ISRN SICS-R--92/05--SE

Methodology and Programming Techniques in GCLA II

by
Martin Aronsson

SICS research report
R92:05
ISSN 0283-3638

Methodology and Programming Techniques in GCLA II

Martin Aronsson
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden
email: martin@sics.se

1992-03-12

Abstract
We will demonstrate various implementation techniques in the language GCLA. First an introduction to GCLA is given, followed by some examples of program developments, to demonstrate the development methodology. Other examples are also given to show various implementation techniques and properties of the system.

1. Introduction 1
2. GCLAII 1
2.1 The Definition 2
2.2 The Rule Definition 3
2.2.1 The inference rules 3
2.2.2 The search strategies 8
3. Developing programs in the GCLA system 9
3.1 A simple example: Default reasoning 9
3.1.1 Step 1: Writing the declarative part 9
3.1.2 Step 2: Writing strategies 11
3.2 Another example: Functional programming 14
3.2.1 Step 1: The declarative part 14
3.2.2 Step 2: Writing strategies 15
3.2.3 Step 3: Writing specialized rules 16
4. Other examples 17
4.1 Sorting 17
4.2 STRIPS like planning 21
4.3 Object oriented programming 24
4.4 Guiding the search among the assumptions 29
5. Conclusion 33
References 33
Appendix A i
Appendix B iii
Appendix C vii

1. Introduction

The programming system GCLA has been developed for some years at SICS. It is a logical programming language, and has similar syntax as Prolog, while the declarative semantics is completely different. While Prolog is based on first order logic, GCLA is based on *Partial inductive definition* (PID), a framework developed by Lars Hallnäs [Hal91, HS-H88].

During the time there have been several versions. Two main versions can be discerned. One older version interpreting a GCLA program, having some inference rules given beforehand, called GCLA I [Aro90]. GCLA I had a very restricted set of control primitives, which led to a large search space for larger programs. From the GCLA I system the GCLA II system was developed. GCLA II generalizes GCLA I in the sense that the inference rules that interpreted the GCLA I program can be defined by the user. The search order among the inference rules can also be freely defined by the user. By this generalization GCLA II consists of two parts; the GCLA I program, which we hereafter will refer to as the *definition* or the *object level*, and the code which implements the inference rules and search strategies which we will call the *rule definition* or the *meta level*. The rule definition is a restricted form of a GCLA I program with some primitives for accessing the definition, so the two parts share the same theoretical basis. For a more complete presentation of GCLA II's theoretical properties and its relation to PID see [Kre92].

The definition is intended to define the declarative knowledge of a domain while the rule definition is intended to define how the declarative knowledge is to be used. The development methodology we think of is a stepwise refinement scheme: first the programmer writes a declarative program, the definition, and starts with a set of general search strategies and inference rules. This general set implements the behaviour of GCLA I. Then, as the programmer gains more experience of *how* the declarative knowledge is to be utilized, other search strategies and restrictions on rules are implemented in the rule definition. Specialized rules can be implemented, and ultimately the "declarative content" of the definition has been efficiently implemented by the definition and a set of specialized rules and search strategies, performing the inferences that one wants to perform, and nothing more. This development procedure gives as a result that the declarative program has been proceduralized, without changing the declaraitve part, and the same definition can be used by several different sets of rules and strategies, depending on what one wants to achieve.

GCLA should not be seen as a programming language for a final implementation, but as a programming environment, where the programmer has a lot of freedom to test different ideas and techniques. When the GCLA programmer is finished, the result is a specification of the behaviour and declarative content the application should have.

We will in this text refer to GCLA II as GCLA, or the GCLA system.

We will give a short presentation of GCLA II, then give a small programming example to show the programming methodology, and then give some further examples to show different programming techniques.

2. GCLAII

The GCLA system is divided into two parts, one declarative part, called the *definition* or the *object level*, and one procedural part, called the *rule definition* or the *meta level*. The

procedural part performs inferences and draws conclusions from the declarative part, but the procedural part is not a meta interpreter, even though it has the same theoretical basis as the declarative part. The rule definition is a subset of the language used at the declarative level, together with some predefined primitives that acts as an interface between the two.

Since the two levels are separated, the symbols and variables are also separated. This means that the variables are of different kinds. For example, a meta level variable can be bound to an object level variable but not the other way around, and an object level variable cannot be bound to a meta level structure, just to object level terms. Object level variables are treated as constants at the meta level.

For a more comprehensive description of GCLAII and its theoretical properties the reader is referred to [Kre92].

The presentation here will focus on the syntax and properties that our prototype implementation has, therefore some differences to [Kre92] can occur.

2.1 The Definition

The definition contains the declarative knowledge of the domain. In GCLA I this part was called the program.

The syntax is similar to Prolog. Since we talk about inductive definitions, we have no predicates or functions, just terms and conditions. A *constant* is a *term*, as well as a *variable*. Constants begin with a lowercase letter, while variables begin with an uppercase letter, or "_". The single symbol "_" denotes an anonymous variable. If A_1, ..., A_n are terms and f is a functor (object level constructor) of arity n, then $f(A_1, ..., A_n)$ is a *term*. All terms are *conditions*, and if C_1 and C_2 are conditions, then so are C_1 -> C_2, (C_1, C_2), $(C_1; C_2)$, `true` and `false`. If X is an object level variable and C a condition, then `pi` $X\ C$ is a *condition*.

The conditions that are not terms are symbols and structures that have corresponding inference rules in the procedural part. They cannot be bound to an (object level) variable, and are in fact meta level structures. They are in some papers refered to as *lifting operators*, since they are reachable from both levels. Such conditions are C_1 -> C_2, (C_1, C_2), $(C_1; C_2)$, `pi X\ C`, `true` and `false`. Other conditions of this kind can be defined by the user.

An *atom* is a term which is not a variable. If A is an atom, C a condition, then A <= C is a *clause*. An ordered set of clauses forms a *definition* $\mathcal{D}$.

Compared to pure Prolog (we discard all such things as `var`, `!`, etc), what has been added is the possibility to assume conditions. For example, the clause

```
a <= (b -> c)
```

should be read as "`a` holds if `c` can be proved while assuming `b`".

There is also a richer set of queries in GCLA than in Prolog. An ordinary Prolog query is written

```
\- a.
```

and should be read "Does a hold (in the definition $\mathcal{D}$)". One can also assume things in the query, for example

```
c \- a.
```

which should be read as "Assuming c, does a hold (in the definition $\mathcal{D}$)", or "Is a derivable from c".

An example of a definition is this small toy expert system.

```
symptom(high_temp) <= disease(pneumonia).
symptom(high_temp) <= disease(plague).
symptom(cough) <= disease(pneumonia).
symptom(cough) <= disease(cold).
```

The definition contains the rules connecting symptoms and diseases, but contains no facts. The facts are submitted by the queries. The intended answer to the query

```
disease(X) \- symptom(high_temp).
```

is that the variable X should be bound to pneumonia, and on backtracking to plague.

The query

```
symptom(high_temp) \- (disease(X), disease(Y)).
```

should give as result two possible bindings:

```
X = pneumonia, Y = plague     or
X = plague, Y = pneumonia
```

The definition should not be seen as a set of logical clauses, but as a set of clauses defining a set of terms and conditions that can be generated by the definition. So we need some rules for how this set of term and conditions should be generated from the definition. These inference rules are part of the rule definition.

2.2 The Rule Definition

The rule definition contains the procedural knowledge of the domain, i.e. the knowledge used for drawing conclusions based on the declarative knowledge in the definition. This part implements both inference rules and search strategies among the rules and among the assumptions in the object level sequents. When the GCLA system is started, the user is furnished with a general set of inference rules and strategies, implementing the behaviour of GCLA I. This startup set can be extended with other rules and/or strategies if the user wants that, or even be discarded. It is all up to the user.

2.2.1 The inference rules

The underlying idea of the rule definition is that the inference rules are coded as functions, from the premises of a rule to the conclusion, with a possible proviso. The inference rule

$$\frac{P_1, \ldots, P_n}{C} \; rule\_name \qquad Proviso$$

is coded by the function

$$rule\_name(P_1, ..., P_n) = (Proviso, P_1, ..., P_n) \rightarrow C$$

where P_i and C are object level sequents, and *Proviso* contains restrictions when the rule is applicable. The arrow "->" should be read as "if ... then ...". So these rules are conditional functions from sequents to sequents.

Each P_i can be obtained by it's derivation leading to P_i. A representation of the derivation is the nested functional expression of rulenames leading to P_i, so P_i can be replaced by this expression. In GCLA such a term is called a *proofterm*, and sequents are normal forms of profterms. So evaluating a proofterm $r_1(r_2(...), ...)$ gives as a result a sequent *Assumptions* $\vdash$ *Conclusion*.

A simple example is

```
axiom(Term,Term)
```

which evaluates to

Term $\vdash$ Term.

So, testing a GCLA query if it holds or not results in the system trying to find a proofterm whose canonical form is the query (the object level sequent) and an (object level) answer substitution.

An inference rule is coded in GCLA as

```
rule_name(PT1, ..., PTn) <=
      (Proviso, (PT1 -> Seq1), ..., (PTn -> Seqn)) -> Seq
```

where `Seq`, Seq_i is object level sequents. The rule should be read: If `Proviso` holds, and if PT_i maps to Seq_i, then `rule_name(PT1, ..., PTn)` maps to `Seq`. The proviso contains restrictions when the rule is applicable, together with some other primitives. These primitives perform tests of various kinds, and are also the interface to the object level definition. Some of the primitives are:

`atom(T)`	Checks whether the term `T` is an object level term, but not an object level variable
`term(T)`	Checks whether the term `T` is an object level term
`clause(T,B)`	Succeeds if `T <= B` is an object level clause in the current definition. The meta level variable `B` can be instantiated by the call. If `T` is undefined and `B` is an uninstantiated (meta level) variable, `B` is bound to `false`. This primitive does not check whether T is an atom or not.
`definiens(T,B)`	Succeeds if the definiens of the term `T` is `B` (`B` is a ';'-separated structure of the bodies in the definiens). In short, the definiens of a term `T` is the (maximal) set of all bodies whose heads are unified with each other, and unified with `T`. There can be more than one definiens for a term `T`. This primitive does not check whether T is an atom or not. The reader is refered to [Kre92] for a formal definition of the definiens operation
`unify(T1,T2)`	Unifies the object level terms `T1` and `T2`.

`not(T)`	`not` is true if `T` is proven false, i.e. `not` could be thought of as having the GCLA definition `not(T) <= T -> false.` Negation in GCLA will be discussed in section 3.1.1.
`inst(X,C,C1)`	Instantiates `X` to a new variable in `C`, resulting in `C1`. This primitive is used in Π-quantified conditions, see below.

The primitives `clause`, `definiens` and `unify` are the only ways to unify object level terms. There are a number of other primitives, which will be introduced when they are called for.

The user has also the opportunity to write his own provisos into a *proviso definiton.* This proviso definition is part of the procedural part. To distinguish the proviso definition from the inference rules, the proviso clauses are written as '*Head :- Body*' (i.e. as in most Prologs). In the current implementation the provisos are in principle compiled to Prolog clauses which are directly executed by the underlying Prolog system.

A GCLA II query consists of two derivation symbols, one object level symbol '\-' and one meta level symbol '\\-':

rule_name(*PT*$_1$, ..., *PT*$_n$) \\- (*Assumption* \- *Conclusion*).

GCLA tries to fill in *PT*$_i$ with proofterms coding the derivation tree corresponding to `rule_name`'s premises.

As an example we give the inference rules implementing GCLA I, or, in fact, the rules for the calculus $OL\mathcal{D}$ in [Kre92], and the corresponding code in GCLA II. The operator "`@`" is an infix (right associative) append operator that appends it's two arguments, "`|`" is an infix cons operator and "`[`" and "`]`" marks a list of assumptions. To improve readability, we have omitted the things making $OL\mathcal{D}$ a linear calculus.

If one reads the premises of the rules from left to right, one gets the linear calculus $OL\mathcal{D}$.

$\dfrac{A\sigma \vdash B\sigma}{A \vdash c}\vdash_{\mathcal{D}}$
where $b \Leftarrow B. \in \mathcal{D}$,
and $\sigma = mgu(b,c)$
	`d_right(C,PT) <=`
	`  atom(C),`
	`  clause(C,B),`
	`  (PT -> (A \- B))`
	`  -> (A \- C).`

$\dfrac{I\sigma, D\sigma, R\sigma \vdash C\sigma}{I, a, R \vdash C}\mathcal{D}\vdash$
where σ is an a-sufficient substitution with respect to $\mathcal{D}$ and D is calculated by the definiens operation.
	`d_left(A,I,PT) <=`
	`  atom(A),`
	`  definiens(A,D),`
	`  (PT -> (I@[D|R] \- C))`
	`  -> (I@[A|R] \- C).`

$\dfrac{A_1, A \vdash C}{A \vdash A_1 \rightarrow C}\vdash\rightarrow$	`a_right((E -> C),PT) <=`
	`  (PT -> ([A1|A] \- C))`
	`  -> (A \- (A1 -> C)).`

$\dfrac{(I, R \vdash B)\ (I, C_1, R \vdash C)}{I, (B \rightarrow C_1), R \vdash C} \rightarrow\vdash$	`a_left((B -> C1),I,PT1,PT2) <=`
	`  (PT1 -> (I@R \- B)),`
	`  (PT2 -> (I@[C1|R] \- C))`
	`  -> (I@[(B -> C1)|R] \- C).`

$\dfrac{}{I, a, R \vdash c}$ *Initial sequent*
and $\sigma = mgu(a,c)$
(This rule is also refered to as *axiom*)
	`axiom(A,C,I) <=`
	`  term(C),`
	`  term(A),`
	`  unify(C,A)`
	`  -> (I@[A|R] \- C).`

$\dfrac{}{A \vdash \top} \vdash \top$	`true_right <= (A \- true).`
$\dfrac{}{I, \bot, R - C} \bot\vdash$	`false_left(I) <=` `  (I@[false|R] \- C).`

$\dfrac{(A \vdash C_1)\quad (A \vdash C_2)}{A \vdash (C_1, C_2)} \vdash,$	`v_right((C1,C2),PT1,PT2) <=`
	`  (PT1 -> (A \- C1)),`
	`  (PT2 -> (A \- C2))`
	`  -> (A \- (C1, C2)).`

$\dfrac{I, C_1, C_2, R \vdash C}{I, (C_1, C_2), R \vdash C} ,\vdash$	`v_left((C1,C2),I,PT) <=`
	`  (PT -> (I@[C1, C2|R] \- C))`
	`  -> (I@[(C1, C2)|R] \- C).`

$\dfrac{A \vdash C_i}{A \vdash (C_1 ; C_2)} \vdash;$	`o_right(1,(C1;C2),PT) <=`
	`  (PT -> (A \- C1))`
	`  -> (A \- (C1; C2)).`
	`o_right(2,(C1 ; C2),PT) <=`
	`  (PT -> (A \- C2))`
	`  -> (A \- (C1; C2)).`

$\dfrac{(I, C_1, R \vdash C)\ (I, C_2, R \vdash C)}{I, (C_1; C_2), R \vdash C} ;\vdash$	`o_left((C1;C2),I,PT1, PT2) <=`
	`  (PT1 -> (I@[C1|R] \- C)),`
	`  (PT2 -> (I@[C2|R] \- C))`
	`  -> (I@[(C1; C2)|R] \- C).`

$\dfrac{I,\ C_2,\ R \vdash C}{I,\ (\Pi x.C_1),\ R \vdash C}\ \Pi\vdash$
 	
where C_2 is C_1 where all occurrences of X has been replaced by a term
	`pi_left((pi X\ C1),I,PT) <=`
	`  inst(X,C1,C2),`
	`  (PT -> (I@[C2|R] \- C))`
	`  -> (I@[(pi X\ C1)|R] \- C).`

Note that in `pi_left` if we are certain that `X` does not occur somewhere else than in `C1`, the `inst` proviso can be removed. This is the case in the current implementation, where all occurences of `X` in `C1` is replaced by a new variable at compile time.

When a clause introduce new variables in the body, these variables are thought of as existentially quantified (c.f. Prolog). Therefore there is an implicit existentially quantifer for all such variables. This means that there is an implicit, trivial rule for handling existential quantified variables to the right of '$\vdash$' ($\vdash \Sigma$). This rule is analogous to existential introduction in natural deduction. To the left existential variables are much more complicated to handle, since a rule for these variables should correspond to existential elimination. Therefore, bodies that introduce new variables should not be reduced by any left hand side rule.

However, there are cases where one wants to introduce variables that are thought of as universally quantified, and these variables are easy to handle to the left of the turnstile but not to the right (the dual behaviour to existentially quantified variables). To introduce such variables in the body of a clause, the Π (`pi`) symbol is used, which is taken care of by the rule `pi_left`. Note that there exist no `pi_right` rule, since the complexity for that rule is to high (c.f. existentially quantified variables to the left). For further information about variables and their interpretation in GCLA see [Eri92].

The Π-rule rule was not part of the GCLA I system, but is useful, for example the functional programming presented in section 3.2 uses this construct. It should also be noted that the system does not check for new variables introduced in the body, so the programmer must be aware of this condition. If one wants to introduce a rule for handling existentially quantified variables, it i easy to do that. In principle the rule looks like the `pi_left rule`, but operates on the object level conclusion instead.

Below is a generic derivation (in tree format) of a single step of a meta level inference step. For a description of this in linear form, and further descriptions about the calculus used at the meta level, we refer to [Kre92]. It should be pointed out that for execution efficiency in the second derivation step the meta level sequent to the right (*Seq' \\- Seq*) is executed before the left one.

```
                              ...                     ...
       ...               ---------------        ---------------
                         PT1 \\- Seq1           PTn \\- Seqn
 ------------------    ------------------     ------------------
 \\- Provisos          \\- PT1 -> Seq1   ...  \\- PTn -> Seqn
 ---------------------------------------------------------------   ---------------
 \\- Provisos, (PT1 -> Seq1), ..., (PTn -> Seqn)                    Seq' \\- Seq
 ---------------------------------------------------------------------------------
          (Provisos, (PT1 -> Seq1), ..., (PTn -> Seqn)) -> Seq' \\- Seq
          -----------------------------------------------------------------
                          rule(PT1, ..., PTn) \\- Seq
```

Sofar we have coded inference rules. We are now going to introduce search strategies into the procedural part.

2.2.2 The search strategies

We have thought of the proofterms as uninstantiated, or instantiated to a rule term. An uninstantiated proofterm was thought of as representing all possible derivation that was possible to perform. When search strategies are introduced, they will be in place of the uninstantiated proofterm, and refer to a particular search strategy instead. So, the proofterms are now always instantiated to a rule term, or a strategy term. The strategies can be seen as indeterministic functions, representing several possible derivations.

A strategy does always contain a vector of proofterms. The comma is to be interpreted disjunctively in this vector, because the vector will occur to the left of the meta level derivation symbol '\\-'. This vector is tried from left to right. It is this vector that implements the indeterminism of a strategy.

The simplest form of a search strategy is

```
strat <= PT1, ..., PTn
```

where each PT_i is a proofterm. PT_1 is tried first and PT_n is tried last. `strat` can also contain one or more arguments. An example of a strategy is

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

where `arl` stands for "axiom, right, left". `axiom` is a rule name while `left` and `right` are strategy names.

Strategies can also contain restrictions on their applicability. These are written

```
strat <= Restriction -> Seq.
strat <= name1, ..., namen.
```

If the first clause holds, the vector in the second clause is used. Since proofterms occur to the left of the derivation symbol '\\-', both clauses should be used conjunctively in the proof. This contrasts to the usual usage where the clauses are read disjunctively (cf Prolog).

Below is a generic strategy derivation. The execution continues in the right branch. As before, when splitting a meta level arrow (->) to the left (of the turnstile) we first perform the meta level axiom rule on the right branch (i.e. on the meta level sequent `Seq' \\- Seq`) before executing the *Provisos*.

$$\dfrac{\dfrac{\dfrac{\dots}{\backslash\backslash\text{-}\ Provisos} \quad \dfrac{}{Seq'\ \backslash\backslash\text{-}\ Seq}}{Provisos \rightarrow Seq'\ \backslash\backslash\text{-}\ Seq} \quad \dfrac{\dfrac{\dots}{strat_i\ \backslash\backslash\text{-}\ Seq}}{(strat_1, \dots, strat_n)\ \backslash\backslash\text{-}\ Seq}}{(Provisos \rightarrow Seq');(strat_1, \dots, strat_n)\ \backslash\backslash\text{-}\ Seq}}{strat\ \backslash\backslash\text{-}\ Seq}$$

The code below implements some common, general search strategies:

```
gcla <= arl.
arl <= axiom(_,_,_), right(arl), left(arl).
alr <= axiom(_,_,_), left(alr), right(alr).
lra <= left(lra), right(lra), axiom(_,_,_).
right(PT) <=
      true_right, v_right(C,PT,PT), a_right(C,PT),
      o_right(N,C,PT), d_right(A,PT).
left(PT) <=
      false_left(I), v_left(C,I,PT), a_left(C,I,PT,PT),
      o_left(C,I,PT,PT), d_left(A,I,PT), pi_left(C,I,PT).
```

For a complete listing of the code which implements the general rules and strategies in GCLA II see appendix A.

For a more thorough treatment of GCLA and its properties we refer to [Kre92].

One should also note that even though the procedural part codes algorithms, which define how the declarative part is to be utilized, the coding of the rules and strategies is a subset of GCLA I and can thus be given a declarative reading.

We are now equipped with a sufficient framework to look at an example program.

3. Developing programs in the GCLA system

One of the main objectives for developing GCLA was to make a system that supports prototyping development of programs. The methodology in GCLA II is to first (try to) write down the declarative knowledge. Then the programmer starts with general inference rules for making derivations from the declarative knowledge. These rules are supported by the system. As the programmer gets more familiar with the domain, the general inference rules can be replaced or extended by more efficient algorithms, implementing special inference rules for this particular domain and application. Derivations that do not contribute or are even thought of as wrong can be cut off. All this can be done without affecting the declarative part. There can also be several different procedural parts for different applications, while the declarative part is shared between the different applications.

3.1 A simple example: Default reasoning

We will use a small example implementing default reasoning for demonstrating the development technique in GCLA.

The declarative content of the program is

> An object is grey if it is an elephant and not an albino elephant
> Clyde and Dumbo are elephants, and Jumbo is an albino elephant
> All albino elephants are elephants.

This program can be categorized as a default reasoning program.

3.1.1 Step 1: Writing the declarative part

We start out by writing the declarative part. In this case it is simple, we just transform the clauses above to GCLA clauses:

```
grey(X) <= elephant(X), (albino_elephant(X) -> false).

elephant(clyde).
elephant(dumbo).
elephant(X) <= albino_elephant(X).

albino_elephant(jumbo).
```

The last condition in the first clause shows how negation is accomplished in GCLA. `false` could be any symbol that is not defined, but it is convenient to decide upon one symbol as the false (or absurd) symbol. By assuming `albino_elephant(X)` and trying to prove the `false` symbol, we accomplish negation. This form of negation behaves as negation as failure when it occurs to the right of the derivation symbol "`\-`" (i.e. positively). When it occurs to the left of the derivation symbol (i.e. negatively), `albino_elephant(X)` is put to the right of "`\-`" in a new sequent according to the rule `a_left`, and thus we can get bindings to the variable `X`. It is easy to see that double negation is stripped off, i.e. `\- ((p -> false) -> false)` is reduced to `\- p`.

Example queries are

```
\- grey(P)
```

which binds `P` to `clyde` or `dumbo`, and

```
grey(P) \- false
```

which binds `P` to `jumbo`. The last query succeeds 9 times with the set of general rules presented in section 2.2, i.e. there exists 9 different derivations of `grey(P) \- false`. We are going to use this last query as an example query throughout this section, so when we give statistical values in this section we are refering to this query.

By using the statistical package of GCLA we can get different statistical values for a query. Below is some values for the 9 derivations of the query `arl \\- (grey(P) \- false)`:

a_left/4	succeeds 1 times	a_left/4	fails 43 times
a_right/2	succeeds 0 times	a_right/2	fails 44 times
arl/0	succeeds 18 times	arl/0	fails 28 times
axiom/3	succeeds 0 times	axiom/3	fails 44 times
d_left/3	succeeds 9 times	d_left/3	fails 35 times
d_right/2	succeeds 1 times	d_right/2	fails 43 times
false_left/1	succeeds 9 times	false_left/1	fails 35 times
left/1	succeeds 20 times	left/1	fails 30 times
o_left/4	succeeds 0 times	o_left/4	fails 44 times
o_right/3	succeeds 0 times	o_right/3	fails 44 times
pi_left/3	succeeds 0 times	pi_left/3	fails 44 times
right/1	succeeds 4 times	right/1	fails 40 times
true_right/0	succeeds 3 times	true_right/0	fails 41 times
v_left/3	succeeds 1 times	v_left/3	fails 43 times
v_right/3	succeeds 0 times	v_right/3	fails 44 times
TOTAL: 66		TOTAL: 602	

Note that a call to a rule or strategy can both succeed and fail. This happens for example when a rule application succeeds, but the rest of the execution fails, or we force the system to backtrack and find another solution.

We can also get statistics about choicepoints. Below is a listing of where the positive choicepoints (i.e. those choices that results in an answer substitution) are for the query `arl \\- (grey(P) \- false)`. The first position is an unique invokation identifier, i.e. the number of the call. For example, the first choicepoint occurs after 46 rule- and strategy calls. The next number is the depth of the call. The third position is the rule or strategy where the choicepoint arises, and the fourth position is the rule choosen. The last position is the sequent where the choice arises. The first two positions are the same as in the tracer, which can be useful to debug programs (a listing of a trace of the call `arl \\- (grey(P) \- false)` is given in appendix B).

```
These are the choicepoints:
46  12   arl              right(arl)     elephant(jumbo)\-true
46  12   arl              left(arl)      elephant(jumbo)\-true
58  10   left(arl)        false_left([elephant(jumbo)])
                              elephant(jumbo),false\-false
58  10   left(arl)        d_left(elephant(jumbo),[],arl)
                              elephant(jumbo),false\-false
72  13   left(arl)        false_left([albino_elephant(jumbo)])
                              albino_elephant(jumbo),false\-false
72  13   left(arl)        d_left(albino_elephant(jumbo),[],arl)
                              albino_elephant(jumbo),false\-false
105 15   arl              right(arl)     albino_elephant(jumbo)\-true
105 15   arl              left(arl)      albino_elephant(jumbo)\-true
117 10   left(arl)        false_left([elephant(jumbo)])
                              elephant(jumbo),false\-false
117 10   left(arl)        d_left(elephant(jumbo),[],arl)
                              elephant(jumbo),false\-false
131 13   left(arl)        false_left([albino_elephant(jumbo)])
                              albino_elephant(jumbo),false\-false
131 13   left(arl)        d_left(albino_elephant(jumbo),[],arl)
                              albino_elephant(jumbo),false\-false
176 10   left(arl)        false_left([elephant(jumbo)])
                              elephant(jumbo),false\-false
176 10   left(arl)        d_left(elephant(jumbo),[],arl)
                              elephant(jumbo),false\-false
190 13   left(arl)        false_left([albino_elephant(jumbo)])
                              albino_elephant(jumbo),false\-false
190 13   left(arl)        d_left(albino_elephant(jumbo),[],arl)
                              albino_elephant(jumbo),false\-false
```

As we can see, there are 8 choicepoints. We will now use this knowledge to write more efficient meta level code, which removes these choicepoints without changing the behaviour of the program with respect to the query (queries).

3.1.2 Step 2: Writing strategies

As we now have defined the declarative part, we can start to define *how* the declarative part is to be used. This is done by starting with the general rules and strategies that GCLA supports from start. These rules and strategies are listed in section 2.2.

As a first refinement to the general set of rules and strategies, we could see that it would be sufficient to use an assumption when the false symbol `false` occurs as a conclusion. We can also note that the axiom rule is never used (there are other rules that are not used either, but these are used in the query `\- grey(P)`, for example). Two new strategies are constructed:

```
es <=                              % Never do axiom!
  right(es),                       % First try standard right strategy
  left_if_false(es).               % else if consequent is false...

left_if_false(PT) <=               % Is right false?
  (_ \- false).
left_if_false(PT) <=               % If so do standard left strategy.
  left(PT).
```

`es` stands for "elephant strategy". By this the number of times the above query `es \\-(grey(P) \- false)` succeeds is reduced by a factor of 3 to 3.

The corresponding table of successes and failures contains the following data:

```
a_left/4        succeeds 1 times    a_left/4        fails 15 times
a_right/2       succeeds 0 times    a_right/2       fails 22 times
d_left/3        succeeds 3 times    d_left/3        fails 13 times
d_right/2       succeeds 1 times    d_right/2       fails 21 times
es/0            succeeds 8 times    es/0            fails 14 times
false_left/1    succeeds 3 times    false_left/1    fails 13 times
left/1          succeeds 8 times    left/1          fails 10 times
left_if_false/1 succeeds 6 times    left_if_false/1 fails 16 times
o_left/4        succeeds 0 times    o_left/4        fails 16 times
o_right/3       succeeds 0 times    o_right/3       fails 22 times
pi_left/3       succeeds 0 times    pi_left/3       fails 16 times
right/1         succeeds 2 times    right/1         fails 20 times
true_right/0    succeeds 1 times    true_right/0    fails 21 times
v_left/3        succeeds 1 times    v_left/3        fails 15 times
v_right/3       succeeds 0 times    v_right/3       fails 22 times

   TOTAL: 34                           TOTAL: 256
```

If we compare these figures to the ones before, we can see that the number of inferences have been reduced, both regarding successes and failures.

The corresponding statistics about choicepoints are now:

```
These are the choicepoints:
56   14   left(es)        false_left([elephant(jumbo)])
                              elephant(jumbo),false\-false
56   14   left(es)        d_left(elephant(jumbo),[],es)
                              elephant(jumbo),false\-false
70   18   left(es)        false_left([albino_elephant(jumbo)])
                              albino_elephant(jumbo),false\-false
70   18   left(es)        d_left(albino_elephant(jumbo),[],es)
                              albino_elephant(jumbo),false\-false
```

The listing above shows that we have reduced the number of choicepoints to two. These two can also be removed. By noticing that every sequent is true if the symbol `false` occurs among the assumptions, and never try any other rules to the left if `false` occurs among the assumptions, the number of times the above query succeeds can be reduced by another factor of 3. The code implementing this is

```
es <=                              % Never do axiom!
  right(es),                       % First try standard right strategy
  left_if_false(es).               % else if consequent is false...

left_if_false(PT) <=               % Is right false?
  (_ \- false).
left_if_false(PT) <=               % If so perform left rules.
  no_false_assump(PT),
  false_left(_).

no_false_assump(PT) <=             % No false assumption
  not(member(false,A))             % i.e. the term false is not a
  -> (A \- _).                     % member of the assumption list
no_false_assump(PT) <=
  left(PT).

member(X,[X|_]).                   % Proviso definition
member(X,[_|R]) :-
  member(X,R).
```

Note how the original strategy `left` is restricted to be applicable only when there is no symbol `false` among the assumptions by the new strategy `no_false_assump`.

The table now looks like

```
a_left/4          succeeds 1 times    a_left/4          fails 12 times
a_right/2         succeeds 0 times    a_right/2         fails 20 times
d_left/3          succeeds 1 times    d_left/3          fails 12 times
d_right/2         succeeds 1 times    d_right/2         fails 19 times
es/0              succeeds 6 times    es/0              fails 14 times
false_left/1      succeeds 1 times    false_left/1      fails 26 times
left/1            succeeds 3 times    left/1            fails 10 times
left_if_false/1 succeeds 4 times      left_if_false/1 fails 16 times
no_false_assump/1 succeeds 3          no_false_assump/1 fails 11
                  times                                 times
o_left/4          succeeds 0 times    o_left/4          fails 13 times
o_right/3         succeeds 0 times    o_right/3         fails 20 times
pi_left/3         succeeds 0 times    pi_left/3         fails 13 times
right/1           succeeds 2 times    right/1           fails 18 times
true_right/0      succeeds 1 times    true_right/0      fails 19 times
v_left/3          succeeds 1 times    v_left/3          fails 12 times
v_right/3         succeeds 0 times    v_right/3         fails 20 times

   TOTAL: 24                             TOTAL: 255
```

and there are no choicepoints left, i.e. the query

```
es \\- (grey(P) \- false)
```

succeeds just once, binding `P` to `jumbo`.

We have not reduced the execution time for the query `grey(P) \- false`, since we use the same number of rules as before. But for queries that fails the execution time is reduced. For example the query

```
es \\- grey(dumbo) \- false
```

takes about 65% of the execution time of the query

```
arl \\- grey(dumbo) \- false.
```

What we have done is to reduce the original search space, i.e. the number of possible derivations, by reducing the applicability of the rules, and to restrict the rules' applicability. In this example we do not need to write special rules, since the original suffice. However, in the next example we will gain by introducing new inference rules.

3.2 Another example: Functional programming

As the second example we choose to show how functions can be implemented and executed in GCLA. The declarative part contains the functions that we want to define, while the procedural part defines how the functions are going to be executed.

3.2.1 Step 1: The declarative part

The definition contains the functions that we want to define. A simple example of a function is addition on successor arithmetic.

```
add(0,X) <= X.
add(s(X),Y) <= succ(add(X,Y)).
add(X,Y)#{X \= s(_), X \= 0} <=
  pi Z\ ((X -> Z) -> add(Z,Y)).

succ(X) <= pi Y\ ((X -> Y) -> s(Y)).
```

The third clause of `add` contains a *unification guard*, '`#{X \= s(_), X \= 0}`', i.e. a restriction on the unifier. The variable `X` is restricted not to be bound to an `s`-structure, or the constant `0`. In case `X` is not bound to anything, `X` is restricted by these guards, and these restrictions are kept for the rest of the execution. This means that these three clauses are mutually exclusive.

To start with, the general inference rules presented above will suffice, if we make some restrictions. The "numbers", i.e. `s`-structures and the constant `0`, are canonical terms, and as such any `d`-rule cannot be applied to it. Therefore we have to restrict them. Below is the code for that, together with the new strategies for `left` and `right`:

```
d_left1(T,I,PT) <=
  not(functor(T,s,1)),
  not(functor(T,0,0))
  -> (I@[T|_] \- _).
d_left1(T,I,PT) <= d_left(T,I,PT).

d_right1(T,PT) <=
  not(functor(T,s,1)),
  not(functor(T,0,0))
  -> (_ \- T).
d_right1(T,PT) <= d_right(T,PT).

right1(PT) <=
  v_right(_,PT,PT), a_right(_,PT),
  o_right(_,_,PT), d_right1(_,PT), true_right.
left1(PT) <=
  v_left(_,_,PT), a_left(_,_,PT,PT), pi_left(_,_,PT),
  o_left(_,_,PT,PT), d_left1(_,_,PT), false_left(_).

lra <= left1(lra), right1(lra), axiom(_,_,_).
```

The query

```
lra \\- add(s(0),s(0)) \- X
```

has four possible answers; `X = s(s(0))`, `X = s(add(0,s(0))`, `X = succ(add(0,s(0))`, and `X = add(s(0),s(0))`. They are all correct answers in some sense, although the first one is (mostly) the intended one.

It is worth noting that if the search order of `lra` is changed, other possible evaluation strategies can be accomplished. For example the search order `lazy`,

```
lazy <= axiom(_,_,_), left(lazy), right(lazy)
```

accomplishes some kind of lazy evaluation, which is reflected in the sequence of the solutions: the answers to the query

```
lazy \\- add(s(0),s(0)) \- X
```

are `X = add(s(0),s(0))`, `X = succ(add(0,s(0))`, `X = s(add(0,s(0))`, and `X = s(s(0))`, presented in this order.

3.2.2 Step 2: Writing strategies

The reason why all partially evaluated answers are returned is that the axiom rule is applicable to all terms, and not just to canonical terms, i.e. the constant `0` and `s`-structures. In this application the axiom rule is used to return answers. If the axiom rule is restricted analogously as the `d_left1` rule, its applicability can be reduced to the terms that are the results of evaluating expressions.

We also introduce a proviso `canonical`, which holds if the term is a canonical answer term.

```
fun_axiom(T,C,I) <=
  (canonical(T),canonical(C)) -> (I@[T|_] \- C).
fun_axiom(T,C,I) <=
  axiom(T,C,I).

d_left1(T,I,PT) <=
  not(canonical(T))
  -> (I@[T|_] \- _).
d_left1(T,I,PT) <= d_left(T,I,PT).

left(PT) <= false_left(_), v_left(_,_,PT), a_left(_,_,PT,PT),
  d_left1(_,_,PT), pi_left(_,_,PT).

eager <= left(eager), a_right(_,eager), fun_axiom(_,_,_).

canonical(X) :- var(X).
canonical(X) :- functor(X,s,1).
canonical(X) :- X == 0.
```

With the code above, the query

```
eager \\- add(s(0),s(0)) \- X.
```

returns just the wanted answer `X = s(s(0))`, and no others.

3.2.3 Step 3: Writing specialized rules

There are still some things to be done better. The `fun_axiom` rule can be changed from a restriction implemented as a strategy (as above) to a specialized rule. The `succ`-clause (referred to as a *substitution clause* or *evaluation clause*, since its argument is substituted, or evaluated, to a canonical value) is always evaluated in a special way, namely first by an application of the `d_left` rule (through the `d_left1` strategy), then by an application of the `pi_left` rule followed by the `a_left` rule. The premise of the arrow is evaluated by `a_right` and the conclusion is a returned, canonical term to which the `fun_axiom` is applied. We implement this sequence of rule applications by the strategy `subst_strat`, which first checks whether the chosen term is a `succ`-term or not, and if so, applies this sequence of rule applications. We have also extended the proviso in the `d_left1` strategy not to handle `succ`-terms.

```
fun_axiom(T,C,I) <=
  canonical(T),
  canonical(C),
  unify(T,C)
  -> (I@[T|_] \- C).

d_left1(T,I,PT) <=
  not(canonical(T)),
  not(functor(T,succ,1))
  -> (I@[T|_] \- _).
d_left1(T,I,PT) <= d_left(T,I,PT).

subst_strat(T,_) <=
  functor(T,succ,1) -> (I@[T|_] \- _).
subst_strat(T,PT) <=
  d_left(T,_,pi_left(_,_,a_left(_,_,a_right(_,PT),
                                fun_axiom(_,_,_)))).

eager <=
  subst_strat(_,eager), d_left1(_,_,eager), pi_left(_,_,eager),
  fun_axiom(_,_,_), a_left(_,_,a_right(eager),eager).

canonical(X) :- var(X).
canonical(X) :- functor(X,s,1).
canonical(X) :- X == 0.
```

By these small changes, we have reduced the execution time by about 50%, compared to the previous solution. This comes from the new rule `fun_axiom` and from the sequence of rule applications in `subst_strat`, which cuts off a lot of rule tests.

The code can still be more efficient by some simple changes. The strategy `subst_strat` can be turned into a new rule, `subst`, which performs the whole sequence of rule applications of `subst_strat` in one rule step. We also remove the `succ`-clause from the definition and "lift" it to be on the same level as the arrow "->". This means that the `succ(...)` condition has its own rule, `subst`. This is done by the proviso `constructor`, which is used to declare such condition-constructors.

We are also introducing a new rule (`eval`) for handling the third clause of `add`, i.e. a rule which evaluates the first argument of add if it is not on canonical form. The proviso `evalschema` defines when and which arguments that should be evaluated. The evaluation takes place in the second row of the rule clause (`PT -> I@R \- C`). An example of a query which uses this schema is `add(add(s(0), s(0)), s(0))` which evaluates to `s(s(s(0)))`.

```
fun_axiom(T,C,I) <=
  canonical(T), canonical(C),
  unify(T,C)
  -> (I@[T|_] \- C).

d_left1(T,I,PT) <=
  not(canonical(T))
  -> (I@[T|_] \- _).
d_left1(T,I,PT) <= d_left(T,I,PT).

subst(succ(A),PTs) <=
  unify(Concl,s(T1)),
  (PTs -> (I@[A|R] \- T1))
  -> (I@[succ(A)|R] \- Concl).

eval(T,PTs) <=
  evalschema(T,T1,C),
  (r(PTs) -> (I@R \- C)),
  (PTs -> (I@[T1|R] \- Concl))
  -> (I@[T|R] \- Concl).

evalschema(add(A,B),add(A1,B),(A -> A1)) :-
  not(A = 0), not(A = s(_)).

eager <= eval(_,eager), subst(_,eager),
  d_left1(_,_,eager), fun_axiom(_,_,_).

r(PT) <= a_right(_,PT), c_right(_,a_right(PT), a_right(_,PT)).

canonical(X) :- var(X).
canonical(X) :- functor(X,s,1).
canonical(X) :- X == 0.

constructor(succ,1).
```

With this coding the execution times are reduced by another one third of the previous version. There are of course further improvements to be done, but we stop here.

4. Other examples

There are a lot of other example programs and applications which have been developed. We will list some of them here, and point out the interesting techniques and other points of interest. We will not present the development of the programs as we did above, but comment on the "final" program.
The primitive `include` that occurs in the beginning of all the rule files loads the file `rules.rul` from the GCLA rules' library, i.e. sets GCLA up with the general rules described in the appendix.

4.1 Sorting

This example shows integration of relational and functional programming, and has been presented in more detail in [Aro91b]. It has some interesting properties. One can note that the functional part, the clauses defining `qsort` and `append`, are executed by one set of inference rules, mostly left hand side rules (see the rules for functional execution above). The relational part is a horn clause definition, which consists of the clauses defining `split`, and these are executed by the right hand side rules (`v_right` and `d_right`). The intersection of these two sets are empty, so there is a "border" between

the functional and relational execution in this application. This means that these two parts can be further developed without disturbing each other, for example it is possible to "plug in" better functional execution strategies than presented here.

Definition:

```
A = A.

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 \= [_|_],X \= []} <=
  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 X1 \ (pi Y1 \ ((X -> X1), (Y -> Y1) -> [X1|Y1])).
```

Below are the code for the inference rules and strategies in the qsort example. In principle we use the general rules, but specializes the strategies.

The provisos < and >= are defined on numbers in the usual way.

(The corresponding rules shows how parts of the underlying system, in this case Prolog, can be incorporated into GCLA. These relations should in fact be part of the object level system in a more complete implementation of GCLA.)

Rules and strategies for qs:

```
:- include(library('rules.rul')).

%%% Rules

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

%%% Restrictions of rules defined in rules.rul
q_axiom(T,I) <=
  (data(T) ->
   (I@[T|_] \- _)).
q_axiom(T,I) <=
  axiom(T,C,I).

q_d_left(T,I,PT) <=
  (not(data(T)) ->
   (I@[T|_] \- _)).
q_d_left(T,I,PT) <=
  d_left(T,I,PT).
```

```
split_right(PT) <=
  (_ \- split(_,_,_,_)).
split_right(PT) <=
  d_right(_,PT).

%%% Strategies
qs <= q_fun(qs),          % Functional execution
  q_rel(qs).              % Relational execution

q_fun(PT) <=
  a_right(_,PT), v_left(_,_,PT), a_left(_,_,PT,PT),
  pi_left(_,_,PT), q_d_left(_,_,PT), q_axiom(_,_).

q_rel(PT) <=
  v_right(_,PT,PT),
  split_right(r).

r <= right(r), right_g_e, right_l.

%%% Provisos
data([]).
data([_|_]).
data(X) :- number(X).
```

`data` fills the same role as `canonical` did in the add-example in section 3.2, i.e. we have restricted the axiom rule to be applicable only on data terms, and the `d_left` rule to be applicable on terms that are not data. In the sorting procedure we have restricted the relational execution to start with the atom `split`, which can be seen in the strategies `split_right` and `q_rel`.

The top level strategy `qs` returns either a strategy for functional evaluation or a strategy for relational evaluation.

A small extension of the general strategies and rules presented in appendix A are used for comparison of figures and behaviour.

Rules and strategies for `lra`:

```
%%% Extension of the general strategy lra
lra <= left1(lra),right1(lra),q_axiom(_,_).

right1(S) <= true_right, v_right(_,S,S), a_right(_,S),
   o_right(_,_,S), d_right(_,S), right_g_e, right_l.
left1(S) <= false_left(_), v_left(_,_,S), a_left(_,_,S,S),
   o_left(_,_,S,S), q_d_left(_,_,S), pi_left(_,_,S).
```

In this example it suffice to look at one kind of queries, queries that sorts a list. The query

```
lra \\- qsort([3,4,1,5,2]) \- P.
```

has the following statistics to find the first solution `P = [1,2,3,4,5]`:

```
a_left/4        succeeds 17 times
a_right/2       succeeds 19 times
axiom/3         succeeds 20 times
d_left/3        succeeds 30 times
d_right/2       succeeds 11 times
false_left/1    succeeds 0 times
left1/1         succeeds 76 times
lra/0           succeeds 150
times
o_left/4        succeeds 0 times
o_right/3       succeeds 0 times
pi_left/3       succeeds 29 times
q_axiom/2       succeeds 20 times
q_d_left/3      succeeds 30 times
right1/1        succeeds 54 times
right_g_e/0     succeeds 2 times
right_l/0       succeeds 4 times
true_right/0    succeeds 5 times
v_left/3        succeeds 0 times
v_right/3       succeeds 13 times

   TOTAL: 480
```

```
a_left/4        fails 151 times
a_right/2       fails 55 times
axiom/3         fails 0 times
d_left/3        fails 29 times
d_right/2       fails 44 times
false_left/1    fails 168 times
left1/1         fails 92 times
lra/0           fails 18 times
o_left/4        fails 151 times
o_right/3       fails 55 times
pi_left/3       fails 92 times
q_axiom/2       fails 18 times
q_d_left/3      fails 121 times
right1/1        fails 38 times
right_g_e/0     fails 42 times
right_l/0       fails 38 times
true_right/0    fails 87 times
v_left/3        fails 168 times
v_right/3       fails 74 times

   TOTAL: 1441
```

The query

```
qs //- qsort([3,4,1,5,2]) \- P.
```

has the corresponding statistics (to find the first answer):

```
a_left/4        succeeds 17 times
a_right/2       succeeds 19 times
axiom/3         succeeds 20 times
d_left/3        succeeds 30 times
d_right/2       succeeds 11 times
o_right/3       succeeds 0 times
pi_left/3       succeeds 29 times
q_axiom/2       succeeds 20 times
q_d_left/3      succeeds 30 times
q_fun/1         succeeds 115
times
q_rel/1         succeeds 12 times
qs/0            succeeds 127
times
r/0             succeeds 23 times
right/1         succeeds 17 times
right_g_e/0     succeeds 2 times
right_l/0       succeeds 4 times
split_right/1   succeeds 5 times
true_right/0    succeeds 5 times
v_left/3        succeeds 0 times
v_right/3       succeeds 13 times

   TOTAL: 499
```

```
a_left/4        fails 91 times
a_right/2       fails 138 times
axiom/3         fails 0 times
d_left/3        fails 0 times
d_right/2       fails 24 times
o_right/3       fails 30 times
pi_left/3       fails 62 times
q_axiom/2       fails 12 times
q_d_left/3      fails 32 times
q_fun/1         fails 12 times
q_rel/1         fails 0 times
qs/0            fails 0 times
r/0             fails 18 times
right/1         fails 24 times
right_g_e/0     fails 22 times
right_l/0       fails 18 times
split_right/1   fails 0 times
true_right/0    fails 36 times
v_left/3        fails 108 times
v_right/3       fails 35 times

   TOTAL: 662
```

and we can see that the number of calls (succeeded and failed) has decreased significiantly. An exhaustive search for this query gives the following table of total succeeded and failed calls:

	lra	qs
Succeeded calls	TOTAL 480	TOTAL 499
Failed calls	TOTAL 2574	TOTAL 1422
Total number of calls	TOTAL 3054	TOTAL 1921

A significiant better performance for the `qs` strategy.

4.2 STRIPS like planning

STRIPS is a planning system, invented by Nils Nilsson [Nil82]. In short, the system has a global database, which is altered when a planning operation is executed. From a starting state the system performs planning operations until it has reached some goal state, and the resulting sequence of planning operations is the plan.

In our case the planning operations are called `action`, and is a conditional function from one state to another. Whether an action is possible to perform or not is determined by the relation `possible`, and `perform` changes the global database. `rem_cl`/2 and `def_cl`/2 are defined in the rule code using the proviso primitives `rem`/1 and `def`/1, and removes respectively defines the clause in its first argument.

Definition:

```
X = X.

%%-------
% Initial state
on(a,b).
on(b,c).
table(c).
clear(a).

% action is a function from an action A to a state sit.
action(A,sit(S)) <= (possible(A) -> perform(A,sit(S))).
action(A,action(X,S)) <=
  pi Y\ ((action(X,S) -> sit(Y)) -> action(A,sit(Y))).

possible(stack(X,Y)) <=
  table(X),clear(X),clear(Y),(X = Y -> false).
possible(unstack(X,Y)) <= on(X,Y),clear(X).
possible(move(X,Y,Z)) <=
  on(X,Y),clear(X),clear(Z),(X = Z -> false).

perform(stack(X,Y),sit(S)) <=
   rem_cl(table(X),rem_cl(clear(Y),def_cl(on(X,Y),sit(s(S))))).
perform(unstack(X,Y),sit(S)) <=
   rem_cl(on(X,Y),def_cl(table(X),def_cl(clear(Y),sit(s(S))))).
perform(move(X,Y,Z),sit(S)) <=
   rem_cl(on(X,Y),
          rem_cl(clear(Z),
                 def_cl(clear(Y),def_cl(on(X,Z),sit(s(S)))))).
```

We present two different sets of meta level code. In the first the inference rules are the common ones, i.e. the general rules presented in section 2.2, together with some new rules. We here show how the primitive provisos `def` and `rem` are incorporated by the rules `def_left`, `def_right`, `rem_left` and `rem_right`. The proviso `def(R)` asserts the clause `R` in the current definition, but upon backtracking removes the same clause

again. `rem(R)` does the opposite, i.e. removes all clauses that are unifiable with `R`, but does not unify variables in `R`. This means that `rem` always succeeds, sometimes without removing anything. Upon backtracking the removed rules are added again.

We declare two new symbols, `def_cl/2` and `rem_cl/2`, as condition constructors, i.e. at the same level as the arrow "->", the comma ", " etc.

The term `sit(...)` is the return answer from the `action`-function, and is returned from the `perform`-clauses. As it is a returned answer, it should not be subject to the `d_left` rule, so the rule `s_d_left` is restricted not to handle that term.

Rules and strategies for `strips`:

```
:- include(library('rules.rul')).

def_left(def(X,Y),I,PT) <=
 def(X),
 (PT -> (I@[Y|R] \- C)) ->
 (I@[def_cl(X,Y)|R] \- C).
rem_left(rem(X,Y),I,PT) <=
  rem(X),
  (PT -> (I@[Y|R] \- C)) ->
  (I@[rem_cl(X,Y)|R] \- C).
def_right(PT) <=
  def(X),
  (PT -> (A \- Y)) ->
  (A \- def_cl(X,Y)).
rem_right(PT) <=
  rem(X),
  (PT -> (A \- Y)) ->
  (A \- rem_cl(X,Y)).

s_d_left(T,I,PT) <=
  not(functor(T,sit,1)) -> (I@[T|_] \- _).
s_d_left(T,I,PT) <=
  d_left(T,I,PT).

strips <= s_left(strips),s_right(strips),axiom(_,_,_).

s_left(PT) <= false_left(_), def_left(_,_,PT), rem_left(_,_,PT),
  pi_left(_,_,PT), v_left(_,_,PT), a_left(_,_,PT,PT),
  o_left(_,_,PT,PT), s_d_left(_,_,PT).

s_right(PT) <=  true_right, def_right(PT), rem_right(PT),
  v_right(_,PT,PT), a_right(_,PT), o_right(_,PT), d_right(_,PT).

constructor(def_cl,2).
constructor(rem_cl,2).
```

With the above setting, i.e. more or less the general rules of GCLA, the behaviour is correct, but inefficient. Some example queries are

1)
```
strips \\- action(X,sit(0)) \- table(a).

X = unstack(a,b) ;

X = unstack(a,b) ;

X = unstack(a,b) ;

no
```

2)
```
strips \\- action(X,action(Y,action(Z,sit(0)))) \- on(c,b).

X = stack(c,b),
Y = unstack(b,c),
Z = unstack(a,b) ? ;

X = stack(c,b),
Y = move(b,c,a),
Z = unstack(a,b) ? ;

no
```

3)
```
strips \\- action(unstack(b,c),action(unstack(a,b),sit(0)))
       \- sit(s(s(0))),table(X).

X = b ? ;

X = a ? ;

X = c ? ;

X = b ? ;

X = b ? ;

X = c ? ;

X = a ? ;

X = c ? ;

... There are 150 answers to this query.
```

The first two queries represent planning, while the third represents simulation. In the third query the answers are repeated over and over again.

A much more efficient rule code is the one presented below. On the top level there are two possibilities, either a planning step should be performed or the execution is finished. We are finished if the only assumption is `sit(_)`, in which case we start to examine the object level conclusion. The strategy `act`'s two proofterms handle the two object level clauses, i.e. the first proofterm performs a planning step and the other defines the sequence of rules for the substitution clause (c.f. functional programming, section 3.2). `perf` together with `remdef` alter the global database.

Rules and strategies for `str`:

```
str <= act(rl,perf(str)), finished.

act(S1,S2) <= ([action(A,S)] \- _).
act(S1,S2) <=
  d_left(_,_,a_left(_,_,S1,S2)),
  d_left(_,_,pi_left(_,_,a_left(_,_,a_right(_,str),str))).

perf(_) <= not(functor(T,sit,1)) -> ([T] \- _).
perf(S) <=
  d_left(_,_,remdef).

remdef <= def_left(_,_,remdef), rem_left(_,_,remdef),
  axiom(_,_,_), finished.

finished <= not(functor(C,sit,1)) -> ([sit(_)] \- C).
finished <= ra.

ra <= right(ra), axiom(_,_,_).
rl <= right(rl), left(rl).
```

The corresponding queries for 1 - 2 before have the same behaviour regarding the answers, but the query 3 has stopped to loop:

```
str \\- action(unstack(b,c),action(unstack(a,b),sit(0)))
   \- sit(s(s(0))),table(X).

X = b ? ;

X = a ? ;

X = c ? ;

no
```

We can also omit the `sit(s(s(0)))` in the query 3. It was used in the first version to assure that the actions were performed before `table(X)` was tested.

If we look at the number of calls for query 1, we have the following table for the two strategies:

Number of calls	strips	str
First answer:	TOTAL: 293	TOTAL: 143
Exhaustive search	TOTAL: 2465	TOTAL: 479

The strategy `str` has significiant better figures than `strips`, and we have also removed redundant answers with the new strategy.

4.3 Object oriented programming

It is possible to get a kind of object oriented programming using the assumptions as objects, whose arguments hold the object's internal state. The objects can communicate with each other by using two techniques: either using shared variables, a common technique in logic programming, or by using the arrow- and axiom rules, which instantiates an argument in a given (named) object.

An object is suspended if its first argument is unbound. This is taken care of by the `susp_object` condition. It acts as `freeze` in many cases, a delaying operation in some Prologs. A `susp_object` assumption is reachable by the `s_axiom` rule, and therefore we can instantiate the variable by an application of the `s_axiom` rule. This gives the two possibilities to send messages: instantiating a variable that is shared, or by knowing the term representing the object and use the `s_axiom` rule.

In our example, the objects of the `ship` class get messages by the `s_axiom` rule, while the objects of the `sailing_ship` class get their messages by a shared variable, i.e. a stream of messages, implemented by a list, where the tail of the list is always unbound and used for the next message.

Definition:

```
X = X.

ship(weight(W),N,W,L) <= susp_object(R,class,ship(R,N1,W1,L1)).
ship(name(N),N,W,L) <= susp_object(R,class,ship(R,N1,W1,L1)).
ship(length(L),N,W,L) <= susp_object(R,class,ship(R,N1,W1,L1)).
ship(new_sailship(S,MaxS),N,W,L) <=
  susp_object(S,N,sailing_ship(S,N,W,L,0,MaxS)),
  susp_object(R,class,ship(R,N1,W1,L1)).
ship([],N,W,L).

sailing_ship([max_sailarea(MaxS)|R],N,W,L,S,MaxS) <=
  susp_object(R,N,sailing_ship(R,N,W,L,S,MaxS)).
sailing_ship([current_sailarea(S)|R],N,W,L,S,MaxS) <=
  susp_object(R,N,sailing_ship(R,N,W,L,S,MaxS)).
sailing_ship([change_sailarea(S1)|R],N,W,L,S,MaxS) <=
  (S1 >= 0,S1 =< MaxS ->
   susp_object(R,N,sailing_ship(R,N,W,L,S1,MaxS))).
sailing_ship([],N,W,L,S,MaxS).
sailing_ship([F|R],N,W,L,S,MaxS)#
       {F \= max_sailarea(_),
        F \= current_sailarea(_),
        F \= change_sailarea(_)} <=
  (susp_object(F,class,ship(Y,N,W,L)) ->
   susp_object(R,N,sailing_ship(R,N,W,L,S,MaxS))).
```

Inheritance is implemented as in the last clause of `sailing_ship`. The arrow is used to pass the message to the class above the current class (in this case from the `sailing_ship` class to the `ship` class). So if a `sailing_ship` does not know how to answer a message, it passes it to the `ship` class. A generic derivation of this message passing is:

$$\dfrac{\dfrac{}{superclass(X) \ \backslash\!\!- \ superclass(1)}\{X/1\} \qquad \dfrac{}{superclass(1),\ subclass(Y) \ \backslash\!\!- \ messages}\ldots}{\dfrac{superclass(X),\ (superclass(1) \rightarrow subclass(Y)) \ \backslash\!\!- \ messages}{superclass(X),\ subclass(1) \ \backslash\!\!- \ messages}}$$

In the left branch of the tree the axiom rule passes the message from *superclass(1)* to *superclass(X)*, binding *X* to *1*, and in the right branch the message has been passed from the subclass to the superclass, and the execution proceeds with superclass.

In this application it is suitable to introduce some new inference rules. The condition `susp_object(...)` is declared to be a constructor, and a rule for `susp_object` is introduced.

The axiom rule is changed to be applicable on suspended objects only, implemented by the new `s_axiom` rule. This is how an object is asked to do something. `ready` is just a rule to finish the execution on top level. It is applicable when we have got an answer instantiation to our query. The rule `obj_left` is used instead of the rule `d_left`, and gets the definition of the method that the object should execute. There are also rules for numeric comparisons.

Rules and strategies for `obj`:

```
:- include(library('rules.rul')).

s_axiom(T,C,I) <=
  functor(T,susp_object,3),
  functor(C,susp_object,3),
  unify(T,C)
  -> (I@[T|_] \- C).

s_o(PT) <=
  nonvar(X),
  (PT -> (I@[Y|R] \- C))
  -> (I@[susp_object(X,Name,Y)|R] \- C).

ready <= nonvar(X) ->
  (_ \- answer(X)).

obj_left(T,X,PT) <=
  functor(T,O,_),
  object(O),
  definiens(T,Dp,N),
  N > 0,
  (PT -> (X@[Dp|Y] \- C))
  -> (X@[T|Y] \- C).

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

constructor(susp_object,3).

object(sailing_ship).
object(ship).

%--------------
obj <= ready,                                % end of execution
  v_right(_,send_message,reduce(obj)). % Next call to an object

send_message <= s_axiom(_,_,_),ar.

reduce(PT) <=
  s_o(obj_left(_,_,handle(PT))).         % Reduce a called object

handle(PT) <=
  a_left(_,_,s_axiom(_,_,_),reduce(PT)),% Calling another object
  s_o(obj_left(_,_,handle(PT))),          % A not empty stream
  a_left(_,_,ar,reduce(PT)),              % An if-stmnt in a method
  v_left(_,_,handle(PT)),PT.              % Splitting of a process
```

```
% ar is used to evaluate conditions in methods
ar <= right_l_e, right_g_e,
  v_right(_,ar,ar), d_right(_,ar), true_right.

gcla <= obj.
```

The top level strategy is `obj`. It is always the case that the left term in a vector should pass a message to an object, so the only rule for the term to the left in a vector is the `s_axiom` rule, or some rule to the right to bind a stream variable. Before considering the right term in the vector, we should reduce the object that got a message, which is handled by `reduce`. When the object has finished its execution, the strategy `obj` is used again.

`s_o` is applicable on objects that have received a message in their message variable, or message streams. `obj_left` finds the definition for the object's action in the definition, and `handle` executes that action. The comments explain what the different proofterms in the vector of `handle` perform.

The reader may have noticed that the rule code can be further improved. For example, since the rule `s_o` is always followed by an application of `obj_left`, these two rules could be concatenated. In the same way `v_right` can be removed.

Some queries are

1) We ask the object `fia` by its stream argument what its current sailarea is:

```
susp_object(X,fia,sailing_ship(X,fia,3600,8,0,45)) \-
  X = [current_sailarea(S)|_],answer(S).

S = 0,
X = [current_sailarea(0)|_A] ? ;

no
```

2) A more complicated query. We ask the sailing ship `fia` about its name, and there are no methods for finding names among the `sailing_ship` methods, so the question is passed to the `ship` class.

```
susp_object(Z,class,ship(Z,A,B,C)),
susp_object(X,fia,sailing_ship(X,fia,3600,8,0,45)) \-
  X = [name(N)|_],answer(N).

A = fia,
B = 3600,
C = 8,
N = fia,
X = [name(fia)|_A],
Z = name(fia) ? ;

no
```

2') This is the same query as above, without a ship class that can answer the name-message:

```
susp_object(X,fia,sailing_ship(X,fia,3600,8,0,45)) \-
  X = [name(N)|_],answer(N).

no
```

3) Here a new sailing ship is created, the current sailarea is changed, and we then ask the new sailing ship about its current sail area.

```
susp_object(Z,class,ship(Z,A,B,C)) \-
  susp_object(new_sailship(S,45),class,ship(_,grete,3600,9)),
  S = [change_sailarea(25),current_sailarea(Area)|_],
  answer(Area).

...
Area = 25,
...
```

For comparison purposes, we include a rule code version based on the general rules and strategies. In order to use one of the general strategies for comparing purposes, `right` and `left` must be extended with the new rules `right_l_e`, `right_g_e`, `ready` and `s_o` above. The rules `d_left` and `d_right` must also be restricted not to handle `susp_object`-conditions, which is done through the `constructor` primitive.

Rules and strategies for `lra1`:

```
:- include(library('rules.rul')).

s_axiom(T,C,I) <=
  functor(T,susp_object,3),
  functor(C,susp_object,3),
  unify(T,C)
  -> (I@[T|_] \- C).

s_o(Cont) <=
  nonvar(X),
  (Cont -> (I@[Y|R] \- C))
  -> (I@[susp_object(X,Name,Y)|R] \- C).

ready <= nonvar(X) ->
  (_ \- answer(X)).

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

constructor(susp_object,3).

%%%---------------------
lra1 <= left1(lra1), right1(lra1), axiom(_,_,_), s_axiom(_,_,_).

right1(S) <=
  true_right, right_l_e, right_g_e, v_right(_,S,S),
  a_right(_,S), o_right(_,_,S), d_right(_,S), ready.
left1(S) <=
  s_o(S), false_left(_), v_left(_,_,S), a_left(_,_,S,S),
  o_left(_,_,S,S), d_left(_,_,S), pi_left(_,_,S).
```

We can now compare these two approaches with each other. The first one is a highly specialized one while in the second one we have not performed any special search behaviour at all. The performance and behaviour are very different, as the table below shows.

Number of answers	lra1	obj
Query no 1	5	1
Query no 2	128	1
Query no 3	140	1

(Note, however, that for the `lra1` strategy, we get the same answer susbtitution all the time.)

For the queries above (1 - 3) the number of calls performed are shown in the table below. (The missing figures for exhaustive search are due to enourmous execution times. Compare the number of answers given above with the number of calls for the query 1, and scale the figures for `lra1` appropriately to get the figures for query 2 and 3.)

Query number	lra1		obj	
	First answer	Exhaustive search	First answer	Exhaustive search
1	102	640	26	28
2	187	-	32	46
3	264	-	61	113

4.4 Guiding the search among the assumptions

In many applications it happens that many assumptions are applicable simultaneously. At these points it is very desirable to reduce the search space by taking one of the applicable assumptions and leave the others. For example consider the definition

```
p(X) <= b1(X).
q(X) <= b2(X).
```

and the sequent

```
p(1), q(2) \- something.
```

In this case there are two possibilities leading to the same sequent, namely first resolve `p(1)` and then `q(2)`, or first try `q(2)` and then `p(1)`, both leading to

```
b1(1), b2(2) \- something.
```

By a few strategy definitions this kind of behaviour can be avoided. We will first demonstrate one version that gives a plausible behaviour in most cases, and then an extended version. These versions can for example be used in the object oriented programming example in section 4.3. The extended version was invented during the development of a terminological reasoning application, which can be found in [Han92].

The example definition that we are going to use here is a small, academic one, since it is the behaviour of the meta level that is interesting.

Definition:

```
p1(1).
p2(2).
p3(3).
```

The first version of the rule definition consists of four strategies and one proviso. `ars` is the top level strategy. The ordinary axiom rule and right strategy are tried, and then the search strategies are tried. `search1` and `search` are mutually exclusive; `search1` handles the case when the left most term among the assumptions is applicable, and `search` handles the rest of the assumptions. An assumption is applicable if the term to the left of it is not applicable, which means that all derivations where the choosen term has an applicable term to the left are removed.

Rules and strategies for `ars`:

```
:- include(library('rules.rul')).

ars <= axiom(_,_,_),
  right(ars),
  search1(_,ars),
  search(_,ars).

search1(T,PT) <= applicable(T) -> ([T|_] \- C).
search1(T,PT) <= left1(T,PT).

search(T,PT) <=
  (applicable(T),not(applicable(T1)) -> (I@[T1,T|Rest] \- C)).
search(T,PT) <= left1(T,PT).

left1(T,PT) <=
  false_left(_),v_left(T,I,PT),a_left(T,I,PT,PT),
  o_left(T,I,PT,PT),d_left(T,I,PT),pi_left(T,I,PT).

%%% Proviso

applicable(T) :- atom(T).
applicable(T) :- functor(T,F,A),
  not(F = true), constructor(F,A).
```

With the query `ars \\- p1(X),p2(Y),p3(Z) \- q(A)` we get 4 possible answers (i.e. 4 derivations). The only way to instantiate a term to the right of another term in the antecedent is to first instantiate the term to the left, as we can see below where first `X` is bound to `1`, then `Y` is bound to `2` etc, but never `Y` bound to `2` without binding `X` to `1`;

```
ars \\- p1(X),p2(Y),p3(Z) \- true.
```

gives the following, complete list of answers (we have compacted the set of possible variable instantiations to tripples):

```
[(X,Y,Z),(1,Y,Z),(1,2,Z),(1,2,3)]
```

The query `arl \\- p1(X),p2(Y),p3(Z) \- q(A)` succeeds 16 times (i.e. 16 derivations):

```
[(X,Y,Z),(1,Y,Z),(1,2,Z),(1,2,3),(1,Y,3),(1,2,3),(X,2,Z),
 (1,2,Z),(1,2,3),(X,2,3),(1,2,3),(X,Y,3),(1,Y,3),(1,2,3),
 (X,2,3),(1,2,3)]
```

The interesting choicepoints for the strategy `arl` are:

```
...
14   2    d_left(p1(1))        arl       true,p2(_1492),p3(_1496)\-true
14   2    d_left(p2(2))        arl       p1(_1370),true,p3(_1378)\-true
14   2    d_left(p3(3))        arl       p1(_1248),p2(_1252),true\-true
...
28   5    d_left(p2(2))        arl       true,true,p3(_2624)\-true
28   5    d_left(p3(3))        arl       true,p2(_2510),true\-true
...
103  5    d_left(p1(1))        arl       true,true,p3(_4083)\-true
103  5    d_left(p3(3))        arl       p1(_3967),true,true\-true
...
178  5    d_left(p1(1))        arl       true,p2(_5534),true\-true
178  5    d_left(p2(2))        arl       p1(_5422),true,true\-true
...
```

The choicepont at call 28 is when the system chooses `p1` first, the choicepoint at call 103 is when `p2` is choosen first, and the choicepoint at call 178 is when `p3` is choosen first.

The number of calls for these two strategies are:

	arl	ars
Number of calls	TOTAL: 240	TOTAL: 61

By comparing the lists above, we can see that to the first query, we have no tripple where `Y` or `Z` is instantiated, but not `X`. In the second query we have. However, this is not a deterministic choice of an assumption in the general case, since the goal sequent

```
ars \\- p1(X),p2(Y),true,p3(Z) \- true
```

where the applicable assumptions are underlined, has two possible choices (among the assumptions), namely `p1(X)` and `p3(X)`. We can see that below, where `Z` can be bound to `3` before `X` is bound to `1` (relate this list of answers with the one before).

```
[(X,Y,Z),(1,Y,Z),(1,2,Z),(1,2,3),(1,Y,3),(1,2,3),(X,Y,3),
 (1,Y,3),(1,2,3)]
```

By introducing a meta level symbol among the assumptions, that in much behaves as a pointer into the assumptions, it is possible to reduce the choice of an assumption to a deterministic choice. This is done in the rule definition below. `ars1` is the top level strategy, which in the last term of the vector introduces a mark among the assumptions, then searches the assumptions and if one applicable is found and reduced, `ars1` is used again. `search` tries to use the `perform` strategy, and if that is not possible, it proceeds with the search. `introduce_mark` just introduces the mark `mark` first among the assumptions. This mark will then be moved to the right as the search is continued until an applicable term next to the right of the mark is found. `proceed_with_search` just moves the mark one step to the right, i.e. exchanges `mark` and the term to the right of `mark`, if this term is not applicable. It is this step that is the crucial step, since if the term was applicable, this step is not allowed, and the system will not be able to proceed with the search, which means that other, applicable terms further to the right will not be choosen.

`perform` picks the term next to the right of `mark`, tests if it is applicable, and if so, removes `mark` by applying `rm_mark`, and then applies the `left1` strategy to it. `rm_mark` just removes the mark before it applies the proofterm in its argument.

Rules and strategies for `ars1`:

```
:- include(library('rules.rul')).

ars1 <= axiom(_,_,_),
  right(ars1),
  introduce_mark(search(ars1)).

search(PT) <= perform(_,_,PT), proceed_with_search(search(PT)).

introduce_mark(PT) <=
  (PT -> ([mark|X] \- C))
   -> (X \- C).

proceed_with_search(PT) <=
  not(applicable(T)),                    % test not applicable
  (PT -> (X@[T,mark|Rest] \- C))
  -> (X@[mark,T|Rest] \- C).

perform(I,T,PT) <= (applicable(T) -> (I@[mark,T|Rest] \- C)).
perform(I,T,PT) <= rm_mark(left1(T,I,PT)).

left1(T,I,PT) <=
  false_left(I), v_left(T,I,PT), a_left(T,I,PT,PT),
  o_left(T,I,PT,PT), d_left(T,I,PT), pi_left(T,I,PT).

rm_mark(PT) <=
  (PT -> (X@Y \- C))
  -> (X@[mark|Y] \- C).

%%% Proviso

applicable(T) :- atom(T).
applicable(T) :- functor(T,F,A),
  not(F = true), constructor(F,A).
```

With this rule definition, we have the same behaviour as the `ars` strategy explained before, except that there will be just one applicable term among the assumptions, the left most assumption for which the `applicable` proviso holds. The query

```
ars1 \\- p1(X),p2(Y),true,p3(Z) \- true
```

now gives the complete answer list

```
[(X,Y,Z),(1,Y,Z),(1,2,Z),(1,2,3)]
```

There is also the possibility to write other search strategies among the assumptions, but they will be more complicated to write down. We are currently working on these matters, and trying to improve the meta level language to support other kinds of techniques to remove and cut away choicepoints.

Also note that in the general case one cannot use the above strategies. There are cases when the above strategies cuts off derivations that leads to other, correct answers, i.e. there are cases when one assumption must be reduced before another one. In other

words, the derivation depends on the outlook of the assumptions *themselves*, and not the *order* of the assumptions.

5. Conclusion

Comparing GCLA II with GCLA I, GCLA II offers a much better way to implement control and search algorithms, which is also much cleaner than in GCLA I. By distinguishing the control part, and making the control part separate from the declarative part, we get a very clear and understandable programming system, where the development of the procedural system can be performed without disturbing the declarative part. This is an advantage, especially if several persons want to use the same declarative database. By having a clear distinction between the object level and the meta level we get a much nicer and clearer understanding of how system behaves. There are, however, further developments to be done. For example it should be possible to express such things as "if this *derivation* succeeds, do not try these possible derivations" etc. As it is now, we cannot express if a derivation succeeds or fails. There are also some other things to be elaborated on, for example meta level negation and some other meta level language improvements. With these additional improvements we expect GCLA II to be a good system for developing programs, especially KBS programs of various kinds.

References

[Aro90] M. Aronsson, L-H. Eriksson, A. Gäredal, L. Hallnäs, P. Olin, *The Programming Language GCLA: A Definitional Approach to Logic Programming*, New Generation Computing 7(4), pp. 381 - 404, 1990.

[Aro91a] M. Aronsson, *GCLA User's manual*, Technical Report SICS T91:21, 1991

[Aro91b] M. Aronsson, *A Definitional Approach to the Combination of Functional and Relational Programming*, Research Report SICS R91:10, 1991

[Eri92] L-H. Eriksson, *A Finitary Version of the Calculus of Partial Inductive Definitions*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, Februari 1991, Springer Lecture Notes in Artificial Intelligence.

[Hal91] L. Hallnäs, *Partial Inductive Definitions*, Theoretical Computer Science vol. 87 pp 115 - 142, 1991

[HS-H88] L. Hallnäs, P. Schroeder-Heister, *A Proof-Theoretic Approach to Logic Programming*, published in two parts in the Journal of Logic and Computation, part I: *Clauses as Rules*, vol. 1(2), pp 261 - 283, 1990, part II: *Programs as Definitions*, vol. 1(5), pp 635 - 660, 1991.

[Han92] P. Hanschke, *Terminological Reasoning and Partial Inductive Definitions*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, Februari 1991, Springer Lecture Notes in Artificial Intelligence.

[Kre92] P. Kreuger, *GCLAII, A Definitional Approach to Control*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, Februari 1991, Springer Lecture Notes in Artificial Intelligence.

[Nil82] N. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag, 1982.

Appendix A

This appendix contains the inference rules and strategies that are loaded into the GCLA II system when it is started. The file is submitted with the system together with some other files implementing common, general inference rules and strategies. The user can freely copy and change this file.

```
% Rules

true_right <= (_ \- true).

false_left(I) <= (I@[false|_] \- _).

axiom(A,C,I) <=
  term(C),
  term(A),
  unify(C,A)
  -> (I@[A|_] \- C).

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

d_left(A,I,PT) <=
  atom(A),
  definiens(A,D),
  (PT -> (I@[D|R] \- C))
  -> (I@[A|R] \- C).

a_left((B -> C1),I,PT1,PT2) <=
  (PT1 -> (I@R \- B)),
  (PT2 -> (I@[C1|R] \- C))
  -> (I@[(B -> C1)|R] \- C).

a_right((E -> C),PT) <=
  (PT -> ([A1|A] \- C))
  -> (A \- (A1 -> C)).

o_right(1,(C1;C2),PT) <=
  (PT -> (A \- C1))
  -> (A \- (C1; C2)).
o_right(2,(C1 ; C2),PT) <=
  (PT -> (A \- C2))
  -> (A \- (C1; C2)).

o_left((C1;C2),I,PT1, PT2) <=
  (PT1 -> (I@[C1|R] \- C)),
  (PT2 -> (I@[C2|R] \- C))
  -> (I@[(C1; C2)|R] \- C).

v_right((C1,C2),PT1,PT2) <=
  (PT1 -> (A \- C1)),
  (PT2 -> (A \- C2))
  -> (A \- (C1, C2)).
```

```
v_left((C1,C2),I,PT) <=
  (PT -> (I@[C1, C2|R] \- C))
  -> (I@[(C1, C2)|R] \- C).

pi_left((pi X\ C1),I,PT) <=
  inst(X,C1,C2),
  (PT -> (I@[C2|R] \- C))
  -> (I@[(pi X\ C1)|R] \- C).

%%%------------------------
%%% Provisos

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

%%=========================
% 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) <= true_right, v_right(_,PT,PT), a_right(_,PT),
      o_right(_,_,PT), d_right(_,PT).
left(PT) <= false_left(_), v_left(_,_,PT), a_left(_,_,PT,PT),
      o_left(_,_,PT,PT), d_left(_,_,PT), pi_left(_,_,PT).
```

Appendix B

This appendix contains the output from the tracer for the query `arl \\- grey(P) \- false`. We have generated all 9 possible derivations.

```
| ?- arl \\- grey(P) \- false.
+ CALL 1 0 arl \\- grey(_47) \- false l
+ CALL 15 3 arl \\- elephant(_47),(albino_elephant(_47)->false) \- false l
+ CALL 26 6 arl \\- elephant(_47),albino_elephant(_47)->false \- false l
+ CALL 38 9 arl \\- elephant(_47) \- albino_elephant(_47) l
+ CALL 46 12 arl \\- elephant(jumbo) \- true l
+ EXIT 46 12 arl \\- elephant(jumbo) \- true l
+ EXIT 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ CALL 50 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 50 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 50 9 arl \\- elephant(jumbo),false \- false l
+ CALL 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 50 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 50 9 arl \\- elephant(jumbo),false \- false l
+ REDO 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ CALL 78 15 arl \\- true,false \- false l
+ EXIT 78 15 arl \\- true,false \- false l
+ EXIT 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 50 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 50 9 arl \\- elephant(jumbo),false \- false l
+ REDO 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ REDO 78 15 arl \\- true,false \- false l
+ FAIL 78 15 arl \\- true,false \- false l
+ FAIL 64 12 arl \\- albino_elephant(jumbo),false \- false l
+ FAIL 50 9 arl \\- elephant(jumbo),false \- false l
+ REDO 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ REDO 46 12 arl \\- elephant(jumbo) \- true l
+ CALL 105 15 arl \\- albino_elephant(jumbo) \- true l
+ EXIT 105 15 arl \\- albino_elephant(jumbo) \- true l
+ EXIT 46 12 arl \\- elephant(jumbo) \- true l
+ EXIT 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ CALL 109 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 109 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
```

```
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 109 9 arl \\- elephant(jumbo),false \- false l
+ CALL 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 109 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 109 9 arl \\- elephant(jumbo),false \- false l
+ REDO 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ CALL 137 15 arl \\- true,false \- false l
+ EXIT 137 15 arl \\- true,false \- false l
+ EXIT 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 109 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 109 9 arl \\- elephant(jumbo),false \- false l
+ REDO 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ REDO 137 15 arl \\- true,false \- false l
+ FAIL 137 15 arl \\- true,false \- false l
+ FAIL 123 12 arl \\- albino_elephant(jumbo),false \- false l
+ FAIL 109 9 arl \\- elephant(jumbo),false \- false l
+ REDO 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ REDO 46 12 arl \\- elephant(jumbo) \- true l
+ REDO 105 15 arl \\- albino_elephant(jumbo) \- true l
+ CALL 164 18 arl \\- true \- true l
+ EXIT 164 18 arl \\- true \- true l
+ EXIT 105 15 arl \\- albino_elephant(jumbo) \- true l
+ EXIT 46 12 arl \\- elephant(jumbo) \- true l
+ EXIT 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ CALL 168 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 168 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 168 9 arl \\- elephant(jumbo),false \- false l
+ CALL 182 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 182 12 arl \\- albino_elephant(jumbo),false \- false l
+ EXIT 168 9 arl \\- elephant(jumbo),false \- false l
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 arl \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false l
```

```
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false 1
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false 1
+ REDO 168 9 arl \\- elephant(jumbo),false \- false 1
+ REDO 182 12 arl \\- albino_elephant(jumbo),false \- false 1
+ CALL 196 15 arl \\- true,false \- false 1
+ EXIT 196 15 arl \\- true,false \- false 1
+ EXIT 182 12 arl \\- albino_elephant(jumbo),false \- false 1
+ EXIT 168 9 arl \\- elephant(jumbo),false \- false 1
+ EXIT 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false 1
+ EXIT 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false 1
+ EXIT 1 0 arl \\- grey(jumbo) \- false 1

P = jumbo ? ;
+ REDO 1 0 arl \\- grey(jumbo) \- false 1
+ REDO 15 3 arl \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false 1
+ REDO 26 6 arl \\- elephant(jumbo),albino_elephant(jumbo)->false \- false 1
+ REDO 168 9 arl \\- elephant(jumbo),false \- false 1
+ REDO 182 12 arl \\- albino_elephant(jumbo),false \- false 1
+ REDO 196 15 arl \\- true,false \- false 1
+ FAIL 196 15 arl \\- true,false \- false 1
+ FAIL 182 12 arl \\- albino_elephant(jumbo),false \- false 1
+ FAIL 168 9 arl \\- elephant(jumbo),false \- false 1
+ REDO 38 9 arl \\- elephant(jumbo) \- albino_elephant(jumbo) 1
+ REDO 46 12 arl \\- elephant(jumbo) \- true 1
+ REDO 105 15 arl \\- albino_elephant(jumbo) \- true 1
+ REDO 164 18 arl \\- true \- true 1
+ FAIL 164 18 arl \\- true \- true 1
+ FAIL 105 15 arl \\- albino_elephant(jumbo) \- true 1
+ FAIL 46 12 arl \\- elephant(jumbo) \- true 1
+ CALL 232 12 arl \\- true;albino_elephant(clyde) \- albino_elephant(clyde) 1
+ CALL 240 15 arl \\- true;albino_elephant(clyde) \- false 1
+ CALL 253 18 arl \\- true \- false 1
+ FAIL 253 18 arl \\- true \- false 1
+ FAIL 240 15 arl \\- true;albino_elephant(clyde) \- false 1
+ CALL 275 15 arl \\- true \- albino_elephant(clyde) 1
+ CALL 283 18 arl \\- true \- false 1
+ FAIL 283 18 arl \\- true \- false 1
+ FAIL 275 15 arl \\- true \- albino_elephant(clyde) 1
+ FAIL 232 12 arl \\- true;albino_elephant(clyde) \- albino_elephant(clyde) 1
+ CALL 307 12 arl \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) 1
+ CALL 315 15 arl \\- true;albino_elephant(dumbo) \- false 1
+ CALL 328 18 arl \\- true \- false 1
+ FAIL 328 18 arl \\- true \- false 1
+ FAIL 315 15 arl \\- true;albino_elephant(dumbo) \- false 1
+ CALL 350 15 arl \\- true \- albino_elephant(dumbo) 1
+ CALL 358 18 arl \\- true \- false 1
+ FAIL 358 18 arl \\- true \- false 1
+ FAIL 350 15 arl \\- true \- albino_elephant(dumbo) 1
+ FAIL 307 12 arl \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) 1
+ FAIL 38 9 arl \\- elephant(_47) \- albino_elephant(_47) 1
+ CALL 385 9 arl \\- true;albino_elephant(clyde),albino_elephant(clyde)->false \-
false 1
+ CALL 397 12 arl \\- true;albino_elephant(clyde) \- albino_elephant(clyde) 1
+ CALL 405 15 arl \\- true;albino_elephant(clyde) \- false 1
+ CALL 418 18 arl \\- true \- false 1
+ FAIL 418 18 arl \\- true \- false 1
+ FAIL 405 15 arl \\- true;albino_elephant(clyde) \- false 1
+ CALL 440 15 arl \\- true \- albino_elephant(clyde) 1
+ CALL 448 18 arl \\- true \- false 1
+ FAIL 448 18 arl \\- true \- false 1
+ FAIL 440 15 arl \\- true \- albino_elephant(clyde) 1
+ FAIL 397 12 arl \\- true;albino_elephant(clyde) \- albino_elephant(clyde) 1
+ CALL 473 12 arl \\- true,albino_elephant(clyde)->false \- false 1
+ CALL 485 15 arl \\- true \- albino_elephant(clyde) 1
+ CALL 493 18 arl \\- true \- false 1
+ FAIL 493 18 arl \\- true \- false 1
+ FAIL 485 15 arl \\- true \- albino_elephant(clyde) 1
```

```
+ FAIL 473 12 arl \\- true,albino_elephant(clyde)->false \- false l
+ FAIL 385 9 arl \\- true;albino_elephant(clyde),albino_elephant(clyde)->false \-
false l
+ CALL 520 9 arl \\- true;albino_elephant(dumbo),albino_elephant(dumbo)->false \-
false l
+ CALL 532 12 arl \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) l
+ CALL 540 15 arl \\- true;albino_elephant(dumbo) \- false l
+ CALL 553 18 arl \\- true \- false l
+ FAIL 553 18 arl \\- true \- false l
+ FAIL 540 15 arl \\- true;albino_elephant(dumbo) \- false l
+ CALL 575 15 arl \\- true \- albino_elephant(dumbo) l
+ CALL 583 18 arl \\- true \- false l
+ FAIL 583 18 arl \\- true \- false l
+ FAIL 575 15 arl \\- true \- albino_elephant(dumbo) l
+ FAIL 532 12 arl \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) l
+ CALL 608 12 arl \\- true,albino_elephant(dumbo)->false \- false l
+ CALL 620 15 arl \\- true \- albino_elephant(dumbo) l
+ CALL 628 18 arl \\- true \- false l
+ FAIL 628 18 arl \\- true \- false l
+ FAIL 620 15 arl \\- true \- albino_elephant(dumbo) l
+ FAIL 608 12 arl \\- true,albino_elephant(dumbo)->false \- false l
+ FAIL 520 9 arl \\- true;albino_elephant(dumbo),albino_elephant(dumbo)->false \-
false l
+ FAIL 26 6 arl \\- elephant(_47),albino_elephant(_47)->false \- false l
+ FAIL 15 3 arl \\- elephant(_47),(albino_elephant(_47)->false) \- false l
+ FAIL 1 0 arl \\- grey(_47) \- false l

no
| ?-
```

Appendix C

This appendix contains the output from the tracer for the query `es \\- grey(P) \- false.`

```
| ?- es \\- grey(P) \- false.
+ CALL 1 0 es \\- grey(_45) \- false l
+ CALL 16 5 es \\- elephant(_45),(albino_elephant(_45)->false) \- false l
+ CALL 28 10 es \\- elephant(_45),albino_elephant(_45)->false \- false l
+ CALL 41 15 es \\- elephant(_45) \- albino_elephant(_45) l
+ CALL 47 18 es \\- elephant(jumbo) \- true l
+ EXIT 47 18 es \\- elephant(jumbo) \- true l
+ EXIT 41 15 es \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ CALL 54 15 es \\- elephant(jumbo),false \- false l
+ EXIT 54 15 es \\- elephant(jumbo),false \- false l
+ EXIT 28 10 es \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ EXIT 16 5 es \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ EXIT 1 0 es \\- grey(jumbo) \- false l

P = jumbo ? ;
+ REDO 1 0 es \\- grey(jumbo) \- false l
+ REDO 16 5 es \\- elephant(jumbo),(albino_elephant(jumbo)->false) \- false l
+ REDO 28 10 es \\- elephant(jumbo),albino_elephant(jumbo)->false \- false l
+ REDO 54 15 es \\- elephant(jumbo),false \- false l
+ FAIL 54 15 es \\- elephant(jumbo),false \- false l
+ REDO 41 15 es \\- elephant(jumbo) \- albino_elephant(jumbo) l
+ REDO 47 18 es \\- elephant(jumbo) \- true l
+ FAIL 47 18 es \\- elephant(jumbo) \- true l
+ FAIL 41 15 es \\- elephant(_45) \- albino_elephant(_45) l
+ CALL 69 15 es \\- true;albino_elephant(dumbo),albino_elephant(dumbo)->false \-
false l
+ CALL 82 20 es \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) l
+ CALL 88 23 es \\- true;albino_elephant(dumbo) \- false l
+ CALL 102 28 es \\- true \- false l
+ FAIL 102 28 es \\- true \- false l
+ FAIL 88 23 es \\- true;albino_elephant(dumbo) \- false l
+ FAIL 82 20 es \\- true;albino_elephant(dumbo) \- albino_elephant(dumbo) l
+ CALL 125 20 es \\- true,albino_elephant(dumbo)->false \- false l
+ CALL 138 25 es \\- true \- albino_elephant(dumbo) l
+ CALL 144 28 es \\- true \- false l
+ FAIL 144 28 es \\- true \- false l
+ FAIL 138 25 es \\- true \- albino_elephant(dumbo) l
+ FAIL 125 20 es \\- true,albino_elephant(dumbo)->false \- false l
+ FAIL 69 15 es \\- true;albino_elephant(dumbo),albino_elephant(dumbo)->false \-
false l
+ CALL 170 15 es \\- true;albino_elephant(clyde),albino_elephant(clyde)->false \-
false l
+ CALL 183 20 es \\- true;albino_elephant(clyde) \- albino_elephant(clyde) l
+ CALL 189 23 es \\- true;albino_elephant(clyde) \- false l
+ CALL 203 28 es \\- true \- false l
+ FAIL 203 28 es \\- true \- false l
+ FAIL 189 23 es \\- true;albino_elephant(clyde) \- false l
+ FAIL 183 20 es \\- true;albino_elephant(clyde) \- albino_elephant(clyde) l
+ CALL 226 20 es \\- true,albino_elephant(clyde)->false \- false l
+ CALL 239 25 es \\- true \- albino_elephant(clyde) l
+ CALL 245 28 es \\- true \- false l
+ FAIL 245 28 es \\- true \- false l
+ FAIL 239 25 es \\- true \- albino_elephant(clyde) l
+ FAIL 226 20 es \\- true,albino_elephant(clyde)->false \- false l
+ FAIL 170 15 es \\- true;albino_elephant(clyde),albino_elephant(clyde)->false \-
false l
+ FAIL 28 10 es \\- elephant(_45),albino_elephant(_45)->false \- false l
+ FAIL 16 5 es \\- elephant(_45),(albino_elephant(_45)->false) \- false l
+ FAIL 1 0 es \\- grey(_45) \- false l
```

```
no
| ?-
```