolog).

If all the goals are suspended, the computation flounders.

Once the execution enters the scope of a commit, it must solve the scope entirely before pruning can occur. If no literal in the scope can be selected, so that execution must leave the scope of the commit before solving it completely, then that commit becomes permanently disabled, that is it will never prune even if its scope is eventually solved.

The order in which clauses are selected in the execution of a Gödel program is not defined.

## 4.4 Constraint solving

Implementation of constraint solving in the modules `Integers`, `Rationals`, `Strings` and `Sets` is primitive. The basic idea in the implementation is to use the coroutining mechanism to suspend constraints until they become solvable. A constraint is solvable when there are sufficient bindings to produce a finite number of solutions. Sophisticated constraint solving methods are not supported yet. The following two goals demonstrate solvable constraints.

```
<- x Mod 3 = 1.      % suspends and eventually flounders
<- x^2 = 4.         % gives two answers.
```

It's hard to define what is currently available. Generally, this release supports:

1. Evaluating expressions with data structures defined in system modules `Integers`, `Rationals`, `Strings` and `Sets`. For example, the following procedures can be executed.

```
PREDICATE P : Integer.

P((2*3+4) Div 6).

PREDICATE Sum : Set(Integer) * Integer.

Sum({}, 0).
Sum(set, sub_total + x) <-
  x In set &
  Sum(set\{x}, sub_total).

PREDICATE AddPostfix : String * String.

AddPostfix(file_name, file_name ++ ".prm").
```

2. Generating integers in an interval. E.g.  $1 < x \leq 5$ .
3. Solving linear equations which are in triangular form. For example,

```
<- 2*x+1 = y+2 & 3*y-2 = 1.

<- x/2 + y/3 = 5/6 & y/2 + 1/3 = 5/6.
```

4. Exhaustive search with coroutining.

```
<- x^2 + y^2 = z^2 & 1<x<50 & 1<y<50 & 0<z.
```

## 4.5 Set unification

Full set unification is not implemented, because it would be too inefficient to support it at the Prolog level. Unification can be performed for ground sets, and a variable and a set; attempts to do more complex set unifications will flounder. For example,

```
[Sets] <- {1,2,3} = {3,2,1,2}.
```

```
Yes
```

```
[Sets] <- {1,2,3} = {3,2,1,4}.
```

```
No
```

```
[Sets] <- x = {3,2,1,2,3}.
```

```
x = {1,2,3} ?
```

```
Yes
```

Set operations which contain non-ground sets will be delayed until the non-ground sets become ground. For example,

```
[Sets] <- x In ({y} + {z}) & y=1 & z=2.
```

```
x = 1,
```

```
y = 1,
```

```
z = 2 ? ;
```

```
x = 2,
```

```
y = 1,
```

```
z = 2 ? ;
```

```
No
```

```
[Sets] <- x In {y} + {z} & y=1.
```

```
Floundered. Unsolved goals are:
```

```
Goal: {v_1}=v_2
```

```
Delayed on: v_1
```

```
Goal: v_3={1}+v_2
```

```
Delayed on: v_2
```

```
Goal: Sets:(v_4 In v_3)
```

```
Delayed on: v_3
```

## 5 Interface to Prolog and C

In this section we explain how to call Prolog from Goedel. By using the C interface in SICStus Prolog, and the Gödel to Prolog interface, you can access C code from your Gödel programs. Please refer to *SICStus Prolog User's Manual* [Carlsson&Widén 92] for the details of the C interface.

Almost every symbol, and every user-declared symbol, in a Gödel program is given its *flat name* when translated into Prolog, to ensure its uniqueness despite any overloading. Flat names look like this:

$$\langle Module \rangle . \langle Symbol \rangle . \langle Category \rangle . \langle Arity \rangle$$

where  $\langle Module \rangle$  is the name of the module in which the symbol is declared,  $\langle Symbol \rangle$  is its declared name,  $\langle Category \rangle$  is one of the letters (B, S, C, F, O, P) depending on whether the symbol is a base, constructor, constant, function, proposition or predicate respectively, and  $\langle Arity \rangle$  is its arity (a positive integer, or 0 for Bases, Constants and Propositions). So a predicate + declared in a module called `Utilities` will become `'Utilities+.P2'` when compiled. Notice the quotes that are needed for Prolog to accept this as a predicate symbol.

The only important exceptions to this translation rule are integers and lists, which are both translated directly into their Prolog equivalents, so that we can take advantage of Prolog's support for these data types. There are several other Gödel system data types that are also handled specially, such as strings, sets and flocks, but it is not advisable or necessary to touch these or any other system data type in Prolog code; the system modules should provide a sufficient kit of operations for these types.

In order to call Prolog from Gödel, you need first to declare a predicate in a Gödel module, and then provide an definition in Prolog for the flat name of this predicate, taking care also to use the flat names for any constant or function symbols that appear in terms passed to or from Gödel. There is, however, a slight complication caused by negation. Inside negation, Gödel commit is disabled. Since we simulate commit with Prolog cut, we need a second copy of the compiled program without cuts for use with negative calls. This copy has a `~` character prepended to the flat name of all predicates: `'Utilities+.P2'`. You must provide a definition for this negative version of any predicate you implement in Prolog, otherwise any negated calls (such as in the condition of an IF-THEN-ELSE) will behave incorrectly. Normally the negative version can simply call the positive one. The Prolog code will then be invoked whenever the Gödel predicate is called.

Take care with delay declarations. Calls to a Gödel predicate that has delay declarations but no definition (because the definition is in Prolog) will not delay as expected. If you need delays, it is simplest to give all Gödel predicates with delays a definition in Gödel; such a definition might simply be a single statement that calls directly another predicate which is not delayed and implemented in Prolog.

Every Gödel module is compiled into a SICStus module with the same name. The Gödel compiler will expect to find the code for predicate declared in a particular module in that same module, and SICStus modules must occupy single files. Consequently, Prolog code defining a Gödel predicate must be incorporated in the file containing the Prolog code compiled from the Gödel module that declares it. The Gödel system will do this for you automatically if you put your prolog code in a file called *<Module>.sup* where *<Module>* is the name of the Gödel module.

For example, if you have a Gödel module *M* consisting of files *M.exp*, *M.loc* and *M.sup*, when it is compiled *M.q1* will include the Prolog code in *M.sup*. Note that you may get into trouble if you attempt meta-programming (such as partial evaluation) with an object program containing user modules with Prolog additions.

The following rather contrived example should make the mechanism clear. It shows naive reverse partially implemented in Prolog, but using the append predicate from the *Lists* system module.

```

-----
MODULE    Naive.

IMPORT    Lists.

PREDICATE Rev : List(a) * List(a). % Reverse a list

DELAY Rev(x, _) UNTIL GROUND(x).

Rev(xs, ys) <- Rev1(xs, ys).

PREDICATE Rev1 : List(a) * List(a). % Implemented in Prolog
-----

% Naive.sup, Prolog code for Rev1

'Naive.Rev1.P2'([], []).

'Naive.Rev1.P2'([X|Xs], Ys) :-
    'Naive.Rev1.P2'(Xs, Zs),
    'Lists':'Lists.Append.P3'(Zs, [X], Ys).

'~Naive.Rev1.P2'(Xs, Ys) :-
    'Naive.Rev1.P2'(Xs, Ys).
-----

```



## 6 Some hints for good programming in Gödel

The following offers some general hints on good programming style and how to use the Gödel system effectively.

- Gödel programs are very close to logic specifications. Usually the program itself serves as the specification. It is highly recommended that the naming of variables and symbols be done carefully in the sense that the name should reflect the intended interpretation of the object. Sometimes, it is difficult to give an object a good name. But remember that if you cannot find a name for a symbol, the idea behind it may be wrong.
- Singleton variables are most often produced by misspelling and unfinished clauses. So do not ignore singleton variable warnings and unused quantified variable warnings.
- Be careful of the limitations of safe negation, and note that all pruning is disabled within the condition of an IF-THEN-ELSE as well as within explicitly negated calls. It is better to write procedures to be determinate wherever possible without using commit to enforce determinacy. Experience suggests that it is a bad idea to use the failure of a procedure to return information, even the information that an error occurred, because testing for failure may compromise the efficiency of the procedure. Return error indications in arguments instead.

Here are answers to some common questions:

- Why does my program flounder?

It is our experience that if the logic of a program is correct the program should run. When a goal flounders, it is useful to look at the quantifiers, negations, IF-THEN-ELSE constructs and intensional sets. The floundering message may give you some clue which part of program to look at. Particularly, local variables in the condition part of IF-THEN-ELSE should all be quantified. Beware also of attempts to unify nonground sets.

- How can I make my Gödel program run faster?

The underlying Prolog system supports first argument indexing and tail recursion optimisation. Because the mapping from Gödel to Prolog is direct, the Gödel system does too. When properly used, first argument indexing and tail recursion optimisation can improve the speed by a factor of anything from 2 to 100.

- How can I write more efficient constraint solving programs?

The Bristol Gödel system supports only coroutining with exhaustive search. Sophisticated constraint solving methods have not yet been implemented. It would be

hard to write very efficient constraint solving programs in Gödel. However, there are a few rule of thumbs to help to improve the efficiency.

- Break down large constraints into smaller ones. This splits up the search space and programs may benefit from early failure. In the case of solving the SEND + MORE = MONEY puzzle, the Bristol Gödel takes a very long time to solve the following:

```
[Sets] <- 1000*s + 100*e + 10*n + d
          + 1000*m + 100*o + 10*r + e
          = 10000*m + 1000*o + 100*n + 10*e + y &
          1=<m=<9 & 0=<d=<9 & 0=<e=<9 &
          0=<r=<9 & 0=<n=<9 & 1=<s=<9 &
          0=<o=<9 & 0=<y=<9 &
          Size({s,e,n,d,m,o,r,y}, 8).    % the digits have
                                          % to be different.
```

But it can solve the following in a few seconds:

```
[Sets] <- y = (d+e) Mod 10 &
          sum1 = n + r + (d+e) Div 10 &
          e = sum1 Mod 10 &
          sum2 = e + o + sum1 Div 10 &
          n = sum2 Mod 10 &
          sum3 = s + m + sum2 Div 10 &
          o = sum3 Mod 10 &
          m = sum3 Div 10 &
          1=<m=<9 & 0=<d=<9 & 0=<e=<9 &
          0=<r=<9 & 0=<n=<9 & 1=<s=<9 &
          0=<o=<9 & 0=<y=<9 &
          Size({s,e,n,d,m,o,r,y}, 8).
```

- If there are producer and consumer goals (and they have proper delay declarations), put the consumers before the producers. This invokes the corouting mechanism which in many cases leads to early failures so as to reduce search space. In the program EightQueen, which can be found in the book *The Gödel Programming Language* and the Gödel release package under directory /programs/demo,

```
<- Safe(x) & Permutation([1,2,3,4,5,6,7,8], x).
```

is about 19 times faster than

```
<- Permutation([1,2,3,4,5,6,7,8], x) & Safe(x).
```

in finding the first answer.

- Arrange goals with a small search space before those with a large search space.

- What are Gödel's counterparts to Prolog *var*, *nonvar*, *=..*, *assert* and *retract*?

Prolog's *var*, *nonvar*, *assert* and *retract* are not supported in Gödel because they are not declarative. Univ (=..) would break Gödel's type system. However, the meta-logical use of *var* and *nonvar* is supported by Gödel meta programming facilities, i.e. `Variable/1` and `NonVarTerm/1` in the `Syntax` module.

It is a common practice among Prolog programmers to write multi-moded predicates (i.e. predicates that can run “backwards”) by using *var* to select between several (hopefully equivalent) logics according to the mode of the call. Because Gödel does not provide *var*, it is not possible to use this trick to have one entry point for several procedures, and only in simple cases can efficient multi-moded procedures be constructed. Where a relation is required to be computed in different modes, the recommended technique is to implement a distinct predicate for each mode, with a name suited to its mode. The modes can be enforced by delay declarations. Of course, this means that the mode must be known at the point of call, but in practical programs this is rarely a problem.

The meta-logical use of Prolog's *assert* and *retract* is provided by Gödel's meta-programming facilities, i.e. `InsertStatement/4` and `DeleteStatement/4` respectively in the `Programs` module. There is no equivalent to *assert* and *retract* for implementing global variables in Gödel; instead add an extra argument to the relevant predicates in order to pass around state information.

## 7 Facilities absent from this release

Several facilities of the Gödel language are not implemented in this release.

1. The occur check is not implemented.
2. Full commit is not supported. Only bar commit and one solution commit are allowed. Co-routining across the bar commit and one solution commit causes the commit to be switched off.
3. Implementation of constraint solving is primitive as has been discussed in Section 4.4.
4. The Gödel system utilities: `script-view`, `theory-compile` and `theory-decompile` are not available.
5. The following system modules are not available: `Floats`, `Numbers`, `NumbersIO`, `Theories`, and `TheoriesIO`.

## 8 Known bugs and limitations

There are a few known bugs or limitations which we have not had time to fix or cannot be fixed in this implementation. These are as follows:

1. It is known that parsing Gödel programs is an NP-complete problem, owing to the overloading of symbols causing difficulty in type checking [Henderson 94]. The Bristol Gödel parser works efficiently for most application programs, but its approach is simple and it is not hard to find examples that break it. Generally, the parser is slow at sorting out heavily ambiguous expressions; for example, it takes a few minutes for the parser to parse `x = 1+2+3+4+5+6+7+8+9` in a module which imports `Integers`, `Rationals` and `Sets` modules, because `+` has been declared in all these three modules. One clumsy solution to the problem is to break up the expression into different clauses. Future releases may improve on examples of this sort, but the general problem is here to stay.
2. If the Gödel system is compiled using SICStus 2.1 #8 (or earlier versions), a Gödel string is limited to 512 characters. Gödel systems compiled with SICStus 2.1 #9 (or later versions) will not have this problem.
3. Arithmetic functions and repeated variables in the head of a `DELAY` declaration may cause `DELAY` declarations to behave incorrectly in some rare cases.
4. In both the `Integers` and `Rationals` modules, if you use the power function backwards to compute the root of a number, when the answer is greater than  $10^{14}$ , it is usually wrong. For instance,

```
[Integers] <- x^2 = 10^28.  
x = 1000000000000000 ?  
Yes  
[Integers] <- x^2 = 10^30.  
No.
```

This is because there is no general algorithm for this class of problems and the inaccurate logarithm function has to be used.

5. No opaque terms are generated by the meta modules. This has little practical consequence, but means that it is sometimes possible to access the internal structure of terms that should be hidden inside closed system modules. Programs that rely on this will cease to work once opaque terms are implemented correctly.
6. The `Succeed` and `Compute` predicates in `Programs` can go wrong in certain very obscure circumstances where the object program itself calls `Succeed` or `Compute` on a different object program with the same name.

7. If you use SICStus Prolog 2.1 #6 (or earlier) to compile the Gödel system, you may be affected by a SICStus bug which causes a flounder message to be printed out even when your program has terminated normally. This bug was fixed in SICStus 2.1 #7 (or later versions).

## 9 Acknowledgements

John Lloyd and Pat Hill are the originators of Gödel. Constructive criticisms and testing and debugging efforts were contributed by Dominic Binks, Corin Gurr, Feliks Kluzniak, Nick Moffat and André de Waal. Corin Gurr implemented the `Syntax` module and assisted with the implementation of other meta modules. We thank Mats Carlsson and Stefan Andersson at SICS for their prompt and unfailing technical support.

This work was partly supported by the ESPRIT Basic Research Action 3012 (Compu-log) and Project 6810 (Compulog 2), SERC Grants GR/F/26256, and the University of Bristol.

## 10 References

[Carlsson&Widén 92] M. Carlsson & J. Widén. *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, August 1992.

[Henderson 94] F. Henderson. *NP-completeness*. March 1994.

[Hill&Lloyd 94] P.M. Hill & J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.