GCLA User's Manual

by

Martin Aronsson
Swedish Institute of Computer Science

GCLA User’s Manual

Februari 1994

T91:21A

Martin Aronsson

Swedish Institute of Computer Science
PO Box 1263
S-164 28 KISTA, Sweden

This manual corresponds to GCLA version 0.5.

This manual is based where possible on Sicstus Prolog User’s Manual by Mats Carlsson and Johan Widén.
# Table of Contents

**Introduction** ......................................................... 1

1 **How to Run GCLA** .............................................. 2
   1.1 Getting started ............................................. 2
   1.2 Reading in Programs ....................................... 2
   1.3 Queries ....................................................... 4
   1.4 Undefined Terms ........................................... 4
   1.5 Program Execution and Interruption ....................... 4
   1.6 Information About the State of the Program .............. 5

2 **Debugging** ......................................................... 6
   2.1 Basic Debugging Predicates ............................... 6
   2.2 Leashing ...................................................... 6
   2.3 Spy-points ................................................... 7
   2.4 Format of Debugging Messages ............................. 7
   2.5 Options Available During Debugging ...................... 8
   2.6 Advanced Options ......................................... 9
   2.7 Proviso Debugging ....................................... 9

3 **Loading Programs** ................................................ 11
   3.1 Saving and Loading Definitions ......................... 11
   3.2 Loading and Compiling Rules ............................. 12
       3.2.1 Declarations ....................................... 13

4 **Writing Rules** .................................................. 15
   4.1 Rules ......................................................... 15
   4.2 Strategies .................................................. 17
   4.3 Provisos ..................................................... 18
       4.3.1 Built-In Provisos .................................. 18
       4.3.2 Index Functions .................................... 20
   4.4 An Example: Some of the Default Rules .................. 21

5 **Examples** ........................................................ 24
   5.1 A Small Expert System ................................. 24
   5.2 Default Reasoning ...................................... 25
   5.3 A Functional Program ................................... 27
5.4 Mixing Relational and Functional Programming ..................... 29

6 Statistic package ................................................. 32
   6.1 Getting Information ........................................ 32

7 Formal Syntax ..................................................... 35

References ........................................................... 39
Introduction

GCLA is a programming system developed at the Swedish Institute of Computer Science (SICS) [Hal91, Aro90, Aro92, Kre91]. GCLA is best regarded as a logic programming language, although it shares some features commonly found among functional languages. One of the main objectives is to provide a powerful tool which supports the development of knowledge based systems.

For an introduction of how to program the GCLA system, the reader is recommended to consult "A Survey of GCLA: A Definitional Approach to Logic Programming" [Aro91], "GCLAI, A Definitional Approach to Control" [Kre91], and "Programming methodology and techniques in GCLA" [Aro92].

This manual describes the system developed at SICS. The system consists of a runtime system written in Sicstus Prolog and GCLA, and a preprocessor/compiler written in Prolog. There is a library containing some files implementing common used default rules and strategies, and an example library.

This manual is based where possible on the Industrial Sicstus Prolog User's Manual by Mats Carlsson et. al.

Notational Conventions

Defined atoms in GCLA are distinguished by their name and their arity. The notation name/arity is therefore used when it is necessary to refer to a defined atom unambiguously; e.g. append/3 specifies the term which is named "append" and which takes 3 arguments.

When introducing a built-in defined atom, we shall present its usage with a mode spec which has the form name(arg, ..., arg) where each arg can be one of the forms: +ArgName – this argument should be instantiated in goals for the defined atom. -ArgName – this argument should not be instantiated in goals for the defined atom. ?ArgName – this argument may or may not be instantiated in goals for the defined atom.

When referring to program code or keyboard characters, these are written like foo for an actual condition, and bar for an arbitrary condition.
1 How to Run GCLA

1.1 Getting started

To start GCLA, issue the command `use_module(library(gcla))` to Sicstus’ top level. When GCLA is initialized, it first loads a set of default inference rules, and then looks for a file "/.gclarc, and consults it, if it exists. From this file other files can be reloaded or included, and the global environment can be changed, but one cannot define GCLA clauses in this file. GCLA programs must be loaded through the loading primitives (see below).

To install your own runnable gcla, run the command `make_gcla(GCLAsavedfile)`, where `GCLAsavedfile` is the name that you want to use from the unix shell. This command creates a Prolog save file, which can be executed directly from the shell. The GCLA system has no own top level, but uses Sicstus’ top level.

1.2 Reading in Programs

A GCLA program consists of two parts: One part is used to express the declarative content of the program, called the definition or the object level, and the other part is used to express rules acting on the declarative part, called the rule definition or meta level, or just the rules. For a complete description of how GCLA works, we refer to [Kre91, Aro92]. When the system is started, it loads the default rule definition in the file `rules.rul`, consisting of the basic rules. Some of the rules are described in section 4.4. To load another set of rules, use the built in command

```prolog
| ?- reload_rules('myrules.rul').
```

which completely erases the current set of rules and loads the rules in the file `myrules.rul`.

The command

```prolog
| ?- include_rules('additional_rules.rul').
```
includes the rules in additional_rules, i.e. does not throw away all the current rules, as the 
command reload_rules does. However, rules defined in the named file replaces rules in the old 
definition, if they are not add_to_rule - declared (see section 3.2.1).

To load a definition, use the built-in command

\[ \text{?- reload_def('mydef.def')}. \]

which loads the definition in the file mydef.def into the system. The old definition is completely 
lost.

The command

\[ \text{?- include_def('additional_defs.def')}. \]

includes the definition 'additional_defs.def' into the current loaded definition. Any atoms defined 
in the file erases any clause for that atom already present in the current definition.

If the rule file and the definition file have the same name, say my_application, one can use the commands

\[ \text{?- reload(my_application)}. \]

to reload both my_application.def and my_application.rul, and

\[ \text{?- include(my_application)}. \]

to include both my_application.def and my_application.rul.

The files may have all types of extensions, but we recommend that the rule file and the definition 
file have the same name and the above extensions.
1.3 Queries

There are two derivability symbols, one for the object level derivations, denoted by '\-\', and one for the rule derivations, denoted by '\\-\'.

A generic query looks like

```
| ?- rule \\- antecedent \- consequent.
```

If '\\-\' is omitted, the query is interpreted as

```
| ?- gcla \- antecedent \- consequent.
```

and if there is no antecedent, the query is written

```
| ?- rule \\- (\- consequent).
```

or

```
| ?- \- consequent
```

1.4 Undefined Terms

The feature for generating errors on undefined predicates should be turned off (i.e. all predicates that are undefined should fail and not be treated as errors). Otherwise the GCLA interpreter will not perform as it should. (It is turned off when the GCLA system is loaded).

1.5 Program Execution and Interruption

The execution of a GCLA program starts with a query to the interpreter. To abort the execution the original abort facility of Sicstus should be used. If the GCLA flag gcla_interruption is set
to yes, and the current query is executed by the debugger, the execution can be stopped by typing control-C.

1.6 Information About the State of the Program

gcla_listing
  lists all the clauses in the current definition.
gcla_listing(+N/A)
  lists all clauses in the current definition with name N and arity A.
gcla_listing(+N)
  lists all clauses in the current definition with name N.
gcla_listing_rules
  lists all the rule- and strategy clauses currently loaded.
gcla_listing_rules(+N/A)
  lists all the rule- and strategy clauses with name N and arity A currently loaded.
gcla_listing_rules(+N)
  lists all the rule- and strategy clauses with name N currently loaded.
gcla_flag(+Flagname, +Value)
  sets the gcla flag Flagname to the value Value. Currently, the following GCLA flags are defined:
gcla_interruption
    If set to yes, the debugger can be interrupted by typing control-C.
write_asserting_text
    If set to yes, various output is produced during loading of GCLA code.
load_default_rules
    if set to yes, the file lib('rules.rul') is loaded when GCLA is initialized.
2 Debugging

The debugging package of GCLA works in much as Prolog's ordinary debugger, i.e. uses the box model. We assume some familiarities with an ordinary Prolog debugger. The debugger is centered around the rule code, i.e. spypoints refer to rules or strategies, the debugger creeps from one rule call to the next etc.

2.1 Basic Debugging Predicates

The basic commands are

`gcla_trace`

Switches the debugger on, if it is not already. The debugger will first creep, stopping at the first GCLA call. The possible options here are listed below (section 2.5).

`gcla_notrace`

Switches the debugger off. However, spypoints are kept in memory, and if the debugger is turned on again, these spypoints will be activated.

2.2 Leashing

Leashing is used to inform the debugger when it should stop the execution and write out a debugging message. There are four points where the debugger could stop; when the execution enters a rule or strategy (call), when it exits a rule or strategy (exit), when backtracking occurs and a rule or strategy is tried again (redo) and when a rule or strategy fails (fail).

`gcla_leash(+Mode)`

Leashing mode is set to Mode. Leashing Mode determines the points at which you are to be prompted when you creep through your program. Mode can be a subset of the following, specified as a list.

- call Prompt on call
- exit Prompt on exit
- redo Prompt on redo
- fail Prompt on fail

Leashing Mode is initially set to [call, exit, redo, fail].
2.3 Spy-points

Spy-points can be set at rule names and strategy names, i.e. if a spy-point is placed at a certain rule or strategy, the debugger will stop when that rule or strategy is to be invoked. There is no way of spying terms in the antecedent or spying a consequent. Spy-points are set and removed by the following built-in primitives, which are also declared as operators (i.e. gcla_spy(foo) can also be written gcla_spy(foo)):

\[\text{gcla\_spy} + \text{Spec}\]

Sets a spy-point on the rule or strategy given by Spec. Spec should be the main functor of the rule or strategy where the debugger should halt.

\[\text{gcla\_nospy} + \text{Spec}\]

Removes +Spec from the list of functors that are spyped.

\[\text{gcla\_nospyall}\]

Removes all spypoints.

The options available when you arrive at a spy-point are described below (section 2.5)

2.4 Format of Debugging Messages

The basic format is

\[S \ 3 \ 4 \ \text{CALL} \ \text{Rule} \ \text{\textbackslash\textbackslash} \ \text{Antecedent} \ \text{\textbar} \ \text{Consequent} \ ?\]

- \(S\) is a spy-point indicator: It is printed as '+ ' if the current rule \textit{Rule} is a spy-point, and as ' ' if not.
- \(3\) is the unique invocation identifier. It increases whether or not the actual invocations are seen or not. It is not reset during backtracking. It gives the number of invocations that the system has performed since the start of the execution (of the top level query).
- \(4\) denotes the depth of the current invocation, i.e the number of ancestors to this goal.
- The next word specifies the current port, if it is an entry or return port (one of \textit{CALL}, \textit{EXIT}, \textit{REDO}, \textit{FAIL}). In this case \textit{CALL}.
- \textit{Rule} is the current rule.
- \textit{Antecedent} is the current antecedent
- \textit{Consequent} is the current consequent
2.5 Options Available During Debugging

This section describes the particular options that are available when the system prompts for input during debugging. All options are one letter, of which some could have an optional integer. Not all options are available at all ports.

- = remove spypoint
  c = creep,
  s = skip,
  a = abort
  @ = directive
  l = leap
  f = fail
  p = proviso trace, only applicable at CALL ports
  + = add spypoint
  RET = creep
  s xxx = skip to call number xxx
  h = this text
  r = retry
  r xxx = retry at call number xxx
  n = notrace

**c RET**
Creep causes the interpreter to step to the very next port and print out the debugging message.

**l**
Leap causes the interpreter to resume running the program, only stopping when a rule that is declared as a spypoint is reached.

**s**
Skip is only valid for Call and Redo ports. It skips over the entire execution of this goal, until the goal either succeeds or fails, where the appropriate debugging message is given and control is given to the user. All spypoints are ignored.

**s xxx**
Skip can be supplied with an integer, in which case the debugger skips to that particular invocation.

**r**
Retry restarts an invocation from the beginning. The state is the same as when the goal was tried first. When a retry is performed the invocation counter is reset to the goal's original number.

**r xxx**
Retry can be supplied with an integer, in which case the debugger tries to retry that particular invocation. If that is not possible, the debugger resumes the execution at the nearest invocation before the supplied number. The debugger does this by failing until the right invocation is reached.

**f**
Fail causes the invocation to fail prematurely.

**n**
Nodebug switches the debugger off, and the execution is finished without debugging messages.

**+**
Spy this. Set a spypoint at this rule.

**-**
Nospy this. Remove spypoint from this rule.

**a**
Abort causes an abortion of the current execution. The interpreter returns to the top level.
Command, gives you the possibility to execute arbitrary calls to the system while debugging. The command is read and executed as if it was given at the top level. After the call is executed, the debugger resumes its execution.

Proviso trace, starts the proviso trace, which is described in its own section.

Help prints out the table of options.

2.6 Advanced Options

This section describes some advanced primitives to change the behaviour of the debugger.

trace_print_condition(Cond)

User defined Prolog predicate. If defined, the conditions (i.e. every element of the antecedent and the conclusion of a sequent) will be printed according to this predicate. If the predicate fails, the ordinary tracer primitives will be used.

trace_print_rule(ProofTerm)

User defined Prolog predicate. If defined, the proof term in a trace printout will be printed through this primitive. If trace_print_rule fails, the ordinary built in primitive will be used.

user_defined_stop(ProofTerm, Antecedent, Consequent)

User defined predicate. If defined, and the execution is in leap mode, the debugger stops when user_defined_stop is true (or a spy point is reached, see section 2.3). This predicate gives the possibility to define points where the debugger stops, by looking at the sequents outlook.

WARNING! These predicates do not get through the GCLA preprocessor, and therefore one cannot rely on GCLA primitives. Also, one has to be very careful NOT to instantiate anything in the predicates. If these predicates are used, we strongly recommend that all unification is done explicitly through such primitives as var/1, arg/3, atomic/1, number/1, ==, \= etc.

2.7 Proviso Debugging

The proviso debugger can be invoked from the main debugger, by typing p to the main debugger. The format of the proviso debugger is

Proviso trace: CALL 3 Proviso ?
where

- The word after the string Proviso trace: specifies the current port, if it is an entry or return port (one of CALL, EXIT, REDO, FAIL). In this case CALL.
- $3$ is a sub index to the unique invocation identifier in the main debugger. It increases whether the actual invocations are seen or not. It is not reset during backtracking. It gives the number of invocations that the system has performed since the latest rule call in the main debugger.
- Proviso is the proviso call. It is not possible to see calls to some of the system defined meta constructions, such as if-then-else constructs, index functions etc.

The options available during proviso debugging are

\[
\begin{align*}
\text{c} &= \text{creep} & \text{CR} &= \text{creep} \\
\text{f} &= \text{fail} & \text{l} &= \text{leap} \\
\text{h} &= \text{help} & \text{s} &= \text{skip} \\
\text{q} &= \text{quit proviso tracing} & \text{a} &= \text{abort execution}
\end{align*}
\]

where most of the options work analogously as in the main debugger. leap is leaping in the main debugger, since there is no possibility to set spy points on arbitrary provisos, and quit is quitting the proviso debugger (but still be in the main debugger). abort is aborting the whole execution and return to the top level.

There is a shortcoming of the proviso debugger. If the user performs too many proviso debuggings from the main debugger, the system can get slow, and in extreme cases crash.
3 Loading Programs

Programs must be loaded through one of four primitives; 
reload_def, include_def, reload_rules and include_rules. When a definition is reloaded, 
the previous one is lost. Due to implementation limitations, one cannot alter the definitional 
clauses from within the GCLA system, unless they are dynamic_term-declared, and one cannot 
mix compiled and interpreted clauses (interpreted clauses are clauses that can be asserted and 
retracted by the provisos add and rem, see section 4.3.1).

3.1 Saving and Loading Definitions

Definitions may be loaded by using the primitives reload_def and include_def:

| ?- reload_def(+File).

The current definition in the systems memory is discarded, and the file's content is read into 
the system's memory. When a directive is read it is immediately executed. A directive is a term 
preceded by the symbol ':-' (for example, :- write('Ready ') is a directive). A directive could 
be any Prolog command, or GCLA query.

| ?- include_def(+File).

The current definition in the systems memory is kept, but if the file contains clauses that defines 
atoms that are already defined in the system's memory, the new definition replaces the old one. 
When a directive is read it is immediately executed. A directive is a term preceded by the symbol 
':-' (for example, :- write('Ready ') is a directive). A directive could be any Prolog command, 
or GCLA query.

Definitions can be saved in an internal format, to be reloaded later, by the primitive

| ?- save_defs(List,File).
where List is a list of Functor/Arity-definitions, and File is the name of the file where to store
the definitions. The stored definition can be read back again in two ways. It can be reloaded again
by the primitive

\[ \texttt{?- reload_compiled_def(File)} \]

in which case all currently loaded definitions are removed, and the compiled definition from File
is stored in the working memory. The saved definitions can also be included by the primitive

\[ \texttt{?- include_compiled_def(File)} \]

in which case the current working memory is kept, except for definitions overwritten by defini-
tions in File.

save_defs/2 and include_compiled_def/1 are useful when large databases are handled, in
which case they can be read through the GCLA preprocessor once, and then saved in the internal
format which is much faster to read into the working memory.

### 3.2 Loading and Compiling Rules

Rule definitions are loaded by using the primitives reload_rules and include_rules:

**reload_rules(+File)**

`reload_rules` discards the current set of rules, and reads the rules in the file `File` into
the system’s memory.

**include_rules(+File)**

`include_rules` does not discard the current set of rules, but adds the rules in the
named file `File` to the current rules in the system’s memory. If there is already a rule
with the same name in the system, that rule is overwritten with the new rule, if it is
not declared otherwise (see section 3.2.1).

*File* can have the following formats:
lib(File) where File is an atom, denotes a file File sought in the GCLA library directory.

File where File is an atom, denotes a file File sought in the current working directory.

reload_rules is intended to load the rules for a specific application into the system's memory, and include_rules is intended to load library files containing common rule definitions.

The current loaded control definitions (inference rules, strategies and provisos) can be compiled by the Prolog compiler, to gain efficiency. By the primitive

\[ \text{\texttt{| ?- compile_all_rules.}} \]

all currently loaded control definitions are compiled and are read back into the memory. However, it is not possible to mix compiled control code and interpreted control code, i.e. once compile_all_rules has been used, it is not possible to include other control definitions by using the primitive include_rules(File). One has to reload some control definition by the primitive reload_rules(File) again to remove the compiled code.

### 3.2.1 Declarations

:- multifile(+Functor/+Arity).

:- multifile(+Functor/+Arity, ..., +FunctorN/+ArityN)).

Multifile declaration can only be used in rule files, and only for proviso clauses. It causes the specified predicates to be multifile, which means that if more clauses are subsequently loaded from other files for the same proviso definition, the new clauses will not replace the old ones, but will be added instead. The old clauses are erased only if the proviso definition is reloaded from its "home file" (the file containing the multifile declaration), or if it is reloaded from a different file declaring it as multifile (in which case the user is queried first).

The declaration multifile is intended for adding clauses, which lists a certain property. For example the proviso definition constructor/2 defined in the GCLA default rules uses this declaration (see section 4.4).

:- dynamic_term(+Functor/+Arity).

dynamic_term declares definitional clauses defining terms with the principal functor functor and arity arity to be dynamic, i.e. it is possible to add and remove clauses to
the definition. It is recommended to use `dynamic_term` only together with the provisos add and rem.

`:- complete(+Functor/+Arity)`

`complete` declares the definition of `Functor/Arity` to be compiled to a complete definition, i.e. the extremal clause (or, if one prefers, the completion of the program) of the definition is calculated by the A-sufficiency algorithm. Normally the system does not calculate that clause due to efficiency reasons.

`:- add_to_rule(+Functor/+Arity)`

`add_to_rule` declares the clauses defined by the atom `Functor/Arity` in the file where `add_to_rule` occurs to be added to the currently loaded rule, instead of replacing it. This directive is useful for adding restrictions to general rules. The directive must appear in a file.
Chapter 4: Writing Rules

4 Writing Rules

This section describes how to define rules and strategies in GCLA. For a complete description of how rules and strategies work, we refer to [Kre91, Ar92]. Usually it is the strategies that are changed first. One wants to cut away some branches that either computes the same answer as another branch, or computes an unwanted answer, or does not terminate. Often new rules are created from old ones by restricting the terms that the rules work on.

There are some initial library files containing common rule definitions. A rule definition file contains three kinds of clauses:

- **Proviso clauses**, which are common Horn clauses, and thus could be any kind of Prolog clauses.
- **Inference rule definitions**, which define how different inference rules should act.
- **Strategies**, which are used to form search strategies among the rules.

These three parts form a specific interpreter for the object level.

It should be noted that in the theory the object level variables in the declarative part (i.e. the definition) are different from the meta level variables introduced in the rules (i.e. they are on different levels). This means that variables introduced in a definition are treated as constants in a rule. The only way to bind object level variables is in the primitives unify, clause and definiens, described below. However, in the current implementation the two levels are implemented as the same, and thus one has to be careful when and how variables are bound.

4.1 Rules

A definition of a rule has the following general form

\[
\text{Rule name}(A_1, \ldots, A_m, PT_1, \ldots, PT_n)\{C_1, \ldots, C_k\} \leftarrow \text{Proviso,} \\
(PT_1 \rightarrow \text{Seq_1}), \\
\ldots, \\
(PT_n \rightarrow \text{Seq_n}) \\
\rightarrow \text{Seq}.
\]
where

- $A_i$ are arbitrary arguments, commonly used for specifying which term in a sequent that this rule operates on (typically a partially instantiated term or an index).
- $Proviso$ is a conjunction of provisos, i.e. calls to Horn clauses defined elsewhere. The proviso could be empty.
- $Seq, Seq_i$ are sequents, in a special syntax, described below.
- $PT_i$ are proofterms, i.e terms representing the proofs of the premises $Seq_i$ of the rule.
- $Ci$ are constraints, i.e constraints on variables occurring in the arguments of the head of the rule. Currently, only constraints of the form $X \not\equiv Y$ is supported, where the interpretation is 'X is not equal to Y', where X and Y may be any term. A clause where the macro $else$ occurs is to be regarded as a catch-all clause, i.e. the guard is expanded so that all cases where there does not exists another head (and guard) of some clause of the definition is caught by this clause. There can only be one clause with an else-guard for each definition of $RuleName/m*n$.

Provisos are executed from left to right. New sequents (i.e. the rule's premises) are also executed from left to right. Sequents are on the form $Antecedent \leftarrow Consequent$, where Consequent is an ordinary GCLA term. Antecedent is a list, where two operators can be used:

- The cons operator, $\langle$, which concatenates a condition to the antecedent list.
- The append operator, $\otimes$, which appends two antecedent lists.

Examples of such antecedent lists are $[c1,c2]\otimes[c3][c4]$, which equals $[c1,c2,c3,c4]$, and $[c1]\otimes[c2,c2]$ which equals $[c1,c2,c3]$.

An example of a rule is the rule definition-right:

```
d_right(C, PT) <=
  atom(C),
  clause(C, B),
  (PT -> (A \leftarrow B))
  -> (A \leftarrow C).
```

$\%$ Head of rule
$\%$ Proviso
$\%$ Proviso
$\%$ New call to rule PT (premise of rule)
$\%$ Original sequent (conclusion of rule)

Another example is the rule arrow-left:
4.2 Strategies

Strategies are used to constrain search of proofs in the formal system set up by the inference rules. A strategy definition has the following general form:

\[ \text{Strat}\{C_1, \ldots, C_k\} <= PT_1, \ldots, PT_n. \]

or

\[ \text{Strat}\{C_1, \ldots, C_k\} <= \\
(Proviso_1 -> Seq_1), \\
\ldots, \\
(Proviso_k -> Seq_k). \\
\text{Strat} <= PT_1, \ldots, PT_n. \]

A strategy has always a clause with a vector, implemented as a comma-separated structure of rule- or strategy-calls. This vector should be interpreted as a nondeterministic choice of the next rule or strategy call. Proviso can be omitted, in which case \((Proviso -> Seq_i)\) is replaced with \(Seq_i\).

An example of a strategy is

\[ a rl <= \\
  \text{axiom}(\ldots), \quad \% \text{First try the axiom rule} \\
  \text{right}(arl), \quad \% \text{then try the rules to the right} \\
  \text{left}(\ldots, arl). \quad \% \text{and last the rules to the left} \]

Another example is
\[
\text{left}(T, I, PT) \iff \begin{cases}
(I \circ (T \downarrow I) \downarrow \_).
& \% \text{First chose a condition } T \\
\text{left}(T, I, PT) \iff \begin{cases}
\text{d_left}(T, I, PT),
& \% \text{then chose an appropriate rule for } T \\
\text{a_left}(T, I, PT).
& \% \text{Try the d_left rule or try the a_left rule}
\end{cases}
\end{cases}
\]

where a particular condition is chosen in the first clause, and the execution continues with either \text{d_left} or \text{a_left}.

### 4.3 Provisos

There are a number of built-in provisos, which are listed below. To define a new proviso just define a new clause with the ':-' constructor. A new proviso can of course make use of built-in provisos.

An important thing to remember is that binding object variables should only be done with the primitives \text{unify}, \text{definiens} and \text{clause}. In the theory it is not possible to bind an object level variable to a meta level condition/term, but the current implementation does not check the level of a variable, so one has to be careful when and how variables are bound.

#### 4.3.1 Built-In Provisos

\textit{functor}(+Atom, ?Functor, ?Arity)
\begin{itemize}
  \item functor gets the principal functor \textit{Functor} of \textit{Atom}, together with its arity \textit{Arity}.
\end{itemize}

\textit{arg}(+Atom, +Number, -Arg)
\begin{itemize}
  \item arg relates the \textit{Number}th argument of \textit{Atom} and binds \textit{Arg} to that term.
\end{itemize}

\textit{term}(+Term)
\begin{itemize}
  \item term is true if \textit{Term} is
  \begin{itemize}
    \item a variable
    \item an atom
  \end{itemize}
  Otherwise term is false.
\end{itemize}

\textit{atom}(+Atom)
\begin{itemize}
  \item atom is true if \textit{Atom} is
  \begin{itemize}
    \item a number
  \end{itemize}
\end{itemize}
Chapter 4: Writing Rules

- a symbol
- a structure other than given by the table constructor(Functor, Arity)

Otherwise atom is false.

not(+Call)

not is true if all derivations of Call is false.

unify(+Term1, +Term2)

unify unifies Term1 and Term2. If this unification succeeds, unify is true, otherwise false.

definiens(+Atom, -Dp, ?N)

definiens calculates the definiens operation, which in principle is: for a given atom, collect all clauses' bodies that defines that atom. The operation is described in [HS-H91,Kre91]. If there could be more than one substitution binding variables in +Atom, definiens will calculate them upon backtracking. Dp is bound to the resulting bag of that operation, each body separated by ';'. If there is no definition of Atom, Dp is bound to the symbol false. N is the number of clauses that defines Atom (equal to the number of elements in Dp, and 0 if Dp equals false). The bodies in the bag are ordered corresponding to the order in the current definition. definiens is nondeterministic, i.e. it generates next possible Dp upon backtracking.

Since the mgu algorithm does not have an occurs check, there is a risk that this operation constructs a circular structure, which can cause the GCLA system to loop. For example, with the definition

```
foo(f(X),[X|R]) <= ...
foo(Y,[Y|R]) <= ...
```

definiens(foo(X,Y),Dp,N) creates the structure f(f(f(f(...)))).

inst(+X,+C)

The object level variable X is instantiated to a new variable in the condition C. Currently this proviso performs no operation in the runtime system, where relies on that X is only used inside C and nowhere else. When inst/2 is found during loading of a file, X is replaced with a new variable in C.

clause(+Atom, -Body)

If a clause A <= B exists in the program, and Atom and A are unified successfully, B is returned in Body. If there are no clauses unifiable with Atom, the symbol false is returned in Body.

add(+Clause)

add asserts the clause Clause into the system's object level. The change is local to the rest of the execution, and disappears when the execution either is finished or when the system backtracks over add. Clause must be declared dynamic_term (see section 3.2.1).
rem(\(+\)Clause)

rem removes all clauses that are unifiable with Clause in the system’s current object level. No variables are bound, and 0 or more clauses are removed. This primitive succeeds always. The change is local to the rest of the execution, and disappears when the execution either is finished or when the system backtracks over rem. Clause must be declared dynamic_term (see section 3.2.1).

4.3.2 Index Functions

Index functions are a way in GCLA to generate bags of terms. The system defined index function is a proviso call to the system defined function i/2, which as a result returns the bag. The syntax is

\((\text{ListOfTerms} \ i \ \text{ProvisoCallOrRuleCall}) \rightarrow \text{Bag}\)

where ListOfTerms is a list of terms, whose instantiations are collected in Bag. ProvisoCallOrRuleCall is the call that generates the instantiations. An example of a rule that uses an index function is bagof_right/3

\[
\text{bagof\_right}(\text{bagof}(T,C,L),\text{PT}) \leftarrow \\
([T] \ i \ \text{PT}(\text{PT} \rightarrow (A \ \leftarrow \ C)) \rightarrow \text{L}) \rightarrow \\
(A \ \leftarrow \ \text{bagof}(T,C,L)).
\]

The operator */2 is used to existentially bind a variable inside a condition, in the example above is PT existentially bound inside the rule call \((\text{PT} \rightarrow (A \ \leftarrow \ C))\), which means that if PT contains some variables, these variables can take different values during the generation of the bag. Note that this only applies to occurrences outside the actual call, i.e. if we write

\[
\text{bagof\_right}(\text{bagof}(T,C,L),\text{PT}) \leftarrow \\
([T] \ i \ \text{PT}(\text{PT} \rightarrow (A \ \leftarrow \ C-C)) \rightarrow \text{L}) \rightarrow \\
(A \ \leftarrow \ \text{bagof}(T,C,L)).
\]

we have to have an inference rule for */2 in the control definition, an example can be found in the Example section later (sigma_right/2).
4.4 An Example: Some of the Default Rules

This example is a file containing some default rules, corresponding to some of the rules that
are loaded when GCLA is started. The default rules actually loaded can be found in the file
siestus/Library/gcla/rulelib/rules.rul. Other examples can be found in /gcla/examples/.
By issuing the directive

\[ ?- \text{absolute_file_name(library('gcla/examples/'),Path).} \]

the path can be found.

\[
\begin{align*}
\text{\texttt{--------}} \\
\text{\texttt{\% Rules}} \\
\text{\texttt{\texttt{-} multife({constructor/2).}}} \\
\text{\texttt{true_right \texttt{\texttt{\_ \texttt{-} \texttt{true).}}} }} \\
\text{\texttt{false_left(I) \texttt{\texttt{\texttt{\_ \texttt{-}} \texttt{\_).}}} }} \\
\text{\texttt{axiom(T,C,I) \texttt{\texttt{\_ \texttt{-}} \texttt{term(T),}}} \\
\text{\texttt{\texttt{\_ \texttt{-}}} \texttt{term(C),}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{unify(T,C)} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(I\_}[\_|\_] \texttt{-} \texttt{\_).}}} \\
\text{\texttt{d_right(C,PT) \texttt{\texttt{\_ \texttt{-}} \texttt{atom(C),}}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{clause(C,B),}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(PT \texttt{-} \texttt{\_ \texttt{-}} \texttt{P \texttt{-} \texttt{\_ \texttt{-}}} \texttt{B))}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(P \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C).}} \\
\text{\texttt{d_left(T,I,PT) \texttt{\texttt{\_ \texttt{-}} \texttt{atom(T),}}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{definiens(T,Dp,N),}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(PT \texttt{-} \texttt{\_ \texttt{-}} \texttt{I\_}[\_Dp|\_Y \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C))}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(I\_}[\_T|\_Y \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C).}} \\
\text{\texttt{a_left((A \texttt{-} \texttt{-} \texttt{\_ \texttt{-} C1),I,PT,PT1) \texttt{\texttt{\_ \texttt{-}} \texttt{(PT \texttt{-} \texttt{\_ \texttt{-}} \texttt{I\_Y \texttt{-} \texttt{\_ \texttt{-}}} \texttt{A))},}}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(PT1 \texttt{-} \texttt{\_ \texttt{-}} \texttt{I\_}[\_C1|\_Y \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C))}} \\
\text{\texttt{\_ \texttt{-}}} \texttt{(I\_}[\_A \texttt{-} \texttt{\_ \texttt{-}} \texttt{C1|\_Y \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C).}}} \\
\text{\texttt{a_right((A \texttt{-} \texttt{-} \texttt{\_ \texttt{-} C),PT) \texttt{\texttt{\_ \texttt{-}} \texttt{(PT \texttt{-} \texttt{\_ \texttt{-}} \texttt{[A\_P] \texttt{-} \texttt{\_ \texttt{-}}} \texttt{C))}}}}
\end{align*}
\]
\rightarrow (P \rightarrow (A \rightarrow C)) .

o_right(1,(C1 ; C2),PT) <=
(PT \rightarrow (\text{Ass} \rightarrow C1))
\rightarrow (\text{Ass} \rightarrow (C1 ; C2)).

o_right(2,(C1 ; C2),PT) <=
(PT \rightarrow (\text{Ass} \rightarrow C2))
\rightarrow (\text{Ass} \rightarrow (C1 ; C2)).

o_left((A1 ; A2),I,PT1,PT2) <=
(PT1 \rightarrow (I\oplus[A1|R] \rightarrow C))
(PT2 \rightarrow (I\oplus[A2|R] \rightarrow C))
\rightarrow (I\oplus[(A1 ; A2)|R] \rightarrow C).

v_right((C1,C2),PT1,PT2) <=
(PT1 \rightarrow (A \rightarrow C1))
(PT2 \rightarrow (A \rightarrow C2))
\rightarrow (A \rightarrow (C1,C2)).

v_left((C1,C2),I,PT) <=
(PT \rightarrow (I\oplus[(C1,C2)|Y] \rightarrow C))
\rightarrow (I\oplus[(C1,C2)|Y] \rightarrow C).

pi_left((pi X \rightarrow C),I,PT) <=
\text{inst}(X,C),
(PT \rightarrow (I\oplus[C|R] \rightarrow C1))
\rightarrow (I\oplus[(pi X \rightarrow C)|R] \rightarrow C1).

sigma_right((X^C),PT) <=
\text{inst}(X,C,C1),
(PT \rightarrow (A \rightarrow C1))
\rightarrow (A \rightarrow (X^C)).

% Provisos

constructor(';',2).
constructor('->',2).
constructor(true,0).
constructor(false,0).
constructor(';',2).
constructor(pi,1).
constructor('^',2).

% Strategies
gcla <= arl.

arl <= axiom(_,_,_),
    right(arl),
    left(arl).
alr <= axiom(_,_,_),
    left(alr),
    right(alr).

lra <= left(lra),
    right(lra),
    axiom(_,_,_).

no_left <= axiom(_,_,_),
    right(no_left).

right(PT) <=
    v_right(_,PT,PT),
    a_right(_,PT),
    o_right(_,PT),
    d_right(_,PT),
    true_right.

left(PT) <= false_left(_),
    v_left(_,_,PT),
    a_left(_,_,PT,PT),
    o_left(_,_,PT,PT),
    d_left(_,_,PT),
    pi_left(_,_,PT).
5 Examples

This chapter describes some example programs, both the definition files and the control files. These and other examples can be found in in /gcla/examples/. By issuing the directive

\[ \text{?- absolute_file_name(library('gcla/examples/'),Path).} \]

the path can be found.

5.1 A Small Expert System

This is a small example of an expert system for diagnosing diseases. Of course the definition should contain much more information.

The definition contains the rules connecting symptoms and diseases, but contains no facts. The facts are submitted by the queries. Note the circular definition of disease, which is handled by the rules where the loop is detected by the not-equal checks.

Definition:

\[
\begin{align*}
\text{symptom(high_temp)} & \iff \text{disease(pneumonia)}. \\
\text{symptom(high_temp)} & \iff \text{disease(plague)}. \\
\text{symptom(cough)} & \iff \text{disease(pneumonia)}. \\
\text{symptom(cough)} & \iff \text{disease(cold)}. \\
\text{disease}(X) & \iff \text{disease}(X).
\end{align*}
\]

Rules and strategies:

\[
\text{:- include_rules(lib('rules.rul'))}. \\
\text{d_right}(C,PT) \iff \\
\text{atom}(C), \\
\text{clause}(C,B), \\
\text{not}(C = B),
\]

(PT -> (P - B))
-> (P - C).

d_left(T,X,PT) <=
atom(T),
definiens(T,Dp,N),
not(T = Dp),
(PT -> (X@[Dp|Y] - C))
-> (X@[T|Y] - C).

Queries:

1) Assuming high temperature, what possible diseases follows:

| ?- symptom(high_temp) \- disease(X) ; disease(Y).

X = pneumonia,
Y = plague ? ;

X = plague,
Y = pneumonia ? ;

no
| ?-

2) Try to find a disease that causes high temperature and coughing:

| ?- disease(X) \- symptom(high_temp),symptom(cough).

X = pneumonia ? ;

no
| ?-

5.2 Default Reasoning

This example has two main ingredients. Firstly, it implements default reasoning, and secondly it shows how negation is accomplished. It is the famous bird-penguin example: An object can fly if it is a bird and it is not a penguin, Tweety and polly are birds as well as all penguins, and Pendo is a penguin.
Definition:

flies(X) <=
  bird(X),
  (penguin(X) -> false).

bird(tweety).
bird(polly).
bird(X) <= penguin(X).
penguin(pengo).

Rules and strategies:

:- include_rules(lib('rules.rul')).

fs <=
  right(fs),
  left_if_false(fs).

left_if_false(PT) <=
  (_ \- false).
left_if_false(PT) <=
  no_false_assump(PT),
  false_left(_).

no_false_assump(PT) <=
  not(member(false,A)) ->
  (A \- _).
no_false_assump(PT) <=
  left(PT).

member(X,[X1_]).
member(X,[_1R]) :-
  member(X,R).

Queries:

1) Which birds can fly:

    |- ?- fs \- \- flies(X).
    X = tweety ? ;
\[ X = \text{polly} \ ? ; \]

no
|  ?-  

2) Which birds cannot fly?

|  ?- \( \text{fs} \ \\ \text{flies}(X) \ \false \).  
\[ X = \text{pengo} \ ? ; \]

no  
|  ?-  

5.3 A Functional Program

This example shows how a functional program can be written in GCLA. The definition defines the function \textit{add/2}, which adds its two arguments, which are in successor arithmetic. The definition of \textit{succ/1} and the third clause of \textit{add/2} are a so called evaluation schema, i.e. it evaluates its (first) argument.

Definition:

\[
\begin{align*}
\text{add}(0,X) &= X. \\
\text{add}(s(X),Y) &= \text{succ} \left( \text{add}(X,Y) \right). \\
\text{add}(X,Y) &\# \{ X \neq s(\_), X \neq 0 \} \leftarrow \\
&\quad \pi Z \left( (X -> Z) -> \text{add}(Z,Y) \right). \\
\text{succ}(X) &= \pi Y \left( (X -> Y) -> s(Y) \right). \\
\end{align*}
\]

Rules and strategies:

\[
\begin{align*}
:- \text{include_rules(lib('rules.rul'))}. \\
\text{d_left1}(T,I,PT) &\leftarrow \\
&\quad \text{not} \left( \text{canonical}(T) \right) \\
&\quad \leftarrow (I \\leftarrow \_). \\
\text{d_left1}(T,I,PT) &\leftarrow \text{d_left}(T,I,PT). \\
\end{align*}
\]
\texttt{left(PT) <= v\_left(\_,\_,PT),}
\texttt{a\_left(\_,\_,PT,PT),}
\texttt{o\_left(\_,\_,PT,PT),}
\texttt{d\_left(\_,\_,PT),}
\texttt{false\_left(\_),}
\texttt{pi\_left(\_,\_,PT),}
\texttt{canonical(X) :- var(X).}
\texttt{canonical(X) :- nonvar(X),functor(X,s,1).}
\texttt{canonical(X) :- X == 0.}
\texttt{eager <= left(eager),}
\texttt{a\_right(\_,eager),}
\texttt{axiom(\_,\_,\_).}
\texttt{lazy <= axiom(\_,\_,\_),}
\texttt{a\_right(\_,lazy),}
\texttt{left(lazy).}

\textbf{Queries:}

1) Add 1 and 1 eagerly:

\texttt{\_ ?- eager \\- add(s(0),s(0)) \- P.}
\texttt{P = s(s(0)) ? ;}
\texttt{P = s(add(0,s(0))) ? ;}
\texttt{P = succ(add(0,s(0))) ? ;}
\texttt{P = add(s(0),s(0)) ? ;}
\texttt{no}
\texttt{\_ ?-}

2) Add 1 and 1 lazy:

\texttt{\_ ?- lazy \_\_ \- add(s(0),s(0)) \_\_ P.}
\texttt{P = add(s(0),s(0)) ? ;}
\texttt{P = succ(add(0,s(0))) ? ;}

P = s(add(0,s(0))) ;

P = s(s(0)) ;

no

5.4 Mixing Relational and Functional Programming

This example shows how relational and functional programming can be mixed. qsort/1 and append/2 are functions while split/4 is a relation. The example also shows how the underlying Prolog system can be used to implement for example relations on numbers by the two primitives < and >.

Definition:

qsort([], []) .
qsort([F|R]) <-
    pi L \ (pi G \ (split(F,R,L,G) ->
        append(qsort(L),cons(F,qsort(G))))).

append([], F) <= F.
append([F|R], X) <= cons(F, append(R, X)).
append(X, Y) #\{X \notin [_|_], X \notin []\} <=
    pi Z \ ((X -> Z) -> append(Z,Y)).

split(_, [], [], []).
split(E,[F|R],[F|Z], X <= E => F,split(E,R,Z,X).
split(E,[F|R],Z,[F|X]) <= E < F,split(E,R,Z,X).

cons(X,Y) <= pi Z \ (pi W \ ((X -> Z), (Y -> W) -> [Z|W])).

Rules and strategies:

:- include_rules(lib('rules.rul')).

%%% Rules

right_l <=
    X <= Y ->
    (_ \-- X < Y).
right_g_e <=

Chapter 5: Examples

29
\[ X \geq Y \rightarrow \\
(\_ \setminus X \geq Y). \]

%%%% Restrictions of rules defined in rules.rul
q_axiom(T,I) <=
    (canonical(T) ->
     (IE[\_\_] \setminus \_)).
q_axiom(T,I) <=
    axiom(T,\_,I).
q_d_left(T,I,PT) <=
    (not(canonical(T)) ->
     (IE[\_\_] \setminus \_)).
q_d_left(T,I,PT) <=
    d_left(T,I,PT).

%%%% Provisos
canonical(X) :- var(X).
canonical(X) :- X == \_.
canonical(X) :- functor(X,'.',2).
canonical(X) :- number(X).

%%%% Strategies
%%%% Top level strategy
qs <= q_fun,
    \_q_rel. % Functional execution
    q_rel. % Relational execution

qs1 <= d_right(\_,q_rel),
a_right(\_,q_fun),
v_right(\_,qs1,qs1).

q_fun <=
a_right(\_,q_fun),
a_left(\_,qs1,q_fun),
pi_left(\_,q_fun),
q_d_left(\_,q_fun),
q_axiom(\_,\_).

q_rel <=
    right_q_e,
    right_1,
    true_right,
a_right(\_,qs),
v_right(\_,q_rel,q_rel),
d_right(\_,q_rel).

Queries:
1) Sort the list [4,2,1,3,9,7,8,6]:

   | ?- qs \qsort([4,2,1,3,9,7,8,6]) \- P.
   P = [1,2,3,4,6,7,8,9] ? ;

   no
   | ?-
Chapter 6: Statistic package

6 Statistic package

The statistic package of GCLA is a tool for gathering information about the execution of a query/queries. It is based on the debugger, and thus are all invocation numbers and the like the same as in the debugger. The package is loaded by posing the directive

\[ ?- gcla_load_statpack. \]

to the GCLA top level.

The package defines a new meta level deduction symbol, //-. Whenever a query \texttt{strat \texttt{//- Antecedent} \texttt{- Consequent}} is posed to the system, the package is invoked, the counters reset and the collecting of information is started. Backtracking and presentation of answer substitutions are the same as usual. The statistic package cannot be combined with the debugger, i.e. one cannot debug a query and collect statistic information about the same query at the same time.

6.1 Getting Information

There are 6 different information-retrieving directives.

\texttt{get_calls}

Gives the number of times each rule and strategy is called, c.f. \texttt{call} in the debugger.

\texttt{get_exits}

Gives the number of times each rule and strategy succeeds, c.f. \texttt{exit} in the debugger.

\texttt{get_retries}

Gives the number of times each rule and strategy is retried, c.f. \texttt{redo} in the debugger.

\texttt{get_failures}

Gives the number of times each rule and strategy fails, c.f. \texttt{fail} in the debugger.

\texttt{get_not_used}

Gives the rules and strategies that are called but never succeeds.

\texttt{compare_paths}

If several answers have been generated, \texttt{compare_paths} tries to find where the proofs differ for the first time. The format of the answer is
3 4 Rule \\ Antecedent \- Consequent \\
5 4 Rule' \\ Antecedent' \- Consequent'

where two immediate rows are the result of comparing the two immediate proofs. The first number (3 and 5) are the same unique invocation number as in the debugger, and 4 is the same depth number as in the debugger.

The format could also be in one of the following formats:

- When the proviso clause is nondeterministic, it is shown in the print-out like this:

3 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: clause(Atom1,Condition1)
5 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: clause(Atom2,Condition2)

- When the proviso definiens is nondeterministic, it is shown in the print-out like this:

3 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: definiens(Atom1,Condition1,Num1)
5 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: definiens(Atom2,Condition2,Num2)

- When the append operator \( \oplus \) is indeterministic, it is shown in the print-out like this:

3 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: List_A1@List_B1
5 4 Rule \\ Antecedent \- Consequent \\
\ PROVISO: List_A2@List_B2

In the following, 'answer number N' stands for the path, i.e. sequence of rule and strategy applications, associated with the returned answer number N. Answer number 1 is the answer that the interpreter returns first. The answers generated by backtracking are numbered increasingly by 1.

path path lists the sequence of rule- and strategy applications associated with the first answer returned to the query.

path(N) path(N) lists the sequence of rule- and strategy applications associated with the answer number N.

applications(R/A) applications(R/A) lists the goals, which are applications of the rule or strategy R with arity A, for the first answer returned.

applications(R/A,N) applications(R/A,N) lists the goals, which are applications of the rule or strategy R with arity A, for the first answer returned, for answer number N.
following_applications(R/A,N)

following_applications(R/A,N) lists the goals where the rule or strategy R with arity A is used, followed by N other rule- and strategy applications, for returned answer number 1.

following_applications(R/A,PN,N)

following_applications(R/A,PN,N) lists the goals where the rule or strategy R with arity A is used, followed by N other rule/strategy applications, for returned answer number PN.
7 Formal Syntax

This section is based on the syntax described in [Kre91]. Some minor changes have been made, due to the current implementation.

The operators defined by the GCLA system are:

op(900, fy, gcla_nospy)
op(900, fy, gcla_spy)
op(700, xfx, ('\='))
op(1150, xfx, ('\-\-'))
op(1150, fx, ('\-'))
op(1175, xfx, ('\\-\-'))
op(1175, fx, ('\\-\-'))
op(950, yfx, ('\<-'))
op(1200, xfy, ('\<='))
op(900, xfy, (@))
op(900, xfy, (#))
op(1100, xfy, \)
op(1100, fy, pi)
op(850, xfy, i)
op(1175, xfx, '/\-\-')
op(1175, fx, '/\-\-')

The syntax of a definition is given as:

Definition ::= "Empty Definition" | DefClause. Definition

DefClause ::= Atom <= Condition | Atom#Guard <= Condition | Atom | Atom#Guard

Atom ::= tc if tc/0 is a TermConstr |

tc(Term1, ..., Termn) if n > 0 and tc/n is a TermConstr

TermConstr ::= "Non-capitalized alphanumeric string that is not a CondConstr"/n for some n >= 0
Term ::= Atom | Var | List
List ::= [] | [Term, ..., Term] | [Term|List]
Var ::= "Capitalized alphanumeric string" | _ | "Alphanumeric string beginning with _"
Condition ::= Term | CCond
CCond ::= pi Var \ Condition | cc if cc/0 is a CondConstr |
cc(Condition1, ..., Conditionn) if n > 0 and cc/n is a CondConstr (possibly infix)
CondConstr ::= ...
Guard ::= {Var \= Term, ..., Var \= Term} | else

Proof terms are representations of proofs (or set of proofs). They can be seen as functional expressions that compute sequents from proofs with respect to a certain rule definition.

The syntax of a proof term is:

ProofTerm ::= InfBody
SeqTerm ::= (AntTerm \- CondTerm)
AntTerm ::= ProofVar | [] | [CondTerm, ..., CondTerm] | [CondTerm | AntTerm] | AntTerm@AntTerm
ProofVar ::= "Capitalized alphanumeric string" | _ | Alphanumeric string beginning with _
ProofAtom ::= pc(PAtomArg1, ..., PAtomArgn) if pc/n is a ProofConstr
PAtomArg ::= AntTerm | ProofVar | ProofAtom | CondTerm
ProofConstr ::= "Noncapitalized alphanumeric string that is not ProvConstr"/n, for some n >= 0.
CondTerm ::= ObjTerm | CCond
CondConstr ::= 
given by the table constructor(cc, n). The default table contains ./2, ;/2, pi/1, ~/2, 
true/0, false/0, ->/2, add_def/2, rem_def/2.

ObjTerm ::= 
Term (see BNF for definitions)

Inference rules and search strategies are given by the BNF below.

RuleDef ::= 
"Empty Definition" | InfDef. RuleDef | StratDef. RuleDef

InfDef ::= ProofAtom <= InfBody | ProofAtom#Guard <= InfBody

StratDef ::= 
ProofAtom <= InfBody | ProofAtom#Guard <= InfBody

InfBody ::= 
pi ProofVar \ InfBody | SeqTerm | (ProvOrRule -> InfBody) | (InfBody , InfBody) | (InfBody ; InfBody)

ProvOrRule ::= 
(InfBody -> SeqTerm) | Proviso | (ProvOrRule , ProvOrRule) | (ProvOrRule 
; ProvOrRule) | not(ProvOrRule) | ((Template i ProvOrRule) -> ProofVar) |
(i(Constr, Condition, Condition, ProvOrRule) -> ProofVar)

Template ::= 
[Condition] | , (Condition)

The syntax of provisos are given by the following BNF:

ProvDef ::= 
"Empty proviso definition" | ProClause. ProvDef

ProClause ::= 
ProvAtom :- ProvBody | ProvAtom | ProvAtom#Guard :- ProvBody |
ProvAtom#Guard

ProvAtom ::= 
pc if pc/0 is a ProvConstr | pc(ProvTerm1, . . . , ProvTermn) if n >= 1 and pc/n is 
a ProvConstr
\textbf{ProvConstr ::=}

"Noncapitalized alphanumeric string"/n, for some n \geq 0

\textbf{ProvTerm ::=}

\textbf{CondTerm} \mid \textbf{ProofVar}

\textbf{ProvBody ::=}

\textbf{ProvOrRule}

\textbf{Proviso ::=}

\textbf{ProvAtom} \mid \textbf{true} \mid \textbf{false} \mid \textbf{Proviso} \textbf{, Proviso} \mid \textbf{Proviso} \textbf{; Proviso} \mid \pi \textbf{ProofVar} \setminus \textbf{Proviso}

Queries are described by the following BNF:

\textbf{Query ::=} \textbf{ProofTerm} \setminus \textbf{Sequent} \mid (\textbf{ProofTerm} \setminus \textbf{Sequent}) \textbf{, Query}

\textbf{Sequent ::=}

\textbf{Antecedent} \setminus \textbf{Consequent}

\textbf{Antecedent ::=}

\textbf{CondTerm} \mid \textbf{CondTerm} \textbf{, Antecedent}

\textbf{Consequent ::=}

\textbf{CondTerm}
References


