Implementational Issues in GCLA:
Compiling Control

by

Martin Aronsson
Implementational Issues in GCLA: Compiling Control

by

Martin Aronsson
Knowledge Based Systems Laboratory; 1993-06-23
Swedish Institute of Computer Science
Box 1263
S - 164 28 Kista
Sweden
e-mail: martin@sics.se

Abstract

The paper describes the basic implementation of GCLA II's control level. The basis of the implementation is a compiling scheme for transforming inference rules and strategies operating on the object level to an interpreter in Prolog, where the inference rules of the control level are coded inline. This is possible since the operational semantics of the control level is deterministic, i.e. the choice of inference rule to apply on a control level goal is determined solely by the parts of the goal. To handle dynamic clauses, a context list, accessible through some new C-functions linked together with the Prolog system. GCLA I and GCLA II are described shortly, followed by a discussion of a Horn clause representation of inference rules versus functions coding inference rules. Then the transformation of inference rules and strategies is described followed by some examples.

Keywords: GCLA, inference rules, interpreters, program transformation
1. Introduction

GCLA is a specification tool for KBS applications that has been under development at SICS for some years, starting off with a one layer language [NGC], which we will refer to as GCLA I, and extended with a control level to guide the execution [Kre92], which we will refer to as GCLA II. GCLA II consists of two levels: the object level which contains the declarative content of the application, and the control level which contains the inference rules and inference strategies that operate on the object level. Both levels of GCLA II are based on the same formalism, Partial inductive definitions [Hal91], but the control level is a restricted version of the full language presented in [HS-H90, HS-H91].

The control level is the basis for the operational behaviour of GCLA II. It describes declaratively the inference rules used to build proofs of object level queries. To describe how GCLA II is implemented is therefore in principle to describe how the control level is implemented, which is the focus of this paper.

The basic idea is to transform the control level definition into Prolog code, with the control level inference rules coded implicitly. This can be done since the choice of inference rule of the control level is deterministic, and only dependent on the control level sequent. We will show how object level inference rules and strategies are compiled into an interpreter and how user defined provisos are transformed into Prolog procedures.

We will often make use of the terminology 'to the right' and 'to the left'. By 'to the left' we mean some term or condition to the left of the derivation symbol (object level, '$\vdash$', or control level, '$\vdash$', will be clear from context), i.e. a member of the antecedent of a sequent. 'On the right' is defined analogously.

The term 'context' is used in two different ways; as a name of an inference rule, initial-context, and as a name on the global environment that dynamically changes during the execution. These two has nothing to do with one another, and there will be no doubt of which of the cases we refer to.

Sicstus Prolog offers the possibility to group code into modules, to separate different parts of the code and avoid name clashes. We have used it to distinguish between GCLA II primitives and user defined provisos, and other user defined Prolog code. To refer to a defined atom in a module, the syntax Module:Atom is used. The GCLA II system and its primitives reside in the gla module, from which some definitions are exported to be visible to the user.

The GCLA II system is distributed together with the Sicstus Prolog system as a library utility.

2. Background

GCLA as presented in [NGC, Aro91c] had problems with managing the search space, and lacked the possibility to guide proofs in the preferred direction. There are several reasons why one wants to guide the proof search, for example to generate solutions in a certain order to avoid loops, and to cut away unwanted answers (e.g. duplicates). A solution was presented in GCLA II [Kre92], which is a system based on two levels: one object level, which is conceptually the same as the program in GCLA I, and a control level which is used to specify the inference rules that operate on the object level. Additionally there is the possibility to form strategies by grouping different rules together and leaving others out, which gives the user the possibility to reduce the search space significantly.
Some of the main goals set up for the control language were that it should be

- deterministic, and not use the whole GCLA I on a meta level to specify the control, in which case we would just move the problem one layer,
- at least expressive enough to be able to code the inference rules of GCLA I, with the possibility to express pruning operations on the proof tree
- modular, i.e. several different control definitions should be applicable to the same object level definition, and vice versa.

The development of the theoretical properties and language specification is due to Per Kreuger, in collaboration with the author, Lars Hallnäs and Lars-Henrik Eriksson, and is described in [Kre92]. Most of the language issues and the properties of the language are also described in [Kre92].

The basis of the control part is the possibility to look at a proof as a functional expression, which when evaluated returns the query sequent. The functions code the inference rules, and the functions map proofs of the premises of the inference rule to the conclusion of the inference rule. When posing a query sequent to the system, we are conducting the evaluation backwards, trying to fill in the missing functional expressions that have the query sequent as value. Since functions can be coded in GCLA [Aro91b], the control level is based on what is needed in order to evaluate functions.

2.1 Syntax

The syntax of both the object level and the control level is the same, except for an append operator ',@', used in the control code, and guards used in object level clauses.

A constant is a term, and so is 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 $\varepsilon$ is a function (term constructor) of arity $n$, then $\varepsilon(A_1, ..., A_n)$ is a term. All terms are conditions, and if $C_1$ and $C_2$ are conditions, then so are $(C_1 \rightarrow C_2)$, $(C_1, C_2)$, $(C_1; C_2)$. true and false are conditions, and if $x$ is an variable and $c$ a condition, then $(\pi x \downarrow c)$ is a condition, where $X$ occurs bound in the condition $C$.

A constraint is a term of the form $T_1 |\prec T_2$. If $A_1, ..., A_n$ are constraints, then $\{A_1, ..., A_n\}$ is a guard. Currently, the implementation cannot handle guards in control level clauses.

An atom is a term which is not a variable. If $a$ is an atom, $c$ a condition, $g$ a guard, then $a@g <= c$ is a clause. If $g$ is empty we will use $a <= c$ as a shorthand for $a\emptyset() <= c$. We will refer to $a$ as the head of the clause, $g$ as the guard of the clause, and $c$ as the body of the clause. Often we will use $H$ to denote the head and $B$ to denote the body of a clause.

An ordered set of clauses forms a definition $D$.

Object level sequents are coded by the structure $(\text{AntecedentList} \prec C)$, where $C$ is a condition, and $\text{AntecedentList}$ is a list of the form $[C_1, ..., C_n]$ where $C_1, ..., C_n$ are conditions. On the $\text{AntecedentList}$ an append operator, '@', is defined as: $List_1@List_2 = List_3$. It is conveniently used to code object level antecedent lists in inference rules and strategies, and is processed at unification time. For example, $[1]@[1,2], [1]@[2]$ and $[1,2]@[1]$ all equal $[1,2]$.'@' can be indeterministic, for example there are three different solutions to $A@B = [1,2]$, namely the three cases presented above.

Control level sequents have the structure $(\text{AntecedentList} \\prec C)$, where $\text{AntecedentList}$ may at most contain one condition, and $C$ is a condition.
2.2 Operational Semantics of GCLA I

To be able to talk about what kind of object level inference rules we want to express, we give the inference rules of GCLA I here. The interested reader is referred to [NGC] for a treatment of GCLA I.

We will use $\theta$, $\xi$, $\zeta$ and $\sigma$ to denote substitutions and $\varepsilon$ for the empty substitution, $\emptyset$ to denote the empty list or set, and $\cdot$ for concatenation of an element to a list. $\Sigma$ is used to denote a list of yet unproved goal sequents. We will use $'\vdash'$ as the derivation symbol for the object level, to distinguish it from the control level derivation symbol $'\\vdash'$ introduced later, and give the inference rules for GCLA I as one example of inference rules operating on the object level.

**Initial context**

$$\langle \emptyset, \varepsilon \rangle$$

**Truth**

$$\langle \Sigma, \emptyset \rangle$$

$$\langle (A \vDash T) \cdot \Sigma, \emptyset \rangle$$

for any $\theta$

**Falsity**

$$\langle \Sigma, \emptyset \rangle$$

$$\langle (A_1 \vDash [L] A_2) \vDash C) \cdot \Sigma, \emptyset \rangle$$

for any $\theta$

**Initial sequent**

$$\langle \Sigma \sigma, \emptyset \rangle$$

$$\langle (A_1 \vDash T_1 \vDash A_2) \vDash T_2) \cdot \Sigma, \emptyset \sigma \rangle$$

if $T_1$ and $T_2$ are terms, there exists a $\sigma$ such that $T_1 \sigma = T_2 \sigma$. Also referred to as **axiom**.

**Arrow right**

$$\langle (A_1 \vDash A \vDash C) \cdot \Sigma, \emptyset \rangle$$

$$\langle (A \vDash (A_1 \rightarrow C)) \cdot \Sigma, \emptyset \rangle$$

**Arrow left**

$$\langle (A_1 \vDash A_2) \vDash A) \cdot (A_1 \vDash C) \vDash A_2) \vDash C) \cdot \Sigma, \emptyset \rangle$$

$$\langle (A_1 \vDash (A \rightarrow C_1) \vDash A_2) \vDash C) \cdot \Sigma, \emptyset \rangle$$

**Product right**

$$\langle (A \vDash C_1) \cdot (A \vDash C_2) \cdot \Sigma, \emptyset \rangle$$

$$\langle (A \vDash (C_1, C_2)) \cdot \Sigma, \emptyset \rangle$$

Also referred to as **vector-right**.

**Product left**

$$\langle (A_1 \vDash [C_1, C_2] \vDash A_2) \vDash C) \cdot \Sigma, \emptyset \rangle$$

$$\langle (A_1 \vDash (C_1, C_2) \vDash A_2) \vDash C) \cdot \Sigma, \emptyset \rangle$$

Also referred to as **vector-left**.
\(\langle (A \ \vdash \ c_1) \cdot \Sigma, \theta \rangle\)
\(\langle (A \ \vdash \ (c_1 : c_2)) \cdot \Sigma, \theta \rangle\)
\(\langle (A \ \vdash \ c_2) \cdot \Sigma, \theta \rangle\)
\(\langle (A \ \vdash \ (c_1 : c_2)) \cdot \Sigma, \theta \rangle\)

Also referred to as or-right.

\(\langle (A_1 \theta [c_1 | a_2] \ \vdash \ c) \cdot (A_1 \theta [c_2 | a_2] \ \vdash \ c) \cdot \Sigma, \theta \rangle\)
\(\langle (A_1 \theta [(c_1 : c_2) | a_2] \ \vdash \ c) \cdot \Sigma, \theta \rangle\)

Also referred to as or-left.

\(\langle (A_1 \theta [(C_1 \zeta) | a_2] \ \vdash \ c) \cdot \Sigma, \theta \rangle\)
\(\langle (A_1 \theta [(\Pi x : c_1) | a_2] \ \vdash \ c) \cdot \Sigma, \theta \rangle\)

where \(\zeta\) substitutes a unique new variable for all free occurrences of x in \(c_1\).

\(\langle (A \ \vdash \ c_1 \sigma \zeta) \cdot \Sigma \sigma, \theta \sigma \rangle\)
\(\langle (A \ \vdash \ a) \cdot \Sigma, \theta \sigma \rangle\)

if \(a\) is an atom, \((a_1 \leftarrow c_1)\) is in the definition, \(\sigma = mgu (a, a_1)\) and \(\zeta\) is a substitution that assigns unique new variables for all free variables in \(c_1\) that are not also free in \(a_1\). All instances of the rule will be tried by backtracking over the clauses \((a_1 \leftarrow c_1)\) in the order in which they appear in the definition.

\(\langle (A_1 \theta [D(a) \sigma | a_2] \ \vdash \ c) \cdot \Sigma \sigma, \theta \sigma \rangle\)
\(\langle (A_1 \theta [a | a_2] \ \vdash \ c) \cdot \Sigma, \theta \sigma \rangle\)

if \(\sigma\) is an \(a\)-sufficient substitution with respect to the definition and \(D(a)\) is the definiens operation as explained in [Aro93a]. All instances of the rule will be tried by backtracking over the substitutions generated by the algorithm for determining \(a\)-sufficient substitutions.

To guide the search among the possible applicable inference rules in GCLA I, there were some global parameters that the user could set to get different search behaviour, typically on which side of the turnstile the search for an applicable inference rule should be performed, if some rule should not be considered etc. Those global parameters were set for the whole execution, and turned out to be too crude and hard to use. There was a need for a much more accurate and precise way of describing which search behaviour the system should have, based on the sequent's structure and what kind of behaviour one would like the system to have. Ultimately, the user should have the possibility to specify the search behaviour in some kind of operational language, which should specify different search strategies, and perhaps also have the possibility to write his own inference rules. From these thoughts, GCLA II emerged. We present some basic thoughts in the next section.
2.3 Interpreters and Inference Rules

Since the search for a proof of a query is conducted backwards, one natural way to represent the inference rules of GCLA in section 2.2 would be as Horn clauses. Such an implementation is presented in the left column of the table below. Another natural way of representing the inference rules is by representing the proof of a query as a functional expression, which when evaluated gives the query sequent as value. Proof search is conducted by evaluating the functions 'backwards', i.e. trying to fill in the functional expression which has the given query sequent as its value. Such functions are presented below in the right column, in the style in which functions are expressed in GCLA.

<table>
<thead>
<tr>
<th>(A - true).</th>
<th>true_right &lt;= (  - true).</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A[false] - C).</td>
<td>false_left(I) &lt;=</td>
</tr>
<tr>
<td>(I[false] - )</td>
<td></td>
</tr>
<tr>
<td>(A - (A1 - C)) &lt;= (PT - ((A1[A] - C)) - ) - (A - (A1 - C)).</td>
<td></td>
</tr>
<tr>
<td>(I[A - C] - A), (I[C - C] - C).</td>
<td></td>
</tr>
<tr>
<td>a_right((A - C), PT) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT - ((A1[A] - C)) - ) - (A - (A1 - C)).</td>
<td></td>
</tr>
<tr>
<td>(I[C - C] - C).</td>
<td></td>
</tr>
<tr>
<td>a_left(((A - C1),I,PT1,PT2) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT1 - (I[A - C1] - A)), (PT2 - (I[C - C] - C)) - (I[A - C1] - C)).</td>
<td></td>
</tr>
<tr>
<td>(A - (C1,C2)) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(A - C1), (A - C2).</td>
<td></td>
</tr>
<tr>
<td>v_right((C1,C2),PT1,PT2) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT1 - (I[C1,C2] - C)) - (I[C1,C2] - C)).</td>
<td></td>
</tr>
<tr>
<td>(I[C1,C2] - C).</td>
<td></td>
</tr>
<tr>
<td>v_left((C1,C2),I,PT) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT - ((I[C1,C2] - C) - C)) - (I[C1,C2] - C)).</td>
<td></td>
</tr>
<tr>
<td>(A - (C1,C2)) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(A - C1); (A - C2).</td>
<td></td>
</tr>
<tr>
<td>o_right((C1 ; C2),PT1,PT2) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT1 - (A - C1)); (PT2 - (A - C2)) - (A - (C1 ; C2)).</td>
<td></td>
</tr>
<tr>
<td>(I[C1; C2] - C).</td>
<td></td>
</tr>
<tr>
<td>o_left((C1 ; C2),I,PT1,PT2) &lt;=</td>
<td></td>
</tr>
<tr>
<td>(PT1 - (I[C1; C2] - C)) - (I[C1; C2] - C)).</td>
<td></td>
</tr>
<tr>
<td>(I[C1; C2] - C).</td>
<td></td>
</tr>
<tr>
<td>(I[I; T] - T2) &lt;=</td>
<td></td>
</tr>
<tr>
<td>atom(T1), atom(T2), unify(T1,T2).</td>
<td></td>
</tr>
<tr>
<td>axiom(T1,T2,I) &lt;=</td>
<td></td>
</tr>
<tr>
<td>term(T1), term(T2), unify(T1,T2) - (I[I; T] - T2).</td>
<td></td>
</tr>
<tr>
<td>(A - T) &lt;=</td>
<td></td>
</tr>
<tr>
<td>atom(T), clause(T,B), (A - B).</td>
<td></td>
</tr>
<tr>
<td>d_right(T,B) &lt;=</td>
<td></td>
</tr>
<tr>
<td>atom(T), clause(T,B), (PT - (A - B)) - (A - C).</td>
<td></td>
</tr>
<tr>
<td>(I[T; R] - C).</td>
<td></td>
</tr>
<tr>
<td>d_left(T,I,PT) &lt;=</td>
<td></td>
</tr>
<tr>
<td>atom(T), definiens(T,D), (I[D; R] - C).</td>
<td></td>
</tr>
</tbody>
</table>

As can be seen there is a close correspondence between the two versions, the largest difference being that the rules are given by name in the functional version, and that the PT arguments of the functions contain the functional expressions that prove the premises of the inference rules. Such expressions are called proofterms. The proofterm gives the whole proof of a sequent in the functional version. If we are satisfied with having the rules given without names and not interested in the proof term, the Horn clause version is adequate; it is concise and easy to understand and conforms to the mainstream of logic programming. In principle, such an interpreter formed the basis of the implementation of
GCLA I. However, if we want to conduct search of possible ways to prove a sequent on the basis on the sequent's content, it is much easier if there are explicit names for the rules.

The Horn clause version above could be extended with the names, which results in the interpreter below:

\[
\begin{align*}
\text{inter}(\text{true}_\text{right}, (A \leftarrow \text{true})). \\
\text{inter}(\text{false}_\text{left}(I), (I \not\in (\text{false}_R \leftarrow C)). \\
\text{inter}(\text{a}_\text{right}(A \leftarrow C) \land PT), (A \leftarrow (A \leftarrow C)) &\Leftarrow \text{inter}(PT, (A \not\in \text{A} \leftarrow C)). \\
\text{inter}(\text{a}_\text{left}(A \leftarrow C) \land I, PT_1, PT_2), (I \not\in (A \leftarrow C) \land R \leftarrow C1)) &\Leftarrow \text{inter}(PT_1, (I \not\in A \leftarrow A1)), \\
&\text{inter}(PT_2, (I \not\in (A \not\in C) \land R \leftarrow C1)). \\
\text{inter}(\text{v}_\text{right}(C, C_1, C_2) \land PT_1, PT_2), (A \leftarrow (C, C_2))) &\Leftarrow \text{inter}(PT_1, (A \leftarrow C)), \\
&\text{inter}(PT_2, (A \leftarrow C_2)). \\
\text{inter}(\text{v}_\text{left}(C, C_1, C_2, I, PT), (I \not\in (C, C_2) \land R \leftarrow C)) &\Leftarrow \text{inter}(PT, (I \not\in (C, C_2) \land R \leftarrow C)). \\
\text{inter}(\text{o}_\text{right}(C, C_1, C_2) \land PT_1, PT_2), (A \leftarrow (C, C_2))) &\Leftarrow \\
&\text{inter}(PT_1, (A \leftarrow C)), \\
&\text{inter}(PT_2, (A \leftarrow C_2)). \\
\text{inter}(\text{o}_\text{left}(C, C_1, C_2, I, PT_1, PT_2), (I \not\in (C, C_2) \land R \leftarrow C)) &\Leftarrow \text{inter}(PT_1, (I \not\in C_1 \land R \leftarrow C)), \\
&\text{inter}(PT_2, (I \not\in C_2 \land R \leftarrow C)). \\
\text{inter}(\text{axiom}(T_1, T_2, I), (I \not\in (T_1 \land R \leftarrow T_2)) &\Leftarrow \\
&\text{atom}(T_1), \\
&\text{atom}(T_2), \\
&\text{unify}(T_1, T_2). \\
\text{inter}(\text{d}_\text{right}(T, PT), (A \leftarrow T)) &\Leftarrow \text{atom}(T), \\
&\text{clause}(T, B), \\
&\text{inter}(PT, (A \leftarrow B)). \\
\text{inter}(\text{d}_\text{left}(T, I, PT), (I \not\in (T \land R \leftarrow C)) &\Leftarrow \text{atom}(T), \\
&\text{def}_\text{Tuffix}(T, D), \\
&\text{inter}(PT, (I \not\in (D \land R \leftarrow C)).
\end{align*}
\]

This is in principle an interpreter for function evaluation, where the first argument of \text{inter/2} is the functional term, and the second argument is the result, and the body of each clause contains calls to determine an evaluated value for the first argument. In principle, this is the way GCLA II's control inference rules are implemented as we will see later. But the picture is not complete. One of the main aims of the control level was that the user must have the ability to cut away branches of the proof tree that he is not interested in, and that he can specify how the search for a proof is conducted. The three types of implementation presented above do not give the user such facilities. Therefore, the control level of GCLA must also enable the user to group inference rules together into (named) useful entities, which we will call a strategy. It is partly for this reason that we demand that the rules should be referred to by name.

The most basic search behaviour, coding the default behaviour of GCLA I, is presented in the table below, the Horn clause version in the left column and the functional version in the right column.
As can be seen the second argument of the Horn clause interpreter, the object level sequent, is just passed around, while it does not appear in the functional version. The functional version is a more concise one, and it is also quite natural to create hierarchies of strategies by functions, while for relations this is not as natural, since systems of relations are by nature flat. Since the Horn clause version in principle implements an interpreter for the functional expressions of the right column, why not let the user specify the functions, and then compile them into a Horn clause representation, as above? This is the basic idea for the implementation of the control level. The functional version is more concise and lets the user concentrate on how he wants to conduct the proof, and the functions are then compiled into an interpreter that implements the evaluation of the strategies and inference rules.

Another interesting modularity of the control level, when a functional approach is chosen, is the possibility to restrict rules by adding new clauses to the control definition. If there are more than one clause defining a function, we have to prove that all the bodies of the defining clauses give the same result. If they do, we have proved that the result holds according to the given definition, and is therefore a valid consequence of the definition, since it does not matter which one of the clauses we take, and therefore we have proved that it holds according to all possible clauses of the definition. Since the functions are executed backwards when used for proof-search, the result of some yet not known proofterm is given, but we should find which proofterm gives the result. If we add clauses that give the same result as some inference rule, we have to pay attention to the added clauses as well.

Suppose that we want to restrict the applicability of the `d_right/2` rule defined by

\[
\begin{align*}
    d_{\text{right}}(C, PT) & \leftarrow \\
    \text{atom}(C), \\
    \text{clause}(C, B), \\
    (PT \rightarrow (A \leftarrow B)) \\
    \rightarrow (A \leftarrow C).
\end{align*}
\]

By adding a new clause

\[
d_{\text{right}}(C, PT) \leftarrow \text{restriction}(A, C) \rightarrow (A \leftarrow C).
\]

the applicability of the `d_right` rule is restricted to those cases where `restriction` holds, since both defining clauses of `d_right/2` must be taken into account to produce the result `(A \leftarrow C)`. The possibility to add restricting clauses is not limited to rules, but applies to strategies as well.
2.4 Operational semantics of GCLA II

The operational semantics is the basis for the implementation of the runtime system. The operational semantics presented herein is a linear calculus referred to as the DOLD calculus in [Kre92], which has been the basis for the material presented in this section. It is a linear calculus where the leftmost sequent of a list of goal sequents is considered in each step. Later in this paper, when discussing the properties and choice of rules etc, we will consider only the leftmost sequent, and leave the other goal sequent as implicitly there, since solving each goal sequent is only dependent on that sequent itself.

The inference rules of GCLA II's control level, which give the operational semantics for evaluating proofterms, differ from the inference rules of GCLA I in several ways: 1) the sequents of the object level occur as terms of the sequents of the control level, 2) the sequents of the control level may at most have one element in the antecedent, 3) there are two substitutions to keep track of; the object level substitutions and the control level substitutions, 4) some of the consequents are primitive operations that operate on object level terms. As mentioned before, we will use the derivation symbol \( \bot \) for the control level sequents, to distinguish them from the object level sequents \( \bot \).

A goal in the operational semantics of GCLA II is a triple \( \langle \text{Sequence}, \text{ControlSubst}, \text{ObjectSubst} \rangle \), where Sequence is a sequence of control level sequents to be solved, ControlSubst determines substitutions for the control level and ObjectSubst determines substitutions for the object level. Note that ObjectSubst is only involved together with the three primitive provisos clause/2, definiens/2 and unify/2. There are other primitives explained in section 3.2, which are not presented here in the operational semantics, but are handled analogously as unify/2.

**Initial context**
\[ \langle \emptyset, \{ \}, \{ \} \rangle \]
**termination**

**Truth**
\[ \langle \Sigma, \theta, \xi \rangle \]
\[ \langle (\bot \implies \top) \cdot \Sigma, \theta, \xi \rangle \]
for any \( \theta \) and \( \xi \)

**Falsity**
\[ \langle \Sigma, \theta, \xi \rangle \]
\[ \langle (\bot \bot) \cdot \Sigma, \theta, \xi \rangle \]
for any \( \theta \) and \( \xi \)

**Initial sequent**
\[ \langle \Sigma \cdot \sigma, \theta, \xi \rangle \]
\[ \langle ((\bot \implies \bot) \implies (\bot \implies \bot)) \cdot \Sigma, \theta \cdot \sigma, \xi \rangle \]
if there exists a \( \sigma \) such that \( (A \implies C) \cdot \sigma = (A \implies C) \cdot \sigma \). Also referred to as axiom.

**Arrow right**
\[ \langle (A \implies \bot) \cdot \Sigma, \theta, \xi \rangle \]
\[ \langle (\bot \implies (A \implies C)) \cdot \Sigma, \theta, \xi \rangle \]

**Arrow left**
\[ \langle (C \implies \bot) \cdot (\bot \implies A) \cdot \Sigma, \theta, \xi \rangle \]
\[ \langle ((A \implies C \implies C) \cdot (\bot \implies A)) \cdot \Sigma, \theta, \xi \rangle \]
Product right
\[
\langle\langle\langle - c_1 \rangle - (\langle\langle - c_2 \rangle) \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle\langle - (c_1, c_2) \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
Also referred to as \textit{vector-right}.

Product left
\[
\langle\langle c_3 \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle (c_1, c_2) \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle c_2 \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle (c_1, c_2) \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
Also referred to as \textit{vector-left}.

Sum-right
\[
\langle\langle\langle - c_1 \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle\langle - (c_1; c_2) \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle\langle - c_2 \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle\langle - (c_1; c_2) \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
Also referred to as \textit{or-right}.

Sum-left
\[
\langle\langle c_1 \langle\langle - c \rangle \cdot (c_2 \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle (c_1; c_2) \langle\langle - c \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
Also referred to as \textit{or-left}.

Pi-left
\[
\langle\langle\langle\langle\langle c_1 \langle\langle - \circ \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\langle\langle\langle\langle\langle (\Pi\circ_{\langle\langle - \circ \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
where \(\xi\) substitutes a unique new variable for all free occurrences of \(x\) in \(c_1\).

Definition-right
\[
\langle\langle\langle\langle\langle - c_1 \circ \xi \rangle \cdot \Sigma \circ, \quad \theta, \xi \rangle
\langle\langle\langle\langle\langle - a \circ \xi \rangle \cdot \Sigma, \quad \theta, \xi \rangle
\]
if \(a\) is an atom, and \((a_1 \Leftarrow c_1)\) is in the proviso definition and \(\sigma = mgu(a, a_1)\) and \(\xi\) is a substitution that assigns unique new variables for all free variables in \(c_1\) that are not also free in \(a_1\). All instances of the rule will be tried by backtracking over the clauses \((a_1 \Leftarrow c_2)\) in the order in which they appear in the definition. Only meta variables are bound (see section 6 for a discussion of variables).

Definition-left
\[
\langle\langle\langle\langle\langle D(a \circ) \langle\langle - \circ \rangle \cdot \Sigma \circ, \quad \theta, \xi \rangle
\langle\langle\langle\langle\langle (a \langle\langle - \circ \rangle \cdot \Sigma, \quad \theta \circ, \xi \rangle
\]
if \(a\) is an atom, \(\sigma\) is an \(a\)-sufficient substitution with respect to the control definition (i.e. rules, strategies and provisos). All instances
of the rule will be tried by backtracking over the substitutions generated by the algorithm for determining \(a\)-sufficient substitutions. Note that \(\sigma\) binds only meta variables (see section 6 for a discussion of variables)

\[
\langle \Sigma \rho, \theta, \xi \rangle \\
\langle \langle \neg \text{ unify}(T_1, T_2) \rangle \Sigma, \theta, \xi \rho \rangle
\]

if \(T_1\) and \(T_2\) are object level terms, and \(\rho = mgu(T_1, T_2)\)

\[
\langle \Sigma \rho, \theta, \xi \rangle \\
\langle \langle \neg \text{ clause}(a, c_1 \xi) \rangle \Sigma, \theta, \xi \rho \rangle
\]

if \(a\) is an object level atom, \((a_1 <= c_1) \in D, \rho = mgu(a, a_1)\) and \(\xi\) is a substitution that assigns new variables to the free variables in \(c_1\) that are not also free in \(a_1\). All instances of this rule will be tried by backtracking over the clauses \((a_1 <= c_1)\) in the order in which they appear in the object level definition \(D\).

\[
\langle \Sigma \rho, \theta, \xi \rangle \\
\langle \langle \neg \text{ definens}(a, D(a\rho)) \rangle \Sigma, \theta, \xi \rho \rangle
\]

if \(a\) is an object level atom, and \(\rho\) is an \(a\)-sufficient substitution with respect to the object level definition \(D\), and \(D\) is the definens operation. All instances of this rule will be tried by backtracking over the substitutions generated by the algorithm for determining \(a\)-sufficient substitutions. The empty definens is identified with the symbol \textit{false}.

The inference rules presented above are exactly those needed to evaluate functions and to interpret negated atoms in GCLA. Note that the rule \textit{Product-left} differs from the one given for GCLA I. The \textit{Product-left} rule of GCLA II is the pure dual to the rule \textit{Product-right}, but with one contraction applied on the product, it is easy to get the \textit{Product-left} rule of GCLA I. The pure \textit{Product-left} rule of GCLA II gives the desired behaviour with only one element in the antecedent, and gives a nice way of implementing choice of possible proofs, as is shown in [Aro92, Kre92].

2.5 Properties for Deterministic Execution

One of the main aims of the control level was to make it deterministic, which is achieved by imposing some restrictions on the possible control level sequents:

- There can only be one element in the antecedent. This restriction rules out the expression \(\neg (A_1 \rightarrow (A_2 \rightarrow C))\).
- The object level sequent is defined in a circular way, i.e. it has the definition

\[(A \not\rightarrow B) \leq (A \not\rightarrow B)\]

which means that there is no point in applying the rules \textit{Definition-right} and \textit{Definition-left} to object level sequents, which implies that the only way to satisfy a control level sequent when an object level sequent is present, is for the rule

\[1\] These three points are based on [Kre92]
Initial-sequent to become applicable sooner or later. Another 'functional' way of expressing the circular clause above is to say that object level sequents evaluates to themselves.

- Apply rules that operate on the consequent only when the antecedent is empty.

With these restrictions there is only one rule of the operational semantics that is applicable for each possible control level sequent. Depending on the control level sequent, there are four mutually exclusive cases:

1) The control level sequent is an instance of $(A_1 \leftarrow C_1) \ \leftarrow (A \leftarrow C)$, in which case the rule Initial-sequent is used to match the sequent to the right of '$\leftarrow$' with the sequent to the left of '$\leftarrow$'. We use matching since the sequent to the right is the object level goal, with object level variables, and the sequent to the left is the result of some inference rule, with control level variables (see section 6 for a discussion on levels of variables).

2) The control level sequent has the form $(A \ \leftarrow (B \ \leftarrow C))$: then the only possible choice is to apply the appropriate rule on the left side (they are all mutually exclusive).

3) The control level sequent has the form $(A \ \leftarrow \text{false})$: again, the only applicable rules operate on the left side, and there is just one possible to choose depending on $A$.

4) If none of the above cases is applicable, then the execution must continue with some rule operating on the right side, since then the antecedent is empty, or it will fail and backtrack.

Put together this means that it is possible to design a runtime system that implicitly codes the different rules of the operational semantics without having to interpret the control level sequents. This is due to the fact that the appropriate control level inference rule can be determined solely by looking at the atoms and conditions of the control level clauses, and reason about where they will occur in the control level sequent.

3. The Runtime System

The runtime system is implemented in Sicstus Prolog. The top level of the GCLA II system is the same as Sicstus’ own top level, from which the GCLA II system is called by queries containing the control level derivation symbol '$\leftarrow$'.

In principle, the runtime system consists of the top level definition of '$\leftarrow$', which calls the compiler to transform the given query into the internal representation, and then calls a rule interpreter. The interpreter is created by the compiler when a control level file is loaded. Beside the top level primitive '$\leftarrow$', some other primitives have been added to the Prolog primitives.

3.1 Guards

'$\leftarrow/2$ is used in guards of object level clauses. The semantics of '$\leftarrow/2$ is: $T_1 \ \leftarrow T_2$ is satisfied if $T_1$ is not an instance of $T_2$, i.e. there does not exist a substitution $\sigma$ such that for all variables in $T_2$, $T_1 \sigma = T_2 \sigma$.

'$\leftarrow/2$ is implemented using the coroutining primitive disjunctive_freeze/2. disjunctive_freeze(Vars, Call) suspends on the variables in Vars, until some variable in Vars get instantiated to a nonvariable term, or two variables in Vars become equal. Then Call is executed and removed. The following routine implements '$\leftarrow/2$:
\( \text{op}(700, \text{xfx}, (', \text{'} = ') \text{').} \)

\( \text{\langle X, Y \rangle := \langle + (X = Y), ! \rangle. } \) % (i)
\( \text{\langle X, Y \rangle := } \)
\( \quad \text{term\_variables(X, L), } \) % Find variables in X
\( \quad (L \text{\langle = [}\}
\( \quad \rightarrow \text{disjunctive\_freeze(L, fail), X=Y, !, fail ; } \) % (ii)
\( \quad \text{disjunctive\_freeze(L, \langle = (X, Y) \rangle) ; fail) } \) % (iii)

The three cases given above are explained as:

(i) The two terms \( X \) and \( Y \) are unequal due to some unequal ground terms in \( X \) and \( Y \), and can therefore never be made equal, and hence \( \equal{\text{\langle}} / 2 \) succeeds.

(ii) If \( X \) and \( Y \) can be made equal without binding any free variables (which are in the list \( L \)), i.e. variables that may be bound by the rest of the execution, then \( X \) is an instance of \( Y \), and hence \( \equal{\text{\langle}} / 2 \) can never be satisfied, and therefore \( \equal{\text{\langle}} / 2 \) fails.

(iii) If neither (i) nor (ii) succeeds, then \( \equal{\text{\langle}} / 2 \) may hold, and therefore \( \equal{\text{\langle}} / 2 \) is suspended on the variables in \( L \), waiting for more information.

3.2 Primitive Provisos and Their Implementation

There are a couple of primitive provisos defined by the GCLA system. The basic ones are explained below, together with their implementation in Sicstus Prolog.

- \text{add(R)}, \text{rem(R)}
  
\text{add/1} and \text{rem/1} enables the user to alter the object level definition. \text{add(R)} adds the clause \( R \) to the object level definition and \text{rem(R)} removes all clauses that are instances of \( R \) from the object level definition. \( R \) is treated as if it were read from an object level definition file.

Since the compilation of the definiens operation and the precomputed \( A \)-sufficient substitutions is done at loading time (see \cite{Aro93a}), the user must declare atoms whose clauses are subject to dynamic changes of the program, by the directive \text{dynamic\_term}(Functor/Arity), where \text{Functor} is the principal functor of the atom and \text{Arity} is the arity of the atom. When an atom is declared as dynamic, there will be no computation of \( A \)-sufficient substitutions, or forming of definiens, at compile time, and no indexing of the clause is made, it is simply stored as it is in the definition database.

Upon completion of the query, or when backtracking over \text{add/1} and \text{rem/1}, the operation is undone, i.e. in the case of \text{add/1} the clause \( R \) is removed from the definition, and in the case of \text{rem/1} the clauses that were removed are added to the definition again.

\text{rem/1} does not actually remove clauses from the definition, to prevent accidental loss of loaded clauses in case of interruption, instead the unique identifier of the object level clause is used. A structure \text{rem(Unid\_ID)} is concatenated to the context list, which is checked by the primitives \text{gclause/3} and \text{dp/4}. More about contexts in section 3.3.

To get better performance, only those clauses whose heads have the same main functor as in the given clause \( R \) are considered in \text{rem/1}.

The user provisos \text{add/1} and \text{rem/1} uses the low level primitives \$\text{add/1} and \$\text{rem/1}, presented below:
$add(R) :-
    parse(R, Head :- Body, Gu),
    get_unique_id(X1),
    copy_term((Head, Body, Gu), (H1, B1, Gu1)),
    call_residue((call(Gu1),
      assertz((gclae_base(H1, B1, X1, dyn))),_),
    !,
    undo(retract((gclae_base(_,_,X1,dyn))))).

$rem(R) :-
    parse(R, Head :- Body, Gu),
    copy_term((Head, Body, Gu), (H1, B1, Gu1)),
    call_residue((call(Gu1),
      remove_clauses((Head1 :- Body1))),_),
    !.

remove_clauses((He :- Bo)) :-
    gfunctor(He, F, A),
    functor(Hel, F, A), % get copy of main term
    gclae_base(Hel, Body, N, _), % find clause
    \+ in_context((rem(N))), % not removed before
    \+ instanced((Hel, Body), (He, Bo)), % clause found is in
    !, % stance of input
    add_to_context(rem(N)), % Add N to context
    remove_clauses((He :- Bo)) % add to list, recurse
    remove_clauses(_) :- !. % All instances found

* definiens(A, B)*
Succeeds if the there exists an A-sufficient substitution and a corresponding
definiens of the term A, where B is the definiens of A (B is a ';' separated structure of the bodies of the definiens). There can be more than one A-sufficient substitution for an atom A. This primitive does not check whether A is an atom or not, which is left to the user.

The implementation of the definiens operation together with the generation of an A-sufficient substitution is extensively discussed in [Aro93a]. There are a couple of low level primitives. All clauses whose heads are known not to be dynamic (i.e. not altered by def/l and rem/l, in which case the user can declare them as dynamic) are preprocessed at loading time to determine the possible A-sufficient substitutions and corresponding definiens in the manner of [Aro93a], and are handled through the relation gclae_base2(Head, Definiens). Dynamic clauses are handled with the low level routine clauses(Head, Definiens), where the A-sufficient substitution is calculated at runtime, and dp(Head, Definiens) determines which of the two cases the current atom belongs to, i.e. if the atom is declared as dynamic or not.

* unify(T1, T2)*
Unifies the object level terms T1 and T2. The implementation is easy:

unify(X, X).

* clause(H, B)*
Succeeds if H1 <= B1 is an object level clause in the current object level definition, and there exists a unifier of (H, B) and (H1, B1). (Note that a possible guard contains only constraints on variables in H1, and is executed when H and H1 are unified). If H contains an undefined atom, B is bound to false.gclause/2 does not check whether H is an atom or not.

The implementation is straightforward:
gclause(H,B) :-
  if((gcla_base(H,B,Num,P),
      not_removed_in_context(Num),
      true,
      B = false).

where gcla_base/4 holds the object level definition: H is the head, B is the body, Num is a unique identifier and P is a flag determining whether the clause is static or dynamic (i.e. asserted by the proviso def/3).

• not/1 is true if Call is proven false, i.e. not/1 could be thought of as having the GCLA clause not(Call) <= Call -> false. in the definition. Negation in GCLA is discussed in [Aro92].

not/1 is implemented as a small interpreter. The interesting thing about this interpreter is that it implements a GCLA I interpreter (i.e. the basic inference rules) except the axiom rule or any rule to the right of the turnstile, which cannot be used when the consequence of the sequent is the condition false. In principle it is a dual to a Prolog meta interpreter, here presented in a slightly cleaned-up version:

not(true) :- !, fail.
not(false) :- !.
not((A,B)) :- !,
   (not(A) ; not(B)).
not((A,B)) :- !,
   not(A),
   not(B).
not(gnot(A)) :- !,
call(A). % Back to positive call
not(A) :-
   system_defined(A),!
   \+ call(A). % Not user defined
not(A) :-
   clauses_proviso(A,B),
   not(B). % Definiens of proviso A

• atom(T)
This primitive checks whether the term T is an object level atom. atom(T) is replaced at compile time by the new primitive gatom(T) with the following definition:

gatom(P) :-
  nonvar(P),
  functor(P,F,A),
  \+ constructor(F,A). % Condition structures

where constructor/2 holds the different condition constructors (e.g. >/2, ;/2 etc).

• term(T)
term(T) checks whether the term T is an object level term. It is implemented as

term(P) :-
  (var(P)
   ; functor(P,F,A),
   \+ constructor(F,A)),!.
    % Condition structures
• functor(T,F,A)
  This primitive gets the main functor F and arity A of an object level term T. Note that T is not allowed to be an object level variable. At compile time all occurrences of functor/3 are replaced by the new primitive gfunctor/3 with the following implementation:

  gfunctor(T,F,A) :- nonvar(T), functor(T,F,A).

• append(List1, List2, List3)
  List1 appended to List2 yields List3. The append operator 'g' of the premises of a rule is compiled into this predicate, which is the ordinary append.

• append1(List3, List2, List1)
  List1 appended to List2 yields List3. The append operator 'g' of conclusion sequents of a rule or strategy is compiled into this predicate, which is the ordinary append where the arguments have been switched in order to get indexing over instantiated argument.

• inst(X,C,C1)
  Replaces x with a new variable in c, producing c1. This primitive is used in Π-quantified conditions.

The implementation is straightforward:

  inst(X,C,C1) :-
      term_variables(X,Vars), % Get variables
      subst_vars(Vars,C,C1).  % Substitute variables

• ([X] i Call) -> L
  i/2 is an index function: L is a list containing all the instances of x for which there exist a proof of Call. Call may be any call to the control level, i.e. Call is put to the right of the control level turnstile.

Index functions are compiled into calls to the primitive bagof_gclla/3, roughly ([X] i Call) -> L is transformed into bagof_gclla(X, Call1, L) where Call1 is Call compiled in the same way as a term occurring to the right of the control level turnstile, which is explained in the section 4.4, and with each variable introduced by the compiler as explicitly existentially quantified in Call1 by the operator ^/2.

bagof_gclla(Template, Generator, List) differs from Prolog's ordinary bagof(Template, Generator, List) in two ways:

1) If the generator fails to produce any solution, bagof/3 fails while bagof_gclla/3 returns List bound to [], and

2) The variables of Template are free in bagof_gclla/3, while they are existentially quantified in bagof/3, i.e. Template's unbound variables of bagof_gclla/3 are the same as outside the bagof_gclla/3 call. The reason for this is that variables in Generator should be object level variables, and therefore they should be treated as constants by bagof_gclla, which occurs on the meta level.

The following example shows the difference between Prolog's bagof/3 and bagof_gclla/3:
One can existentially bind variables in Generator (and Template) by the structure \texttt{Var^Generator}.

The implementation of \texttt{bagof_gcla/3} is based on \texttt{bagof/3}, where the routine for finding free variables in Generator is also used to find free variables in Template. This means that one can bind variables in Template in the same way as in Generator, i.e.

\begin{verbatim}
| ?- bagof_gcla(X, Z, member(X, [Y, Z]), L).
L = [(Y, Z), (Z, Z)] ?
| ?- bagof((X, Z), member(X, [Y, Z]), L).
L = [(Y, _A), (2, _B)] ?
\end{verbatim}

The reason for searching for free variables in Template also is that Template contains object level variables, and such variables should be treated as constants by the index function, which occurs at the control level.

- \texttt{findall_gcla(Template, Generator, List)}
  Defined analogously to Prolog’s \texttt{findall/3}, i.e. all variables in Generator are existentially bound.

### 3.3 Handling Dynamic Clauses

To handle the primitive provisos \texttt{def/1} and \texttt{rem/1}, which dynamically assert and retract object level clauses, the runtime system must be able to dynamically assert clauses into the object level definition and retract clauses from the object level definition, and to keep track of the current state of the object level definition. The implementation must also take care of user interruption and abortion, when the definition may need to be cleaned up.

Since there could be several identical clauses in a definition, modulo variable renaming, there is also a need for having unique identifiers for the object level clauses, to unambiguously refer to an added or removed clause. The unique identifier is used by the low level primitive \texttt{$\textsubscript{add}/1$} to remove the added clause upon backtracking, and the unique identifier is also used by \texttt{$\textsubscript{rem}/1$} to unambiguously mark a clause as removed from the definition.

To keep track of the current object level definition through the execution, we need some dynamic and global context, where changes are stored and which restores itself when backtracking occurs as well as when the execution is finished and a new query is posed to the system.

In Sicstus Prolog, there is a primitive \texttt{undo/1}, which in principle solves the problem. It can be used to add goals on the trail, which are executed when backtracking unwinds the trail stack down to the stored goal. Since GCLA uses the Prolog top level, and the Prolog top level is a repeat-fail loop, it will work correctly also for goals that succeed. It will also work correctly for interruption and abortion, since they also backtrack up to the top level. But there is a but: what happens if we get an interruption while executing the undo goal? We will get an interruption, and the current undo goal will be interrupted. This means that it can be unsafe to rely on undo goals to clean up and restore the dynamic clauses, if we cannot turn off interruption during the undo goals.

We therefore use \texttt{undo/1} to remove dynamically asserted clauses in \texttt{$\textsubscript{add}/1$}, but we do not actually remove clauses in \texttt{$\textsubscript{rem}/1$}, in order not to accidentally lose any clauses that are contained in the top level state of the object level definition. Instead a global context list is introduced, to hold the identifiers of the clauses currently removed. It could have
been done as a DCG, i.e. all clauses of the rule interpreter and the proviso clauses are augmented by two arguments, and the context state is passed around as the execution proceeds. This is the normal way of having a global state in an interpreter in pure logic (and functional) programs. But since the global state is not needed at most places, it is not space efficient to have two extra arguments which consume space whenever a choice point is generated and whenever a new stack frame is pushed onto the local stack in connection with a new procedure call in WAM. We have therefore designed a new global context, which is a list located in the heap of the WAM, and with a handle initiated at the beginning of the execution of every top level query through some C-primitives (see [Sicstus] for the foreign language interface)\(^2\). Since the list is not passed around, no extra space is needed and all rule interpreter clauses and proviso clauses that do not use the context are unaware of that it exists. Also, if the list contains logic variables, they will behave correctly. The list is protected from the garbage collector through the C-primitives.

The handle is accessed through three C-procedures, linked together with the Sicstus Prolog, and five new Prolog primitives are introduced to handle the list. We have also found context lists useful for e.g constraint handling, and other usages where a global state is changed by a program, so we have added four new user defined primitives as well.

The C procedures are

```c
unsigned long saved   /* Handle, points into the heap */
int set_global(t)      /* protects pointer t from the GC */
unsigned long get_global() /* Returns the pointer saved */
int free_global()       /* Hides saved from the GC */
```

and the Prolog procedures are

```prolog
init_contexts :-
    set_global(f([],[])), % f(System_context, User_context)
    undo(free_global).

get_context(List) :-
    get_global(Con),
    arg(1,Con, List).

in_context(Elem) :-
    get_global(Con),
    arg(1,Con, List),
    member(Elem, List).

add_context(Elem) :-
    get_global(Con),
    arg(1,Con, List),
    setarg(1,Con, [Elem|List]). % set arg 1 of Con to new value

del_context(Elem) :-
    get_global(Con),
    arg(1,Con, List),
    append(Tmp1, [Elem|Tmp2], List),
    append(Tmp1, Tmp2, List1),
    setarg(1,Con, List1). % set arg 1 of Con to new value
```

The four user defined primitives are defined analogously to the four last Prolog clauses, with the only change that all references to the user context is to the second argument of the structure f(SystemContext, UserContext).

Whenever a clause is removed from the object level definition, the unique identifier of the clause is pushed onto the system context, which is checked by all primitives that access

\(^2\)Thanks to Kent Boortz at SICS for helping me with the issues surrounding the C-primitives
the object level definition. When backtracking occurs, the old state is automatically restored since the context list behaves like any other structure located on the heap. This means that the system does not perform expensive asserts upon backtracking, as we would do if we had used undo/1 instead, and actually removed the clauses. The drawback is that when the context list gets large, the check whether the clause has been removed or not can be slow.

To get a fast clean-up of the object level definition (where some dynamically asserted clauses can still be left from the previous query if it was aborted at a critical point) an extra argument is added to the representation of object level clauses. This argument can take two values, program for 'this clause is defined at the top level' (i.e. has been asserted while loading a file), and dyn for 'this clause is asserted by def/1'.

3.4 Representation of Rules and Strategies

Rules and strategies are compiled into an interpreter of object level sequents. Roughly, the transformation of a rule is described by the following schema. The rule

\[
\text{rule}(A_1, \ldots, A_n) <= \\
\quad \text{Proviso}_1, \\
\quad \ldots, \\
\quad \text{Proviso}_k, \\
\quad (PT_1 \rightarrow \text{Sequent}_1), \\
\quad \ldots, \\
\quad (PT_m \rightarrow \text{Sequent}_m) \\
\rightarrow \text{Sequent}.
\]

is transformed into the rule interpreter clause

\[
\text{r}(\text{rule}(A_1, \ldots, A_n), \text{Sequent}) :- \\
\quad \text{Append_calls_from_sequent}, \\
\quad \text{Proviso}_1, \\
\quad \ldots, \\
\quad \text{Proviso}_k, \\
\quad \text{Append_calls_to_sequent}_1, \\
\quad \text{r}(PT_1, \text{Sequent}_1), \\
\quad \ldots, \\
\quad \text{Append_calls_to_sequent}_m, \\
\quad \text{r}(PT_m, \text{Sequent}_m).
\]

Strategies are transformed in a similar manner. Since clauses defining the same atom should be treated together, the two-clause definition below is transformed to one clause of the rule interpreter. The code

\[
\text{strategy}(A_1, \ldots, A_n) <= \text{Proviso}_1, \ldots, \text{Proviso}_k \rightarrow \text{Sequent}. \\
\text{strategy}(A_1, \ldots, A_n) <= \\
\quad \text{strategy}_1(\ldots A_i \ldots), \\
\quad \ldots, \\
\quad \text{strategy}_n(\ldots A_j \ldots).
\]

is transformed into

\[
\text{r}(\text{strategy}(A_1, \ldots, A_n), \text{Sequent}) :- \\
\quad \text{Proviso}_1, \\
\quad \ldots, \\
\quad \text{Proviso}_k, \\
\quad \text{r}(\text{strategy}_1, \text{Sequent}); \\
\quad \ldots; \\
\quad \text{r}(\text{strategy}_n, \text{Sequent}).
\]
The number of clauses defining a rule or strategy can be arbitrary, and the bodies of the control level clauses are merged into one body of the rule interpreter in the order in which they appear in the control level definition. For example, the strategy

\[
\text{strategy}(A_1, \ldots, A_n) <= \text{Body}1 -> \text{Sequent}.
\]
\[
\text{strategy}(A_1, \ldots, A_n) <= \text{Body}2 -> \text{Sequent}.
\]
\[
\text{strategy}(A_1, \ldots, A_n) <=
\]
\[
\text{strategy}_1(\ldots A_i \ldots),
\]
\[
\ldots,
\]
\[
\text{strategy}_n(\ldots A_j \ldots).
\]

is compiled into

\[
r(\text{strategy}(A_1, \ldots, A_n), \text{Sequent}) :-
\]
\[
\text{Body}1,
\]
\[
\text{Body}2,
\]
\[
r(\text{strategy}_1, \text{Sequent}) ;
\]
\[
\ldots ;
\]
\[
r(\text{strategy}_n, \text{Sequent}).
\]

where \text{Body}1 and \text{Body}2 in turn consist of proviso calls and calls to the rule interpreter.

The rule interpreter always contains as the first case the initial sequent rule, for the case when the control level antecedent consists of an object level sequent, in which case the initial sequent is the only possible inference rule to apply. The clause below is always present in the GCLA system.

\[
r(A, B, C, C) :-
\]
\[
\text{nonvar}(A),
\]
\[
\text{functor}(A, \land, 2), !,
\]
\[
A=B.
\]

### 3.5 Representation of Provisos

Provisos are represented as Prolog predicates. All provisos are also asserted in a predicate proviso(Head, Body). This predicate is used in the debugger and by the gnot/1 predicate, which implements negation for provisos (see section 3.2). In principle, the user defined proviso clause user_proviso/n with the definition

\[
\text{user_proviso}(\text{Arg}_1, \ldots, \text{Arg}_n) :-
\]
\[
\text{provisocall}_1(\text{Arg}_{i1}, \ldots, \text{Arg}_{ik}),
\]
\[
\ldots,
\]
\[
\text{provisocall}_n(\text{Arg}_{m1}, \ldots, \text{Arg}_{mk}).
\]

is transformed into

\[
\text{user_proviso}(\text{Arg}_1, \ldots, \text{Arg}_n) :-
\]
\[
\text{provisocall}_1(\text{Arg}_{i1}, \ldots, \text{Arg}_{ik}),
\]
\[
\ldots,
\]
\[
\text{provisocall}_n(\text{Arg}_{m1}, \ldots, \text{Arg}_{mk}).
\]
\[
\text{proviso}(	ext{user_proviso}(\text{Arg}_1, \ldots, \text{Arg}_n),
\]
\[
(\text{provisocall}_1(\text{Arg}_{i1}, \ldots, \text{Arg}_{ik}),
\]
\[
\ldots,
\]
\[
\text{provisocall}_n(\text{Arg}_{m1}, \ldots, \text{Arg}_{mk}).
\]

where provisocall\_1 can be a call to a user defined proviso, a system defined proviso or a call to a Prolog procedure.
4. The Compiler

The compiler is written in Prolog, and transforms the control code of GCLA II into an interpreter of GCLA II's object level sequents. The compiler consists of three main modules: the reading module and preprocessing module, the transforming module, and the asserting module. In principle, the iteration is described by the following procedure:

```
top(end_of_file) :- !.
top(Cl) :-
    read_equal_heads(Cl, Clauses, NewCl, ProvOrRule),
    (ProvOrRule == rule
     -> group(Clauses, GroupsOfClauses),
     transform(GroupsOfClauses, PrologCode),
     assert_rules(PrologCode)
    ; parse_proviso(Clauses, PrologCode),
    assert_provisos(PrologCode),
    top(NewCl).
```

where `group` groups the clauses in the `Clauses` according to the possible $A$-sufficient substitutions, and returns all possible (positive, i.e. no completion as in [Aro93a]) instances in `GroupsOfClauses`. `transform` transforms the grouped clauses into the corresponding Prolog code, and `assert_code` stores the Prolog code.

4.1 The Reading Module

The reading module contains no complex or otherwise complicated operations. It reads clauses until a clause whose head's main functor differs from the other read clauses head's main functor is encountered, and then returns the list of clauses read, plus the clause which differed (since it is not possible to put a read clause back again when it has been read). We use Prolog's `read-primitive`, which also performs a syntax check that is appropriate at the level of parsing characters into names and symbols. There is no difference in reading rules, strategies or provisos, they are all read by the same routine.

Since rules and strategies are executed to the left, the definiens operation together with the generation of $A$-sufficient substitutions has to be performed on the heads of clauses that are unifiable. By doing this at compile time in the same manner as in [Aro93a], with the third algorithm for substitutions, we get all the possible sets of rules. Since there are no guards at the control level, the generation of $A$-sufficient substitutions together with the definiens operation is quite simple, and is done by the `group/2` procedure in the top level routine above. The output from the `group/2` routine is a list consisting of pairs, where each pair has the head resulting after the $A$-sufficient substitution has been applied to it as one argument and a semi-colon-separated structure consisting of the corresponding bodies as the second argument, also with the $A$-sufficient substitution applied to it. This is the input to the transformation phase, where the bodies are used to form the bodies of the rule interpreter.

Call to provisos from rules or strategies are always made from an atom occurring to the right. The only cases when the execution is switched to the left are when

- some object level sequent shall be proved by some specified proofterm,
- the negation of some atom is encountered, or
- an index function shall be executed.

It is only in the second case that the definiens and the generation of an $A$-sufficient substitution are used, and it turns out in practice that it is a quite simple and cheap operation, and therefore we do not generate a special representation for the definiens and $A$-sufficient proviso as we do for rules, and for object level code (see [Aro93a]).
4.2 The Target Language

The target language consists of some of the Prolog primitives, extended with the general procedures and the primitive operations described in section 3.2.

Below is a short description of the Prolog primitives used by the interpreter. For a complete description of them we refer to [Sicstus].

- \( \text{arg}(\text{Term}, \text{Number}, \text{Argument}) \)
  Argument is the \text{Number}th argument of \text{Term}.

- \( \text{var}(\text{Term}) \)
  Term is currently a variable

- \( \text{nonvar}(\text{Term}) \)
  Term is currently instantiated to some nonvariable term

- \( \text{Term1} \equiv \text{Term2} \)
  Term1 is identical to Term2

- \( \text{if}(\text{If}, \text{Then}, \text{Else}) \)
  If If succeeds, Then is executed, else Else is executed. Note that if Then fails, backtracking occurs into If again, but if If succeeds once, Else is never executed.

- \( \text{Call1 ; Call2} \)
  Call1 or Call2: if Call1 fails, Call2 is executed

- \( \text{Term1 = Term2} \)
  Term1 is unified with Term2

- \( \text{Module:Call} \)
  Call is executed in module Module. This is used by the user to call Prolog primitives and other, user-defined predicates. By explicitly giving Module, such calls are recognized, and are not handled by the proviso-parser but kept intact.

4.3 Compiling the Object Level Definition

Compiling control is mostly concerned with the control level, but the generation of the databases for the object level definition can affect the number of choicepoints generated at compile time, and therefore we will mention the way the object level definition is handled.

The object level definition is compiled into a database, which consists of three different clauses. First of all, there are two different operations; one for collecting one clause at a time, and one operation for returning the definiens and a-sufficient substitution given the atom. The latter is formed as explained in [Aro93a] in the relation \text{gcla_base2/3}, while the former consists of two different Prolog predicates: \text{gcla_base/4} and \text{gbase_name/Arity}, where name is the name of the object level atom that is defined, and \text{Arity} is the arity of the atom being defined. \text{gcla_base/4} is used to get indexing over the principal functor of the atom, and then \text{gbase_name/Arity} is called to get indexing over the first argument of the atom, so there is always one \text{gcla_base/4} clause for all clauses that have the same atom as its head, while there is one clause for every object level clause defining the atom.
As an example, consider how the object level definition of append/3 is compiled. The definition is:

\[
\text{append}([], L, L).
\]
\[
\text{append}([F|R], L, [F|R1]) \Leftarrow \text{append}(R, L, R1).
\]

and it is compiled into

\[
\text{gclla_base}(\text{append}(X, Y, Z), \text{Body}, 1, \text{program}) :- \nonumber \\
\text{gbase_append}(X, Y, Z, B).
\]
\[
\text{gbase_append}([], L, L, \text{true}).
\]
\[
\text{gbase_append}([F|R], L, [F|R1], \text{append}(R, L, R1)).
\]

% for dp/4
\[
\text{gclla_base2}(\text{append}([], L, L), \text{true}, 1).
\]
\[
\text{gclla_base2}(\text{append}([F|R], L, [F|R]), \text{append}(R, L, R1), 1).
\]
\[
\text{gclla_base2}(\text{append}(A, B, C), \text{false}, 0) :- \\
\text{append}(A, B, C) \Leftarrow \text{append}([], L, L),
\]
\[
\text{append}(A, B, C) \Leftarrow \text{append}([F|R], L, [F|R1]).
\]

Note that for \text{gclla_base}/3 we get indexing on the first argument of append/3, while in the latter one we do not. Of course, it is simple to extend \text{gclla_base2}/3 in the same way as \text{gclla_base}/3, to get indexing there too, but indexing does not yield the expected improvement for the \text{gclla_base2}/3 database. Since \text{gclla_base2}/3 is completed with the third case when the atom is defined to be \text{false} (the third clause in the example above), and this clause is usually of the kind where the first argument is a variable, we do not get indexing over the first argument, which we achieve in the \text{gclla_base}/4 case. The reason why the third clause is not generated in the \text{gclla_base}/4 case is that there is hardly any point in generating the case for which the defined atom is \text{false}, since it will occur to the right of the derivability symbol \text{"-"} (for all 'meaningful' rules). Of course, the user can make the definition complete with a clause making the definition of an atom complete, which in the example above would be to give the third clause

\[
\text{append}(X, Y, Z) \#(\text{app}(X, Y, Z) \Leftarrow \text{app}([], L, L),
\]
\[
\text{app}(X, Y, Z) \Leftarrow \text{app}([F|R], L, [F|R1])) \Leftarrow \text{false}.
\]

which would make the definition total, but if we only consider the non-\text{false} cases, as is the common case for \text{gclla_base}/4, we do not get indexing over the first argument.

4.4 Transformation of Control Code into Target Code

Since Sicstus Prolog cannot distinguish between object level variables and meta level variables, we have to implement them as well as we can on a single layer. There are some ways to implement a true object level, for example by using lists to hold the explicit binding environment, or encapsulating all object level terms (including variables) with some specially designated functor, in order to be able to use Prolog's unification. Both these approaches slow down the execution. However, we have tried to keep the levels separate by restricting the use of object level terms, and when they occur in rules and strategies, replace every occurrence of implicit object level unification with explicit unification in terms of matching primitives (e.g. \text{functor/3, var/1} etc). For provisos the situation is more complex, since we cannot determine which terms are object level and which terms are control level. Therefore, all terms are treated as control level terms, i.e. there is no transformation of structures to explicit unification, and the user is strongly encouraged to use the primitives \text{functor/3, var/1} and \text{arg/3} when dealing with object level structures, to implement the matching himself. With abstract interpretation it would perhaps be possible to detect object level terms and control level terms used in the provisos, but we have not explored this possibility yet.
4.4.1 Unification of Object Level Terms

Whenever there exists something other than a variable in the position of an object level term as a value of a rule or strategy (i.e., as a consequence of the object level sequent, or in the object level antecedent, specified through the append operator 'e'), the non-variable term will be matched with the current goal sequent during proof search. Therefore the term must be replaced with primitives that perform the matching instead of unification at runtime (see the rule Initial sequent in section 2.4).

All object terms that are not variables are transformed into a sequence of operations that perform matching with the runtime goal sequent instead, by the operation compile_unify/3, through the operations arg/3, gfunctor/3 and ==/2:

```
compile_unify(X,X1,true) :- var(X),!; X1 = X.
compile_unify(X,X1,X == X1) :- atomic(X),!.
compile_unify([|R|],X1,
               (gcla:gfunctor(X1,'.',2),
                arg(1,X1,X1),arg(2,X1,X2),Sofar1)) :-
    compile_unify(F,Y1,Tmp1),
    compile_unify(R,Y2,Tmp2),
    struct_append(Tmp1,Tmp2,Sofar1).

compile_unify(Struct,X1,(gcla:gfunctor(X1,S,N),Answer)) :-
    functor(Struct,S,N),
    Struct=...[|Args],
    compile_unify_struct(Args,1,X1,Answer).

compile_unify_struct([|X|],N,X1,(arg(N,X1,A),Answer)) :-
    compile_unify(X,A,Answer).

compile_unify_struct([|X|,X2|R|],N,X1,(arg(N,X1,A),Answer)) :-
    compile_unify(X,A,Tmp),
    N1 is N + 1,
    compile_unify_struct([X2|R],N1,X1,TmpR),
    struct_append(Tmp,TmpR,Answer).
```

Note that all arguments of an object level term have to be picked out, since we do not know at this point which arguments are used later. To be able to remove all arg/3 calls which pick out void terms we have to do an abstract interpretation to find out which arguments that are never used, but a lot of the redundant arg/3 calls can be removed simply by checking that its third argument (the 'output' argument) is never used in the resulting rule interpreter clause.

The append operation 'e' used in the object level antecedents is compiled to calls to append/3. There are two cases: either the antecedent occurs as a result of a rule, in which case it will be an input argument of a clause of the interpreter, or the antecedent occurs as a function call of one of the premises of the rule, in which case the antecedent is an output argument of the interpreter, i.e. it occurs as an argument of a call to the interpreter. To distinguish between those two cases, the Comp_unif argument is used. If Comp_unif is instantiated to true, the antecedent occurs in the result of the rule, and the specified terms of the antecedent should be checked to see if they should be replaced by matching primitives. If Comp_unif is out, the antecedent occurs as an argument to one of the premises of the rule, and the specified elements of the antecedent can be kept as they are.

```
transform_append(Atsigns,X,AppendCalls,Comp_unif) :-
    get_append_calls(Atsigns,X,Tmp,Comp_unif),
    get_rid_of_trues(Tmp,AppendCalls).
```

- 24 -
get_append_calls(A@B,Assum, (Rapps1, (gcla:append1(Assum,B1,A1),Rapps2)), in) :- !,
get_append_calls(A,A1,Rapps1,in),
get_append_calls(B,B1,Rapps2,in).
get_append_calls(A@B,Assum, (Rapps1, (gcla:append(A1,B1,Assum),Rapps2)), out) :- !,
get_append_calls(A,A1,Rapps1,out),
get_append_calls(B,B1,Rapps2,out).
get_append_calls([F|R],[F1|R1],Answer,in) :- !,
compile_unify(F,F1,Temp),
get_append_calls(R,R1,Temp2,in),
struct_append(Temp,Temp2,Answer).
get_append_calls([F|R],[F1|R1],Answer,out) :- !,
get_append_calls(R,R1,Answer,out).
get_append_calls(X,X,true,_) .

4.4.2 Rules and Strategies

Rules and strategies are transformed by two procedures, one which transforms terms to the right, and one which transforms terms to the left. Recall that the choice of control level inference rule is deterministic (see section 2.5), so the choice of term to look at in each execution step is dependent on the control level sequent at each step. Since the choice of inference rule (see section 2.4 for the inference rules) is deterministic, we can determine which rule will be used for each condition of a clause in the control code, and thus we can determine at compile time which rule will be used. Therefore, the control level sequents can be compiled away, and replaced with procedures that accomplish the same task, based on the structure of the rules and strategies.

There are two basic transformation schemes for rules and strategies. One for transforming conditions on the right side, which means that the condition will appear on the right of the control level sequent, and an inference rule operating on the right side should be used. Another transformation schema performs an analogous transformation for conditions on the left side.

Transformation of conditions on the right side is in principle to replace the rule calls and some other operations by other Prolog operations. When an arrow is encountered, the execution is moved to the left, and the transformation on the left side is applied to the antecedent, if the consequent of the arrow is not on the form (A \ -> C), in which case it is a call to the rule interpreter.

Transformation of terms on the right side is described by the following principal transformation schema, where italics are the transformation procedures and non-italics are the target code.

right( (A,B) ) \rightarrow (right(A), right(B))

right( (A;B) ) \rightarrow (right(A) ;
right(B))

right( (R -> (A \ -> C)) ) \rightarrow (AppendCalls, left(R, (AVar \ -> C)) )

where transform_append( A, AVar, AppendCalls, out ) holds,
AppendCalls contains calls to append/3, which will bind AVar to a list of object level terms
right(not(A)) ⇒
gnot(right(A))

right(\+(A)) ⇒ \+(right(A))

right((i([Template],Call) -> List)) ⇒
bagof1(Template,Ev^right(Call),List)

where Ev are all variables that should be existentially quantified

right(atom(X)) ⇒ gatom(X)

right(true) ⇒ true

right(functor(X,Y,Z)) ⇒ gfunctor(X,Y,Z)

right(clause(X,Y)) ⇒ gc_clause(X,Y)

right(definiens(X,Y)) ⇒ dp(X,Y)

right(add(X)) ⇒ $add(X)  % $add is low level add/1

right(rem(X)) ⇒ $rem(X)  % $rem is low level rem/1

right(Atom) ⇒ Atom

If Atom is a GCLA primitive other than the above specified, or if Atom is a
Prolog built-in primitive.

right(X) ⇒ parse_atom(X,X1)

Transforming terms on the left side contains more transformations. When the execution
switches to the left of the turnstile (through a consequent (A -> B)), it is because of a
rule call, a call to an index function, or a call to the if-then-else construct (see [Aro93b]
for its semantics). The control level inference rule Initial sequent is coded inline by the
7:th clause below, since if a sequent (A \- C) appears on the left side, the only way to
succeed is to apply the Initial sequent inference rule.

left(A,Seq) ⇒ r(A,Seq)
  if A is currently a variable (often a proof term is not instantiated at compile
time)

left(((A \- B),C),Seq) ⇒
  if(left(A,Seq),
    right(B),
    left(C,Seq))

left((P -> (A \- C)),(AVar \- CVar)) ⇒
  (Calls,AppendCalls,right(P))

where
  • transform_append(A,AVar,AppendCalls,I0) holds, AppendCalls
    contains calls to append/3, which will bind A1 to a list of object level
terms, and I0 is in if (A \- C) is the result of the current rule or
    strategy, otherwise out.
• If \( A \leftarrow C \) is the result of the current rule or strategy, \texttt{Calls} is the code that matches the input variable \texttt{CVar} to the structure of \( C \), which is generated by \texttt{compile_unify/3}. Otherwise, \texttt{CVar} and \( C \) are the same terms, and \texttt{Calls} is simply true.

\[
\text{left}((P \rightarrow A),\text{Seq}) \Rightarrow (\text{right}(P), \text{left}(A,\text{Seq}))
\]

\[
\text{left}((A,B),\text{Seq}) \Rightarrow (\text{left}(A,\text{Seq}); \text{left}(B,\text{Seq}))
\]

\[
\text{left}((A;B),\text{Seq}) \Rightarrow (\text{left}(A,\text{Seq}), \text{left}(B,\text{Seq}))
\]

\[
\text{left}((A \leftarrow C), (\text{AVar} \leftarrow \text{CVar})) \Rightarrow (\text{Calls, AppendCalls})
\]

where

• transformAppend\((A,\text{AVar},\text{AppendCalls},\text{IO})\) holds, \texttt{AppendCalls} contains calls to append/3, which will bind \( A1 \) to a list of object level terms, and \( \text{IO} \) is in if \( (A \leftarrow C) \) is the result of the current rule or strategy, otherwise out.

• If \( (A \leftarrow C) \) is the result of the current rule or strategy, \texttt{Calls} is the code that matches the input variable \texttt{CVar} to the structure of \( C \), which is generated by \texttt{compile_unify/3}. Otherwise, \texttt{CVar} and \( C \) are the same terms, and \texttt{Calls} is simply true.

\[
\text{left}((\text{pi Var} \leftarrow A),\text{Seq}) \Rightarrow \text{left}(\text{NewA},\text{Seq})
\]

where \( \text{NewA} \) is \( A \) with \( \text{Var} \) substituted with a new unique variable.

\[
\text{left}(A,\text{Seq}) \Rightarrow \text{r}(A,\text{Seq})
\]

All other cases are assumed to be calls to rules or strategies

Atoms are kept as they are:

\[
\text{parse_atom}(X) \Rightarrow X
\]

### 4.4.3 Provisos

Provisos could be any Prolog code. The transformation is simply to replace the special constructs that are defined by GCLA, such as \texttt{atom/1} with \texttt{gatom/1}, \texttt{functor/3} with \texttt{gfunctor/3} etc. Also, the extra two arguments for handling the execution state are added to every user defined proviso.

The transformation is similar to the transformation of conditions to the right for rules and strategies.

\[
\text{prov_body}((A,B),I,O) \Rightarrow (\text{prov_body}(A,I,T), \text{prov_body}(B,T,O))
\]

\[
\text{prov_body}((A;B),I,O) \Rightarrow (\text{prov_body}(A,I,T1),T1=0 ; \text{prov_body}(B,I,T2),T2=0)
\]

\[
\text{prov_body}((R \rightarrow (A \leftarrow C)),I,O) \Rightarrow (\text{AppendCalls, left}(R,(\text{AVar} \leftarrow C),I,O))
\]
where \( \text{transform\_append}(A, AVar, \text{AppendCalls}, \text{out}) \) holds, \text{AppendCalls} contains calls to append/3, which will bind AVar to a list of object level terms. Note that \text{left} will return a call to the rule interpreter.

\[
\begin{align*}
\text{prov\_body}(\text{not}(A), I, O) \Rightarrow \\
g\text{not}(\text{prov\_body}(A, I, O)) \\
\text{prov\_body}(\text{atom}(X), I, I) \Rightarrow g\text{atom}(X) \\
\text{prov\_body}(\text{functor}(X, Y, Z), I, I) \Rightarrow g\text{functor}(X, Y, Z) \\
\text{prov\_body}(\text{clause}(X, Y), I, I) \Rightarrow g\text{clause}(X, Y, I) \\
\text{prov\_body}(\text{definiens}(X, Y), I, I) \Rightarrow d\text{p}(X, Y, I) \\
\text{prov\_body}(\text{add}(C), I, O) \Rightarrow \text{add}(C, I, O) \\
\text{prov\_body}(\text{rem}(C), I, O) \Rightarrow \text{rem}(C, I, O) \\
\text{prov\_body}(\text{Module}:\text{Atom}, I, I) \Rightarrow \text{Module}:\text{Atom} \\
\text{prov\_body}(:\text{Atom}, I, I) \Rightarrow \text{Atom} \\
\text{prov\_body}(\text{Atom}, I, O) \Rightarrow \text{parse\_atom}(\text{Atom}, I, O).
\end{align*}
\]

5. Examples

The examples given are selected to show the properties of the compiler; what the output is for different possible inputs. We have tried to make them as different as possible, to give as complete a picture as possible without repeating similar examples.

5.1 Rules and Strategies

The following are examples of rules and strategies, i.e. the functional part of a control level definition. We start with common rules, continue with common strategies and more complicated strategies and lastly how index functions are compiled.

5.1.1 Common Object Level Inference Rules

Some examples of common object level inference rules are:

\[
\begin{align*}
\text{axiom}(T, C, I) \Leftarrow \\
\text{term}(T), \\
\text{term}(C), \\
\text{unify}(T, C) \\
\Rightarrow (I \oplus [T | _ \leftarrow C]). \\
\text{d\_right}(C, PT) \Leftarrow \\
\text{atom}(C), \\
\text{clause}(C, B), \\
(PT \leftarrow (P \leftarrow B)) \\
\Rightarrow (P \leftarrow C).
\end{align*}
\]
\[
d_{\text{left}}(T, I, PT) \Leftarrow \\
\text{atom}(T), \\
\text{definiens}(T, Dp, N), \\
(PT \Rightarrow (I \oplus [Dp|Y] \setminus C)) \\
\Rightarrow (I \oplus [T|Y] \setminus C).
\]

The rules above are compiled to the following clauses of the rule interpreter

\[
r(\text{axiom}(B,C,D), (A \setminus C)) \Leftarrow \\
gcla:\text{append1}(A, [B|\_], D), \\
term(B), \\
term(C), \\
\text{unify}(B, C).
\]

\[
r(\text{d_right}(C,D), (B\setminus C)) \Leftarrow \\
gcla:\text{gatom}(C), \\
gcla:\text{gclause}(C, A), \\
r(D, (B\setminus A), E, F).
\]

\[
r(\text{d_left}(F,G,H), (D\setminus E)) \Leftarrow \\
gcla:\text{append1}(D, [F|C], G), \\
gcla:\text{gatom}(F), \\
gcla:dp(F, B), \\
gcla:\text{append}(G, [B|C], A), \\
r(H, (A\setminus E)).
\]

Note how the append-operator 'eq' is transformed into calls to the gcla primitive `append/3` and `append1/3`.

### 5.1.2 Calling Prolog from GCLA

The following rule shows how Prolog is called from GCLA:

\[
\text{write_right}((\text{write}(X))) \Leftarrow \\
\text{write}(X) \Rightarrow (_\setminus \text{write}(X)).
\]

is compiled into

\[
r(\text{write_right}((\text{write}(B))), (_\setminus A)) \Leftarrow \\
gcla:gfundtor(A, \text{write}, 1), \\
arg(1, A, B), \\
\text{write}(B).
\]

Note that the consequent of the object level sequent (\text{write}(X)) is compiled into the matching code `gcla:gfundtor(A, \text{write}, 1), arg(1, A, B),` to prevent accidental unification of arguments.

### 5.1.3 A Complicated Rule

The following is an example of a complicated rule, to show that when the arrow is iterated twice, we get back the same call

\[
\text{complicated_rule}(PT) \Leftarrow \\
\text{((PT} \Rightarrow (A \setminus C)) \Rightarrow (A \setminus C)) \Rightarrow (A \setminus C).
\]

\[
\text{simple_rule}(PT) \Leftarrow \\
(PT \Rightarrow (A \setminus C)) \Rightarrow (A \setminus C).
\]

\[
r(\text{complicated_rule}(C), (A\setminus B)) \Leftarrow \\
r(C, (A\setminus B)).
\]

\[
r(\text{simple_rule}(C), (A\setminus B)) \Leftarrow \\
r(C, (A\setminus B)).
\]

Note that the two rule interpreter clauses are the same.
5.1.4 Common Strategies
Common strategies consist of a single clause, with a comma-separated vector as body.

\[
\begin{align*}
\text{arl} & \leftarrow \text{axiom}(\_,\_,\_), \\
& \quad \text{right}(\text{arl}), \\
& \quad \text{left}(\text{arl}). \\
\text{alr} & \leftarrow \text{axiom}(\_,\_,\_), \\
& \quad \text{left}(\text{alr}), \\
& \quad \text{right}(\text{alr}). \\
\text{lra} & \leftarrow \\
& \quad \text{pi } A \setminus \text{pi } B \setminus \text{pi } C \setminus \\
& \quad \text{left}(\text{lra}), \\
& \quad \text{right}(\text{lra}), \\
& \quad \text{axiom}(A,B,C).
\end{align*}
\]

Note that we have written out the bound variables in the body of the last clause, which is not necessary since if variables in rules or strategies are not bound by quantifiers, we assume that the variable is universally bound. This is similar to most Horn clause systems, where free variables only occurring in the body of a clause are assumed to be existentially quantified in the body. The above strategies are compiled into

\[
\begin{align*}
\text{r}(\text{arl}, E) & \leftarrow \\
& \quad \{ \text{r}(\text{axiom}(\_,\_,\_), E), \\
& \quad \quad \text{r}(\text{right}(\text{arl}), E), \\
& \quad \quad \text{r}(\text{left}(\text{arl}), E), \\
& \quad \}.
\end{align*}
\]

\[
\begin{align*}
\text{r}(\text{alr}, E) & \leftarrow \\
& \quad \{ \text{r}(\text{axiom}(\_,\_,\_), E), \\
& \quad \quad \text{r}(\text{left}(\text{alr}), E), \\
& \quad \quad \text{r}(\text{right}(\text{alr}), E), \\
& \quad \}.
\end{align*}
\]

\[
\begin{align*}
\text{r}(\text{lra}, E) & \leftarrow \\
& \quad \{ \text{r}(\text{left}(\text{lra}), E), \\
& \quad \quad \text{r}(\text{right}(\text{lra}), E), \\
& \quad \quad \text{r}(\text{axiom}(\_,\_,\_), E), \\
& \quad \}.
\end{align*}
\]

Note that the result is the same whether the variables are pi-quantified or not, since the compiler assumes that variables occurring only in the body of a rule or strategy clause are universally quantified.

5.1.5 Multi-clause Strategies
The examples before have all been examples of single-clause rules or strategies, but there can be several clauses defining a rule or strategy. The following example shows how a multi-clause strategy is compiled into two different clauses, where two different versions of the arrow-left rule are used depending on the conclusion of the arrow, C1:

\[
\begin{align*}
\text{a}_\text{left2}((A \rightarrow C1), I, PT, PT1, \text{left}) & \leftarrow \\
& \quad \text{data}(C1) \rightarrow (I\#[(A \rightarrow C1)\|\_] \setminus \_). \\
\text{a}_\text{left2}((A \rightarrow C1), I, PT, PT1, \text{left}) & \leftarrow \\
& \quad \text{a}_\text{left1}((A \rightarrow C1), I, PT, PT1). \\
\text{a}_\text{left2}((A \rightarrow C1), I, PT, PT1, \text{right}) & \leftarrow \\
& \quad \text{not(data}(C1)) \rightarrow (I\#[(A \rightarrow C1)\|\_] \setminus \_). \\
\text{a}_\text{left2}((A \rightarrow C1), I, PT, PT1, \text{right}) & \leftarrow \\
& \quad \text{a}_\text{left}((A \rightarrow C1), I, PT, PT1).
\end{align*}
\]

The four clauses above are compiled by the compiler into two cases in the rule interpreter:
\[ r(a_{\text{left2}}((D \rightarrow E), F, G, H, \text{left}), (B \rightarrow C)) :-
\]
\[ \text{gcla:append1(B, [A | _], F)},
\]
\[ \text{gcla:gfunder system}(A, \rightarrow, 2),
\]
\[ \text{gcla:arg system}(1, A, D),
\]
\[ \text{gcla:arg system}(2, A, E),
\]
\[ \text{data}(E),
\]
\[ r(a_{\text{left1}}((D \rightarrow E), F, G, H), (B \rightarrow C)).
\]
\[ r(a_{\text{left2}}((D \rightarrow E), F, G, H, \text{right}), (B \rightarrow C)) :-
\]
\[ \text{gcla:append1(B, [A | _], F)},
\]
\[ \text{gcla:gfunder system}(A, \rightarrow, 2),
\]
\[ \text{gcla:arg system}(1, A, D),
\]
\[ \text{gcla:arg system}(2, A, E),
\]
\[ \text{data}(E),
\]
\[ r(a_{\text{left1}}((D \rightarrow E), F, G, H), (B \rightarrow C)).
\]

Note how the last argument of the strategy clauses groups the definition into two cases. This classification (grouping) is done by the definiens and A-sufficient substitution algorithms in the first part of the compiler.

### 5.1.6 Index Functions

The following example shows how a rule implementing bagof/4 (bagof(ExistVars, Template, Call, Collection) on the object level is coded in the control code:

\[ \text{bagof right}(\text{bagof}(EVars, A, B, C), (PT) <=
\]
\[ \text{lift from a}(B, B1, [], Vars),
\]
\[ \text{append}(EVars, Vars, Vars1),
\]
\[ (i([A], Vars1^PT^ (PT -> (Ass \rightarrow B1))) -> C) ->
\]
\[ (Ass \rightarrow \text{bagof}(EVars, A, B, C)).
\]

is compiled into

\[ r(\text{bagof right}(\text{bagof}(I, H, G, F), L), (J \rightarrow K)) :-
\]
\[ \text{gcla:gfunder(K, bagof, 4)},
\]
\[ \text{arg}(I, K, I),
\]
\[ \text{arg}(2, K, H),
\]
\[ \text{arg}(3, K, G),
\]
\[ \text{arg}(4, K, F),
\]
\[ \text{lift from a}(G, C, [], D),
\]
\[ \text{append}(I, D, B), \quad \% \text{User defined}
\]
\[ \text{gcla:bagof1(H, A^N^L^B^ (user:r(L, (J \rightarrow C))), F)).
\]

Note how the bagof-structure in the consequent of the resulting object level sequent is compiled into code for matching with the runtime sequent. Also note how the variables of the bagof-call introduced by the GCLA system have been existentially quantified.

### 5.2 Provisos

As explained, user defined provisos are asserted as ordinary Prolog clauses. The following example shows how two user defined provisos are compiled.

The proviso definition

\[ \text{lift from a}(V, V, L, L) : \text{var}(V).
\]
\[ \text{lift from a}(A, V, V, [V | L]) : \text{functor}(A, a, 1), A = a(V).
\]
\[ \text{lift from a}(\text{Atom}, \text{Atom}, L, L) : \text{atomic}(\text{Atom}).
\]
\[ \text{lift from a}(X, [F | R1], L, L) : \text{nonvar}(X), X = [F | R],
\]
\[ \text{lift from a}(F, F, F1, L, L1),
\]
\[ \text{lift from a}(R, R1, L1, L2).
\]
\[ \text{lift from a}(\text{Str}, \text{Str1}, L1) : \text{nonvar}(\text{Str}),
\]
\[ \text{Str}..[S | A], S \rightarrow '.', S \rightarrow a, A \rightarrow [],
\]
\[ \text{lift from a}(A, A1, L, L1),
\]
\[ \text{Str1}..[S | A1].
\]
data(X) :- functor(X,pl,_) .
data(X) :- functor(X,activity,_) .

is compiled into the Prolog code

lift_from_a(A, A, B, B) :-
  var(A).
lift_from_a(A, B, C, [B|C]) :-
  gcla gfunctor(A, a, 1),
  A=a(B).
lift_from_a(A, A, B, B) :-
  atomic(A).
lift_from_a(G, [E,F], H, I) :-
  nonvar(G),
  G=[C|D],
  lift_from_a(C, E, H, A),
  lift_from_a(D, F, A, I).
lift_from_a(D, E, F, G) :-
  nonvar(D),
  D=..[B|C],
  B\=',',
  B\=a,
  C\=[],
  lift_from_a(C, A, F, G),
  E=..[B|A].

data(A, B, B) :-
  gcla gfunctor(A, pl, _).
data(A, B, B) :-
  gcla gfunctor(A, activity, _).

Note that the GCLA primitives are used instead of the Prolog ones, e.g. functor/3 is replaced by gcla gfunctor/3.

6. Notes on Levels of Variables

GCLA II contains two levels: the object level and the control level. The control level specifies the inference rules and strategies to be used to make proofs of object level queries. Thus from this point of view the control level is a meta level to the object level, which implies that the variables of the two levels should be of different types. Object level variables should be treated as constants when occurring on the meta level: meta level variables can be bound to object level variables, but not the other way around.

In GCLA II binding of object level variables is handled through three primitives: unify/2, clause/2 and definiens/2 (i.e. the generation of an A-sufficient substitution coded inside the definiens operation). But in the implementation there is no distinction between meta level variables and object level variables, due to difficulties in finding an efficient way of separating the two levels. We have implemented a version with explicit handling of object level variables in the primitives unify/2, gclause/3 and dp/3, where object level variables are represented as a structure objvar(Var), to prevent unification of object level variables with meta level structures. This implementation of GCLA II is about 30% to 50% slower than the version where no distinction is made, depending on the amount of unification of object level structures present in the application. The decrease in execution time would be much less if there was a way to distinguish between different levels of variables, as in [Bar89]. However, we can also note that we know quite well which variables are object level variables, and which are meta level. In principle, object level variables occur in object level sequents, and variables in rules and strategies are meta level. The risk that the user will accidentally bind an object level variable to some meta level structure in a rule or strategy is small, since the object level sequents are compiled to matching routines at compile time. It is only for the arguments
of the heads of the rules and strategies that it cannot be determined at compile time whether they contain an object level variable or not, but if type declarations are introduced for the arguments in the same way as for many languages, the possible cases of wrong bindings can be detected. The risk that the user writes some proviso that accidentally binds an object level variable to a meta level structure is much higher, but typing conditions could be introduced here as well. But so far we have not regarded the risk of binding object level variables to meta level structures as high enough to justify the price of 30% to 50% slower execution. We have not implemented any typing declaration either, but will perhaps if there is a need for it.

An interesting question for research is whether the different levels actually are different, or whether they can be viewed as the same. By forming a definition that is self referential, the two levels can perhaps be amalgamated, which in turn will bring the implementation and the theory even closer together than today (see [Kre93] for a discussion of a self-referential calculus). The main motivation for this research is to have a clear understanding of what negation as failure actually means, but if the two levels are amalgamated, this also means that the levels of variables disappear. This is a very interesting approach, which will be further studied.

7. Discussion and Conclusion

We have described how GCLA II's control code can be compiled into an interpreter with the inference rules of the control level coded inline.

The performance of the system is comparable to that of any system implemented as an interpreter in Prolog. Of course, there are also GCLA routines that apply the Sicstus Prolog compiler on the rule interpreter and user defined provisos, which gives the GCLA system the same efficiency as any other Prolog interpreter. The point 'why use GCLA' is not that GCLA itself is more efficient than Prolog, for Horn clause programs, but that GCLA offers the user a clear distinction between object level constructs and control level constructs together with higher expressive power, and by the compilation described here he gets a system with as good performance as he would have got if he implemented parts of the GCLA system himself.

We have also tried another representation of the rules and the strategies in the rule interpreter, where the same compilation scheme is used except for the head of the rule interpreter's heads and the calls to the rule interpreter. Instead the three last arguments of the r-atoms are put last in the rule atoms, i.e. we replace an atom

(i) \( r(\text{rule}(\text{Arg}1, \ldots, \text{Arg}n), \text{Seq}) \)

with the atom

(ii) \( \text{rule}(\text{Arg}1, \ldots, \text{Arg}n, \text{Seq}). \)

in the compilation scheme, and put the rule interpreter clauses in a new module to avoid name clashes with other predicates. One would perhaps expect that the performance would increase with the version (ii), since now indexing is applied to both \( \text{rule} \) and \( \text{Arg}1 \), and not only to \( \text{rule} \). But there is no increase in speed, and that is probably because the rules and strategies are often represented by only one clause, so the extra indexing over \( \text{Arg}1 \) does not circumscribe the choice of clause any further. Also, often the arguments of a rule call are not instantiated, and therefore even if there are several clauses, there is no gain in performance with version (ii). There is also another complication: when a rule is compiled, the proofterm that is going to be used when proving the premises of the rule is unknown at compile time. With version (i), it is easy to build up the r-atom with the three extra arguments in the r-atom, which is done at compile time as a call to \( r/4 \), but for version (ii) we have to build a new structure with an
arity increased by three at runtime, containing the rule's original arguments plus the three extra arguments, and then make a call to the new structure, which is more expensive. Test executions showed that the two versions of the rule interpreter have about the same performance without applying the Prolog compiler, and when the Prolog compiler was applied to the rule interpreter, version (i) was twice as fast as version (ii).

By partially evaluating strategies and replacing all calls to strategies by their unfolded calls to rules instead, efficiency can be improved further. Also, when a proofterm contains more than one function (i.e. its functional expression is nested more than one level) efficiency can be improved by forming a new rule based on the nested expression. This can also be done by partial evaluation of the functional expression, but it is not obvious how to distinguish different arguments of an expression as a proofterm or as data.

Yet another interesting step would be to partially evaluate a control definition with respect to an object level definition, and thus merge the two levels into one single program. It should in principle not be harder than partially evaluating any Prolog interpreter with respect to an object program. So far, we have only been playing with the possibility, but not done any research into the topic.

There are now emerging applications developed with GCLA as a development tool [FW93, ST93, Aro93b], and these applications show that the greater expressiveness of GCLA is useful in the development phase of an application. Together with a good understanding of how to compile GCLA to Prolog, we think that GCLA can be a good alternative as an executable specification language for the development of KB systems.

References


