SICS/R-89/8905B

The Programming Language GCLA: A Definitional Approach to Logic Programming

by
Martin Aronsson Lars-Henrik Eriksson
Anette Gäredal Lars Hallnäs Peter Olin

SICS research report
R89005B
ISSN 0283-3638

SICS/R-89/8905B

The Programming Language GCLA — a Definitional Approach to Logic Programming

Martin Aronsson Lars-Henrik Eriksson Anette Gäredal*
Lars Hallnäs† Peter Olin

November 20, 1989

Swedish Institute of Computer Science (SICS)
Box 1263, S-164 28 Kista, SWEDEN
Phone: +46 8 752 15 00
Telefax: +46 8 751 72 30
E-mail: martin@sics.se

Abstract

We present a logic programming language, GCLA [1] (Generalized horn Clause LAnguage), that is based on a generalization of Prolog. This generalization is unusual in that it takes a quite different view of the meaning of a logic program – a "definitional" view rather than the traditional logical view.

GCLA has a number of noteworthy properties, for instance hypothetical and non-monotonic reasoning. This makes implementation of reasoning in knowledge-based systems more direct in GCLA than in Prolog. GCLA is also general enough to incorporate functional programming as a special case.

GCLA and its syntax and semantics are described. The use of various language constructs are illustrated with several examples.

*Anette Gäredal's present address is: SEB Data, Sergels Torg 2, S–106 40 STOCKHOLM, SWEDEN.

†Lars Hallnäs' present address is: Department of Computer Sciences, Chalmers University of Technology, S–412 96 GÖTEBORG, SWEDEN.

[1] To be pronounced "Gisela".

1 Introduction

Programming languages today could be divided into three different categories: imperative languages (such as C, Pascal etc.), functional languages (ML, Miranda etc.) and logic languages (Prolog etc). In the first category execution of a program leaves the computer (i.e. the memory) in a certain state giving the value of the execution. In the second category an execution of a program amounts to evaluating an expression to a canonical value which is the result of the execution. Executing a logic program means constructing a substitution ("binding" variables to certain values) such that a certain goal is true according to the program under the given substitution.

During the past years logic programming has proved a good framework for writing programs with high expressive power while still providing the ability to execute programs efficiently. The key to this combination is the Prolog language, which uses definite Horn clauses as programs, and resolution as computation rule. Extensions to Prolog have been proposed by several authors, e.g. Gabbay [5], Miller [11] and Naish [12], most extending the language to larger subsets of logic. In contrast, we propose the language of *generalized Horn clauses* (GCLA) as a "non-logical" extension to Prolog. Generalized Horn clauses are a generalization of definite Horn clauses, not within the traditional logical framework, but rather within the framework of *partial inductive definitions* described by Hallnäs and Schroeder-Heister [3], [7], [8], using a more primitive proof-theoretic view of Horn clause programming. As a programming language, GCLA is best regarded as belonging to the group of logic programming languages, but with some properties usually found among functional languages.

The use of (ordinary) inductive definitions in logic programming has been proposed earlier by Hagiya and Sakurai [6]. In contrast to our work, they did not use inductive definitions as the basis for a programming language, but rather as a logic for reasoning about Prolog programs. [2]

Partial inductive definitions gives a definitional framework which, in a sense, is of a more primitive and general nature than a standard logical framework. In GCLA, a program is not looked upon as a set of true facts and implications from which conclusions are drawn using a computation rule, but rather as a definition of rules determining the possible inferences. One could say that in GCLA the program forms the "system's mind", meaning that the inference rules of the "logic" are given by the program. This contrasts with Prolog where the inference rule is given *a priori* and the program is executed within this given framework.

To execute the program, we pose a query to the program by presenting a goal G and ask whether there is a substitution σ such that $G\sigma$ holds according to the logic defined by the program. As we pose a query to the program the system tries to construct a deduction showing that $G\sigma$ holds in the given logic.

[2] Those familiar with the theory of inductive definitions may note that the main theoretical difference between ordinary and partial inductive definitions is that the operator associated with a partial inductive definition is not necessarily monotone (see [7]).

GCLA's generalization of Horn clause programming includes hypothetical and non-monotonic reasoning as integral parts. This makes it easy to handle hypothetical queries, negation and AI techniques like non-monotonic reasoning, simulation and planning, in a natural way. GCLA can be given a query which includes assumptions, or queries can be made where GCLA is asked what assumptions must be made to make a goal true.

Example 1

To illustrate these basic properties of the language, let us consider an example: Assume we are going to construct a system for the diagnostics of infectious diseases. A formalization of general knowledge in this domain will be the base of our program. We could write a program which defines different infectious diseases by listing typical symptoms. Part of the code[3] could look like:

```
disease(cold) :- symptom(cough),temp(normal).
disease(pneumonia) :-
     symptom(persistent_cough),
     symptom(chill),
     temp(high).
```

The expression $C : -A, B$ should be read as "if A and B hold, then C will hold by definition."

A goal consists of two parts: one conclusion and a number of assumptions. When posing a query to the program we are intuitively asking whether the conclusion follows from the assumptions in the given program. If there are no assumptions, we are simply asking if the conclusion is true according to our program. A typical situation here is when we have some symptoms and want to ask about a possible diagnosis. In this case we could pose a query like

```
symptom(cough),temp(normal) |- disease(X).
```

to the program. $A \vdash B$ should be read "B follows from A", so the given expression would when interpreted as a query mean that we are asking for a value of `X` such that the conclusion under this substitution will follow from the assumptions, i.e. what possible infectious disease has as typical symptoms coughing and normal temperature. In this case the system would respond with `X=cold`.

We might also be in a situation where we do not have complete knowledge of the symptoms and want to get some clues for further investigations.

```
symptom(chill),symptom(persistent_cough),temp(X) |-
     disease(Y).
```

[3]To make this program run under an actual GCLA interpreter, the predicates `temp` and `symptom`, must be declared "total" in the sense of section 3.5

Here the system would respond with `X=high,Y=pneumonia`.

A special feature of logic programming languages is the fact that there can be several correct answers to a posed query. This means that if we are interested in answer substitutions other than the first, backtracking will give us those when requested. For instance, assume that we want to list the typical symptoms of a given infectious disease. This could be done by posing the following query:

```
disease(pneumonia) |- X.
```

The first (trivial) answer would be `X=disease(pneumonia)`, as the assumption follows trivially from itself. The interesting answers follows, first `X=symptom(persistent_cough)`, then `X=symptom(chill)`, and finally `X=temp(high)`.

Example 2

It is important to understand that there is an essential conceptual difference between assumptions in a hypothetical query and adding new definitional clauses to the program. The program itself is a definition, it presents a world so to speak. Adding new clauses to such a definition could mean that we change the given world, on the other hand when we assume something to hold according to a given definition this does not mean that we are thereby changing the definition. Let us consider a simple example to illustrate this point:

```
cold_weather :- snow.
```

So we have a world in which snow by definition gives cold weather. Thus assuming cold weather, there must be snow (as there is no other possible cause for the cold weather than snow). But if we changed the world to one where cold weather also holds unconditionally

```
cold_weather.
cold_weather :- snow.
```

it would no longer be possible to derive that it must be snowing in the new world (since the weather now is cold without the explicit cause of snow).

Using the `def` construct described in the next example, local additions of clauses to the program can be done as the program is running. Such additions would change the "world" and, as the present example shows, makes it possible to directly perform non-monotonic reasoning with GCLA.

Example 3

Now assume we would like to think of a certain program as describing general properties of a family of worlds. When posing queries to the program we add

local information that specializes the query to a specific world in the given family.

Suppose the following clause was added to the program of example 1.

```
antibiotic(penicillin) :-
     disease(pneumonia),
     (exception(penicillin) -> false).
```

The construction `exception(penicillin) -> false`, meaning "falsity follows from `exception(penicillin)`", i.e. "`exception(penicillin)` is false" is the way of expressing negation in GCLA [4].

This clause gives part of a definition of what type of antibiotic should be given to a patient. The clause says that if someone is suffering from pneumonia and there is no exception concerning penicillin noted, then he should be given penicillin as treatment. We could use this definition, and locally add a list of exceptions:

```
def(exception(penicillin),disease(pneumonia)) |-
      antibiotic(X).
```

The `def` construct means that `exception(penicillin)` is added to the program as a fact during evaluation of the query

```
disease(pneumonia) |- antibiotic(X).
```

to which the system would respond with `no`. If we instead had posed the following query

```
exception(penicillin),disease(pneumonia) |-
       antibiotic(X).
```

the first answer would have been `X=penicillin`, which would be incorrect.

When using `def` we reason on basis of the fact that there is an exception noted. In the second case we *assume* that there is an exception in the given world. Since the negation construct `exception(penicillin) -> false` is true whenever `exception(penicillin)` is undefined, simply assuming the exception will not make the negation false, as it is still not defined by the program.

In cases like Example 1 where we are simply trying to show that things are true it is possible to represent known facts as assumptions in a query.

This situation illustrates rather well what it means to say that a program in GCLA is a definition rather than a set of formulas in a globally defined logic. The meaning of "follows from" ($\vdash$) in a query will depend on the particular program you pose the query to. As a programming language

[4] "Falsity" could be any symbol that is not defined in the given program. This particular example will behave like negation as failure in Prolog, but is really just an instance of the general scheme for evaluating hypothetical queries.

GCLA is in a sense more primitive than conventional logic programming. It gives a framework for writing definitions of a certain sort just as a functional language like ML gives a framework for writing function definitions.

These examples show that GCLA holds promise to be a suitable language for building expert systems. For instance, in a rule based system implemented in GCLA, the rules would actually form the logic with which GCLA would reason. In queries to the expert system assumptions about the domain can be given, and this makes it possible to ask hypothetical queries, do simulations, etc.

Planning problems like the blocks world can also be expressed in GCLA. The rules for how and when a block can be moved are given as a program. Given a query with an initial state and some actions, the resulting state can be derived. If given two states, a plan containing actions necessary for reaching the second from the first can be generated. This is possible without writing any additional program or interpreter for running the different kinds of queries. The general execution mechanism of GCLA supports the expression of these queries from the beginning.

It also turns out that functional programs can be expressed in GCLA. Thus GCLA integrates logic and functional programming in a single framework. The equations defining pure functional programs are written as Horn clauses and used as the GCLA program, making the GCLA system perform execution steps corresponding to reductions of a functional expression. To find the value of an expression, the expression is assumed and the system is asked what conclusion can be drawn from it. During execution, the expression is successively reduced until the answer has been found.

To demonstrate the feasibility of GCLA, a number of interpreters have been implemented [1], and an abstract machine has been designed. The abstract machine, called GAM (GCLA Abstract Machine) [2], is developed from the WAM [14]. However, a number of instructions in GAM have different definitions than in WAM, and there is also a number of new instructions. A compiler for translating GCLA programs into GAM instructions is under construction, and an emulator for GAM has been written. This experimental work suggests that GCLA programs can be run efficiently. Preliminary results show that when definite Horn clause (pure Prolog) programs are run using GCLA, the loss of efficiency caused by the greater generality of GAM amounts to a 50% increase in run time compared to a WAM implementation of Prolog.

We will in the following sections give an overview of the basic architecture of the programming language GCLA and give examples of different applications. The language description will thus be largely informal. All in all, this is not a manual for the language, but a description of the basic architecture of GCLA together with some illustrative examples.

2 Syntax

Syntactically GCLA is very similar to Prolog. A *clause* is written as

$$H :- [G_1, \ldots, G_n], B_1, \ldots, B_m$$

where H is called the *head* and $B_1, \ldots, B_m$ the *body*, and each B_i is called a *condition*. $[G_1, \ldots, G_n]$ is called the *guard*. Just as in Prolog, the head and the conditions consists of a predicate symbol applied to some number of terms. We say that this clause *defines* H. Again, as in Prolog, the $:-$ sign can be omitted if the body is empty.

In addition, a condition can be hypothetical, i.e. of the form $(A_1, \ldots, A_n \rightarrow C)$ where the A_i:s and C are themselves conditions, called *assumptions* and *conclusion*, respectively. Although such a condition is closely related to logical implication (as its syntax suggests) it does not, in general, have the meaning of implication.

The purpose of the guard is to determine the applicability of a clause according to meta-logical criteria. Each G_i is a call to a built-in predicate, such as `var`, `nonvar` etc[5]. If the guards are not all true when a clause is to be used, the clause is ignored. Such calls can not be inserted in the body as in Prolog, since the semantics of GCLA do not permit the body to be read simply a sequence of procedure calls. An empty sequence of guards ([ ]), can be omitted.

A *goal* is also different from Prolog's goals since GCLA permits hypothetical queries. A goal comprises, analogue to conditions, assumptions and a conclusion and is written as $A_1, \ldots, A_n \vdash C$. This should be read C *follows from* $A_1, \ldots, A_n$ *according to* the program. A general term for the construction $A_1, \ldots, A_n \vdash C$ is a *sequent*. In this article (but not in programs), we use the symbols Γ and Δ among the assumptions to represent zero, one, or several unspecified assumptions.

3 Semantics

3.1 Introduction

A (pure) Prolog program is executed by repeatedly applying the resolution rule to the current goal and some clause from the program. During execution a variable substitution is constructed, so that the initial goal, with the constructed substitution applied, is a logical consequence of the program.

Similarly, during GCLA execution, a substitution is constructed so that the initial goal, with the constructed substitution applied, *holds according to the program* in the calculus of partial inductive definitions. Generally, this *answer substitution* is the desired result when executing a GCLA program.

The process of constructing the substitutions is considerably more complicated in the GCLA case than in the Prolog case. To explain the semantics

[5]Actually, in the current implementation, guards are just Prolog goals.

of GCLA, we will begin by explaining what it means for a goal to hold according to a program in the calculus of partial inductive definitions. Understanding GCLA in this "static" way corresponds roughly to understanding the declarative semantics of a Prolog program.

Once this is understood, the "dynamic" aspects of GCLA semantics, i.e. what actually happens during execution of a GCLA program can be explained. These aspects can be compared with the procedural semantics of Prolog.

Even though the role of the logical variable is essentially the same in GCLA and in Prolog, its meaning is quite different. In Prolog, variables in clauses correspond to universally quantified variables in logic and variables in goals correspond to existentially quantified variables in logic. The theory of partial inductive definitions does not have the concept of a variable, so in GCLA variables have no such logical meaning. Instead, they are best seen as a meta-logical device.

In a clause, such as $p(X)$:-$q(X)$, the variables stand for arbitrary ground terms. In other words, the declarative meaning of a clause is the same as the set of clauses obtained by substituting every possible ground term (of the Herbrand universe) for the variables in the clause. Clauses containing variables are to be regarded as a shorthand notation for such sets of ground clauses.

In a goal with variables, e.g. $\vdash p(X)$, the variables are placeholders that will be substituted by some terms ("bound") during execution. The case where execution succeeds without any term being substituted for a variable, should be interpreted in such a way that the goal holds when any ground term is substituted for the variable in question.

In Prolog, this meta-logical reading of variables coincides with the logical reading — the "Lifting lemma" in Lloyd [10] is an example of this. In GCLA, however, there is no logical reading in the traditional sense. This interpretation of variables in GCLA is the key to the connection between the declarative and operational semantics of GCLA and must be kept in mind.

For a more complete and detailed treatment of the semantics of generalized Horn clauses, see Hallnäs and Schroeder-Heister [8].

3.2 Declarative semantics

The declarative semantics of GCLA are precisely those of the partial inductive definitions by Hallnäs [3],[7].

The declarative semantics will be given as a number of inference rules and an axiom schema. Here we will regard the assumptions of a sequent as forming a set. Thus we assume no particular order among assumptions, and assumptions may be duplicated freely. This is in contrast to the operational semantics where the order of assumptions reflect the order in which different parts of the program will be executed. Another difference from the operational semantics is that the *def* and *rem* constructs, being essentially operational in nature, are not described.

The proper inference rules are divided into two groups. The first two deal with the meaning of hypothetical conditions ("arrow rules"), the second two deal with the inferences determined by the program ("program rules").

3.2.1 The axiom schema

The axiom schema is

$$\Gamma, C \vdash C$$

i.e. something always holds if it is assumed. A sequent of this form is usually called an *initial sequent*.

3.2.2 Arrow rules

$$\frac{\Gamma, A_1, \ldots, A_n \vdash C}{\Gamma \vdash A_1, \ldots, A_n \rightarrow C}$$

$$\frac{\Gamma \vdash A_1 \quad \ldots \quad \Gamma \vdash A_n \qquad \Gamma, C \vdash C'}{\Gamma, (A_1, \ldots, A_n \rightarrow C) \vdash C'}$$

These rules give the semantics of the arrow. The first rule, *arrow-right*, states that a hypothetical condition $A_1, \ldots, A_n \rightarrow C$ holds if C holds with the additional assumptions $A_1, \ldots, A_n$. The second rule, *arrow-left*, states that a hypothetical assumption can be replaced by its conclusion, if the premises of the hypothetical assumptions all hold.

These rules are structurally identical to the rules for implication in the ordinary sequent calculus of logic. Again, it must be emphasized that the arrow does not in general have the same meaning as logical implication. Depending on the form of the program, it might, or it might not. For details see Hallnäs [3] or [7].

3.2.3 Program rules

The two program rules define what it means for a goal to hold "according to a program". Since the exact form of the inference rules depend on what the program is, we are justified in saying that the program actually defines the "logic" used to derive the goal.

$$\frac{\Gamma \vdash C_1 \quad \ldots \quad \Gamma \vdash C_n}{\Gamma \vdash C}$$

where the clause $C :- C_1, \ldots, C_n$ is in the program.

$$\frac{\Gamma, A_{11}, \ldots, A_{1n_1} \vdash C' \quad \ldots \quad \Gamma, A_{k1}, \ldots, A_{kn_k} \vdash C'}{\Gamma, C \vdash C'}$$

where the $C :- A_{i1}, \ldots, A_{in_i}$ are all clauses in the program with head C.

The first rule, *program-right*, simply states that if there is a clause $C :- C_1, \ldots, C_n$ defining C, then C holds according to the program if each C_i holds according to the program. The rule can be understood as simply

replacing C with the conditions of its definition. It is essentially the ground form of the SLD-resolution rule used in Prolog.

The second rule, *program-left*, is more complicated. It is also the most important rule, in the sense that most of the additional power of GCLA over Prolog comes from this rule.

Suppose we want to show that C' holds under some assumptions, including C. Since we assume that C holds, we must also assume that there is some clause $C :- A_1, \ldots, A_n$ defining C, where all the conditions A_i hold — since nothing can hold without being defined. Thus it would be permissible for us to try to show that C' holds under the assumptions A_i. In a sense, we have replaced the assumption C by its definition.

The situation is more tricky when there is more than one way for C to be defined, i.e. more than one clause has C as its head. In that case the body conditions need hold in only one of the clauses. Since we do not know which clause this is, we must try to show that C' holds using the body of each clause defining C. We have now replaced C by each possible definition. This is very similar to the situation in logic where we must show both $A \vdash C$ and $B \vdash C$ to show that $A \vee B \vdash C$.

A special case is where C is not defined by any clause at all. In this case C can never hold, so it is absurd to assume it. In this case the inference rule gets no premises, so its conclusion holds immediately by contradiction. This case is essentially the negation by failure rule in Prolog.

From these rules the important difference between assumptions and program in GCLA are obvious. The program determines the exact form of inference rules, assumptions do not. While assumptions are facts that we assume to hold, clauses are really not facts at all, but relations between facts.

3.2.4 Examples

Consider the following program P.

```
a :- a.
b(1) :- a.
b(Z) :- c.
```

We will give a derivation to show that $a \rightarrow b(1)$ holds according to the program P. This derivation starts with an axiom, then uses the program-right rule, and finally the arrow-right rule.

$$\dfrac{\dfrac{a \vdash a}{a \vdash b(1)}}{\vdash a \rightarrow b(1)}$$

To show that $\{X/1\}$ is a correct answer substitution for the goal `b(X)|-a`, we give a derivation of $b(1) \vdash a$.

$$\frac{a \vdash a \qquad \overline{c \vdash a}}{b(1) \vdash a}$$

The sequent $a \vdash a$ is an axiom. The inference step without premises is an instance of the program-left rule. Since c is not defined in the program, the sequent $c \vdash a$ holds according to this rule without any premises.

Here the last step uses the program-left rule. To motivate it, consider that the clause `b(Z) :- c`, should be read as the set of clauses formed by all possible instantiations of Z by a ground term. Thus $b(1)$ in the goal occurs in the head of exactly two clauses, the second clause of the program, and the instance of the third clause where 1 is substituted for Z. The inference has two premises, each with assumptions coming from the body of one of these two clauses.

3.3 Operational semantics

The operational semantics will also be described as a series of inference rules. The main difference from the inference rules above is that the present rules work on *contexts*, consisting of an ordered set (list) of sequents, a substitution and a program. By using a list of sequents, it is possible to write the rules with only one premise, making the inference rules correspond to state transitions[6].

The inclusion of a substitution in the context makes it explicit how an answer substitution is constructed as the program executes. Likewise, the explicit inclusion of a program permits definition of the *def* and *rem* constructs that manipulates the running program.

To execute a program, we start with a context consisting of a list containing the goal only, an empty substitution, and the program itself. Inference rules are then applied successively in a backward (goal-oriented) fashion, each time giving a new context. As execution proceeds, substitutions will be accumulated in the context. Provided the program does not loop, we will eventually obtain an "initial context" with an empty list of sequents. As this means there are no more subgoals, the execution stops. The substitution in this initial context is the desired answer substitution. In other words, the sequent resulting from applying this substitution to the original goal will hold according to the program in the declarative semantics.

Contexts will be written as

$$\Sigma \qquad \langle \theta, P \rangle$$

where Σ is a list of sequents, θ is a substitution and P the current program. Lists of sequents will be built up using the . (dot) operator, the empty list will be denoted by $\emptyset$

[6]The reason we write the rules as inference rules rather than state transitions is to maintain the similarity with the inference rules of the declarative semantics

Sequents will be written as in the previous section. This time, however, the assumptions of a sequent are assumed to be ordered, and can not be duplicated freely.

Our notation for substitutions will be the usual one. A substitution is a set of bindings $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable and each t_i is a term. If E is some expression and σ a substitution then $E\sigma$ is the expression obtained from E by simultaneously substituting each occurrence in E of every x_i by the corresponding t_i. $\theta\sigma$ is the composition of the substitutions θ and σ, i.e. $(E\theta)\sigma = E(\theta\sigma)$. As usual, $mgu(A, B)$ denotes the most general unifier of A and B.

A *guard* is interpreted as a built-in propositional function. So if G is a guard and σ a substitution we assume it is defined what it means for $G\sigma$ to be true.

3.3.1 Initial context

The *initial contexts* are the axioms of the operational semantics. They all have an empty set of sequents, so no inference rules are applicable to them,

$$\frac{}{\emptyset \qquad \langle\theta, P\rangle}$$

for any θ and P.

3.3.2 Initial sequent

$$\frac{\Sigma\sigma \qquad \langle\theta\sigma, P\rangle}{(C, \Gamma \vdash C').\Sigma \qquad \langle\theta, P\rangle}$$

where σ is a mgu of C and C'.

This rule corresponds to the case where a subgoal is an initial sequent.

3.3.3 Arrow rules

$$\frac{(A_1, \ldots, A_n, \Gamma \vdash C).\Sigma \qquad \langle\theta, P\rangle}{(\Gamma \vdash A_1, \ldots, A_n \to C).\Sigma \qquad \langle\theta, P\rangle}$$

$$\frac{(\Gamma \vdash A_1). \ \ldots \ .(\Gamma \vdash A_n).(C, \Gamma \vdash C').\Sigma \qquad \langle\theta, P\rangle}{((A_1, \ldots, A_n \to C), \Gamma \vdash C').\Sigma \qquad \langle\theta, P\rangle}$$

3.3.4 Program rules

$$\frac{(\Gamma\sigma \vdash A_1\sigma). \ \ldots \ .(\Gamma\sigma \vdash A_n\sigma).\Sigma\sigma \qquad \langle\theta\sigma, P\rangle}{(\Gamma \vdash C').\Sigma \qquad \langle\theta, P\rangle}$$

where $C :- [G_1, \ldots, G_m], A_1, \ldots, A_n$ is a clause in P, σ is an mgu of C and C' and $G_1\sigma, \ldots, G_m\sigma$ are all true.

$$\frac{((A_{11}, \ldots, A_{1n_1}, \Gamma \vdash C'). \ \ldots \ .(A_{k1}, \ldots, A_{kn_k}, \Gamma \vdash C').\Sigma)\sigma \qquad \langle\theta\sigma, P\rangle}{(C, \Gamma \vdash C').\Sigma \qquad \langle\theta, P\rangle}$$

where σ is a C-sufficient substitution as explained below, and the clauses $C_i : -[G_{i1}, \ldots, G_{im_i}], A_{i1}, \ldots, A_{in_i}$ are all clauses in P such that $C_i\sigma = C\sigma$ and $G_{i1}\sigma, \ldots, G_{im_i}\sigma$ are all true.

The algorithm used by GCLA to compute a C-sufficient substitution is as follows:

Let $C_1, \ldots, C_n$ be a permutation of the heads of all program clauses in P and let

$$mgu'(C, C') = \begin{cases} mgu(C, C') & \text{if it exists} \\ \emptyset & \text{otherwise} \end{cases}$$

Then define

$$\begin{aligned} \sigma_0 &= \emptyset \\ \sigma_{m+1} &= \sigma_m mgu'(C\sigma_m, C_{m+1}) \\ \sigma &= \sigma_n \end{aligned}$$

Note that this algorithm considers *all* clause heads of the program. Those clauses that are not applicable for the rule is rejected by the condition $C_i\sigma = C\sigma$ above.

In order for the last rule to be a correct implementation of the program-left rule of the declarative semantics, an additional condition must be imposed. This condition requires that none of the clause bodies, $A_{i1}, \ldots, A_{in}$, contain variables that do not also occur in the corresponding clause head, C_i. This condition is similar to the condition on negation as failure in Prolog that negative goals are all ground when invoked.

It would be possible to avoid this restriction by introducing explicit quantification. Another alternative could be to omit the check altogether, leaving it to the programmer to ensure that no new variables are introduced by some application of the last rule or that any variables introduced will behave as the programmer intends.

The operational semantics of the program-left rule are incomplete in the sense that there are cases where GCLA is unable to find substitutions that would make the goal hold according to the program, given the declarative semantics.

The soundness of the rules of the operational semantics in implementing the declarative semantics is proved by Hallnäs and Schroeder-Heister in [8].

3.3.5 Examples

We will repeat the examples of the last section, this time using the rules of the operational semantics.

Recall that P was the following program

```
a :- a.
b(1) :- a.
b(Z) :- c.
```

To solve the goal `|- a->b(1)` we obtain the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset}{(a \vdash a).\emptyset}\quad\langle\emptyset,P\rangle}{(a \vdash b(1)).\emptyset}\quad\langle\emptyset,P\rangle}{(\vdash a \rightarrow b(1)).\emptyset}\quad\langle\emptyset,P\rangle}{}$$

The goal `b(X)|-a` may be solved in the following manner.

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset}{(c \vdash a).\emptyset}\quad\langle\{X/1,Z/1\},P\rangle}{(a \vdash a).(c \vdash a).\emptyset}\quad\langle\{X/1,Z/1\},P\rangle}{(b(X) \vdash a).\emptyset}\quad\langle\emptyset,P\rangle}{}$$

The bottommost step (the one executed first), is an instance of the program-left inference rule, with $\{X/1, Z/1\}$ as a $b(X)$-sufficient substitution. In the uppermost step (the one executed last), also an instance of the program-left rule, there is no c-sufficient substitution (since there is no clause defining c), so the set of clauses selected is empty.

The initial context at the top of the derivation has the substitution $\{X/1, Z/1\}$, which is an answer substitution for the goal. (In fact $\{X/1\}$ is a sufficient answer substitution, since Z does not occur in the goal.)

3.3.6 Def and rem rules

We will now give the rules defining the *def* and *rem* operations mentioned in example 3 of the introduction. The intentions of these operations are to make local modifications to the program by adding or removing clauses.

The construction *def(K,C)* executes the condition C in a context where the clause K has been added to the program. Similarly, the construction *rem(K,C)* executes the condition C in a context where the clause K has been removed.

$$\frac{(\Gamma \vdash C).\Sigma \qquad \langle\theta, P \cup \{K\}\rangle}{(\Gamma \vdash def(K,C)).\Sigma \qquad \langle\theta,P\rangle}$$

$$\frac{(C,\Gamma \vdash C').\Sigma \qquad \langle\theta, P \cup \{K\}\rangle}{((def(K,C)),\Gamma \vdash C').\Sigma \qquad \langle\theta,P\rangle}$$

$$\frac{(\Gamma \vdash C).\Sigma \qquad \langle\theta, P \setminus \{K\}\rangle}{(\Gamma \vdash rem(K,C)).\Sigma \qquad \langle\theta,P\rangle}$$

$$\frac{(C,\Gamma \vdash C').\Sigma \qquad \langle\theta, P \setminus \{K\}\rangle}{(rem(K,C),\Gamma \vdash C').\Sigma \qquad \langle\theta,P\rangle}$$

When removing clauses using the *rem* rules, clauses will be considered equal if they are variable renaming variants of each other.

Note that the *def* rules are written so that not only the sequent containing the *def*, but also the sequents in Σ are executed in a context where the clause K has been added. The alternative formulation, where only the sequent

containing *def* is executed in a context where K has been added, would also be possible and, perhaps, more natural. However, the present rules reflect how GCLA is currently defined.

3.3.7 Contraction

In the declarative semantics, assumptions could be duplicated freely. To reduce the search space of a GCLA implementation, this is not permitted in the operational semantics. Once an assumption has been used, it is not available any more. Sometimes there is a need to duplicate assumptions (commonly called "contraction" in logic). To permit this the special *contr* construct is used. Writing $contr(N, C)$ as an assumption, permits the condition C to be used as an assumption N times. As a conclusion, the construction is equivalent to simply writing C.

$$\frac{(\Gamma \vdash C).\Sigma \qquad \langle \theta, P \rangle}{(\Gamma \vdash contr(N, C)).\Sigma \qquad \langle \theta, P \rangle}$$

for any N.

$$\frac{(\Gamma \vdash C').\Sigma \qquad \langle \theta, P \rangle}{(contr(0, C), \Gamma \vdash C').\Sigma \qquad \langle \theta, P \rangle}$$

$$\frac{(C, contr(N-1, C), \Gamma \vdash C').\Sigma \qquad \langle \theta, P \rangle}{(contr(N, C), \Gamma \vdash C').\Sigma \qquad \langle \theta, P \rangle}$$

where $N > 0$.

3.4 Control issues

When attempting to show that a certain sequent holds, it is, in general, possible to use different inference rules, or to use one inference rule in different ways. These alternatives introduce nondeterminism in programs in a similar way as in Prolog.

By default, program clauses are ordered top down. Sequents, and assumptions within sequents, are ordered from left to right. Whenever different clauses, sequents or assumptions are applicable, the topmost/leftmost will be used first. As in Prolog, a choice point is made, so that the next one will be tried, should computation fail and backtrack.

Likewise, the inference rules are ordered so that the initial sequent rule is tried first, then the arrow-right or program-right rule (depending on the form of the condition to the right of the turnstile), and finally the arrow-left or program-left rules.

This standard left-to-right order might not only be inefficient, but might also cause some programs to loop. A basic problem is how to determine and specify an execution order that is suitable for a particular program. GCLA currently provides some simple primitives to change the selection order [1].

We are presently working on a more flexible method to provide control information. The basic idea is to separate the control structure from the actual program, using special control clauses ("operators") to guide the execution of the program.

One particular control primitive needs to be mentioned, since it is used by most examples. Often it is desired that the program says nothing about whether a term holds or is absurd. Simply not defining the term in the program is not sufficient, as it implies that the term is absurd. Instead a clause of the form

```
p :- p.
```

is used. Now p is defined, so it cannot be absurd. On the other hand, a circular definition such as this gives no information that can be used to show that the term holds, so the desired effect is reached. Unfortunately, a clause such as this will cause a depth-first interpreter, such as the current GCLA interpreter, to loop.

Instead, a control primitive is used to declare p "total"[7]. The effect is exactly as if the clause above was included in the program, but the interpreter will not loop.

4 Applications and examples

Throughout this section, we assume that all programs include the clauses

```
X=X.

P;Q :- P.
P;Q :- Q.

and(P,Q) :- P,Q.

not(X) :- X -> false.
```

The first clause defines simple equality. The next two clauses define disjunction, and the fourth conjunction. This is all just as in Prolog. The final clause defines negation (to be explained shortly).

Although all programs will run on a GCLA system as they stand, additional control information is needed for efficient execution and/or to avoid duplication of answers. We omit this here, as it would obscure the examples and since the control primitives themselves are an active research topic and subject to major revision.

[7]The terminology is due to [7].

4.1 Negation

The following program illustrates the use of negation in GCLA. This example also shows that GCLA in many cases permits constructive negation.

```
elephant(clyde).
elephant(fido).
elephant(P) :- albino_elephant(P).

albino_elephant(karo).

grey(P) :- elephant(P), (albino_elephant(P) -> false).
```

The first two clauses state that Clyde and Fido are elephants. The third clause states that all albino elephants are elephants. In the fourth clause Karo is defined to be an albino elephant.

What constitutes a grey object in this universe? It should be an elephant that is not an albino elephant, i.e. an object such that assuming it leads to a contradiction. The construction `albino_elephant(P) -> false` expresses this. The symbol `false` is simply some symbol that is never defined. The clause defining `not` in the beginning of this section is a general case of this construction.

Some examples: The goal `|- grey(E)` has the two answer substitutions `E=clyde` and `E=fido`.

The goal `|- not(grey(E))` has the single answer substitution `E=karo`.

Finally, the goal `|- not(not(grey(P))` has the two answer substitutions `E=clyde` and `E=fido`.

4.2 A simple expert system

A knowledge based system is a system which bases its reasoning on domain knowledge to find solutions to specific problems in a domain. Characteristic of such domains is that the reasoning in the domain cannot easily be formulated as algorithms. Rather the knowledge used in the reasoning is represented as rules, frames, semantic nets, etc. When it comes to implementing such a system many languages have been used; expert system shells, AI languages (e.g. OPS5, Prolog and Lisp) and traditional programming languages. For further reading on this subject see for instance the book by Hayes-Roth *et.al.* [9].

A common problem with shells is their rigidity; if the domain doesn't fit perfectly within the limitations of the shell there is no way to make it fit except modifying the domain. A common limitation is that the shell only provides one inference rule, and that it doesn't allow definitions of new inference rules.

Compared to Prolog, GCLA is better suited for implementation of expert systems since in Prolog you are required to write a meta-interpreter to

handle different inference rules, while in GCLA many of these rules are easily expressed in the language itself.

We will illustrate how GCLA can be used to implement a small expert system for the classification of mushrooms, and we will also show how to ask hypothetical and ordinary questions. The expert system is formulated in ordinary if-then rules forming the logic in which GCLA reasons. The way we formulate our queries determines how GCLA interprets and computes the answers.

The knowledge of mushrooms is expressed as rules where each known mushroom is described in terms of different characteristics. For each class of mushrooms, certain characteristics should be included. These characteristics are represented as `c(attribute,value)`.

The strategy for how to classify is given as definitions of those characteristics. A mushroom is classified in one of the following classes: agaric if it has disks, bolete if it has reeds. The strategy is expressed as follows:

```
mushroom(X):-
     bolete(X);
     agaric(X).

agaric(X):-                          bolete(X):-
     c(underneath_hat,disks),             c(underneath_hat,reed),
     type_of_agaric(X).                   type_of_bolete(X).
```

To find out of what type a particular mushroom is, for example an agaric, we have to examine the mushrooms listed as agaric. The characteristics of the mushroom to be classified must correspond to the characteristics in the rule describing a specific agaric.

```
type_of_agaric(amanita_virosa):-
      c(colour_of_spores,white),
      c(colour_of_hat,white),
      (c(form_of_hat,bellshaped);
       c(form_of_hat,eggshaped)),
      c(colour_of_disks,white),
      c(characteristics_of_foot,sock),
      c(characteristics_of_foot,scaly),
      c(smell,musty),
      c(taste,first_mild_then_harsh).

type_of_agaric(agaricus_arvensis):-
      c(colour_of_spores,black),
      (c(colour_of_hat,white);
       c(colour_of_hat,yellow_white);
       c(colour_of_hat,yellow)),
      (c(form_of_hat,bellshaped);
       c(form_of_hat,spread);
```

```
 c(form_of_hat,vault);
 c(form_of_hat,hemispherical)),
(c(colour_of_disks,grey);
 c(colour_of_disks,red);
 c(colour_of_disks,brown)),
c(characteristics_of_foot,ring),
c(smell,anise),
c(habitat,meadow_ground/groves).
```

Additionally, the predicate c, must be declared "total" in the sense of section 3.5.

Among the queries that can be asked in the expert system example, some concerns characteristics of a mushroom. E.g. what does an Amanita Virosa look like? What is the difference between two mushrooms? Which are the similarities between two mushrooms? Etc.

If we want to know what kind of mushroom a set of characteristics describe, we can assume them, posing the hypothetical query

```
c(colour_of_spores,black), c(colour_of_hat,white),
c(form_of_hat,spread), c(colour_of_disks,red),
c(characteristics_of_foot,ring), c(smell,anise),
c(habitat,meadow_ground/groves), c(underneath_hat,disks)
|- mushroom(X).
```

to which the system would respond with `X=agaricus_arvensis`.

Another example is the query

```
mushroom(agaricus_arvensis) |- c(A,V).
```

which would be read as "what characteristics hold if Agaricus Arvensis is a mushroom?" The first answer substitution would be `A=underneath_hat, V=disks)`.

`A` and `V` are instantiated to the first characteristic that describes the Agaricus Arvensis. Here we ask about the knowledge concerning a specific mushroom and not about a specific instance in the program database. The answer to the query is found by examining the rule describing that mushroom. This is what makes GCLA special, the ability to ask this kind of queries without having to write a meta-interpreter to handle them. If we want all the characteristics, we can ask for the rest of the solutions by making use of backtracking, the next solution is:

```
A = colour_of_spores, V = black
```

and so on.

This could of course also be done in Prolog, but then we would have to write a new piece of code for each query.

Other hypothetical queries: What does an Agaricus Arvensis and an Amanita Virosa have in common, i.e. which characteristics are the same for both Agaricus Arvensis and Amanita Virosa?

```
|- and((mushroom(agaricus_arvensis) -> c(A,V)),
       (mushroom(amanita_virosa) -> c(A,V))).

A = underneath_hat, V = disks
```

This gives the first characteristics common to the two mushrooms.

What is the difference between an Agaricus Arvensis and Amanita Virosa, i.e. which characteristics separate Agaricus Arvensis and Amanita Virosa?

```
|- and((mushroom(agaricus_arvensis) -> c(A,V1)),
   and((mushroom(amanita_virosa) -> c(A,V2)),
       not(V1=V2))).

A = colour_of_spores,
V1 = black
V2 = white
```

backtracking yields further differences.

The queries above are hypothetical in the sense that the assumptions for which the conclusion should hold are given in the query itself.

4.3 Functional programming

It also turns out that functional programs can be executed in the framework of GCLA. Thus GCLA integrates logic and functional programming in a single framework. The equations defining pure functional programs are written as generalized Horn clauses and used as the GCLA program, making the GCLA system perform execution steps corresponding to reductions of a functional expression. To evaluate an expression, we use the expression as an assumption when asking the system what conclusions can be drawn from it. During execution, the expression is successively reduced until the answer has been found.

Let's take the add-program below as an example. It defines addition of two natural numbers expressed with the successor function. The first two clauses define the function, and the third defines a substitution schema for substituting the first argument into something else, if at all possible. The clause defining s is also a substitution schema for reducing the argument of the successor function.

```
add(0,X) :- X.
add(s(Y),X) :- s(add(Y,X)).
add(Y,X) :- [Y \= s(_),Y \= 0],((Y -> Z) -> add(Z,X)).

s(X) :- ((X -> Y) -> s(Y)).
```

The constant 0 should be declared "total". Additional control information would be needed to avoid looping search branches if this program is run on an actual GCLA system.

Suppose now that we want to evaluate `add(s(0),s(0))`. The initial goal is to assume the expression, and to put a variable (the result of the calculation) to the right of the turnstile; step 1 in the trace below illustrates this.

```
1.  add(s(0),s(0)) |- X
2.  s(add(0,s(0))) |- X
3.  (add(0,s(0)) -> Z) -> s(Z) |- X
4.  |- add(0,s(0)) -> Z   and   s(Z) |- X
5.  add(0,s(0)) |- Z   and   s(Z) |- X
6.  s(0) |- Z   and   s(Z) |- X
7.  s(s(0)) |- X
```

Step 2 and 3 simply expands the assumption using program-left. In step 4 arrow-left splits the goal into two subgoals. Step 5 uses arrow-right on the left subgoal. In step 6 and 7 initial-sequent is used to give the desired answer `X=s(s(0))`.

This deduction illustrates eager evaluation. If we instead use initial-sequent in step 2 we will get lazy evaluation. So by simply rearranging the order in which the operations are tried we can determine if lazy or eager evaluation should be used.

4.4 Planning

In planning the main goal is to find a sequence of actions that leads to a desired state. There are many different approaches to how this can be done and how to recover from an unsuccessful path during plan generation. Our approach has been to use the strength of GCLA's different deduction strategies to investigate in what ways plans can be generated by proofs.

Planning can be illustrated with the blocks world example. In the blocks world we have a table and three blocks, A, B and C. The blocks world program is centered around the definition of rules. An `action` is `performed` if it is `possible` in the current state. The program consists of the actions applicable in the domain. Below is a definition of the program and the initial state of the world (see Fig. 1). The initial state is

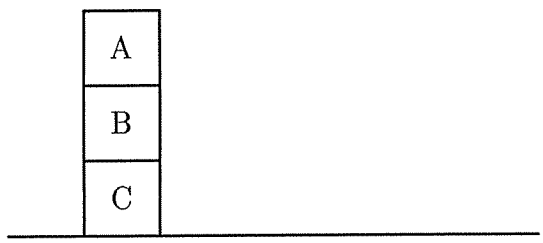

Figure 1: Initial state in Blocks world.

```
on(a,b).    on(b,c).    table(c).    clear(a).
```

and an action is defined as

```
action(E,sit(S)) :- possible(E) -> perform(E,sit(S)).
action(E,action(X,S)) :-
   ((action(X,S) -> sit(Y)) -> action(E,sit(Y))).
```

Here we use the functional programming techniques of the previous section. The second clause of `action` is a substitution schema.

The three actions are defined as `possible-perform` pairs. Recall that `rem(A,B)` means that we remove A from the program when executing B.

```
% STACK moves a block from the table to the top of a stack.
possible(stack(X,Y)):-
     table(X),
     clear(X),
     clear(Y),
     not(X = Y).

perform(stack(X,Y),sit(S)):-
   rem(table(X),
    rem(clear(Y),
     def(on(X,Y),sit(s(S))))).
```

```
% UNSTACK moves a block from the top of a stack to the table.
possible(unstack(X,Y)):-
     on(X,Y),
     clear(X).

perform(unstack(X,Y),sit(S)):-
    rem(on(X,Y),
     def(table(X),
       def(clear(Y),sit(s(S))))).
```

```
% MOVE moves a block from one stack to another.
possible(move(X,Y,Z)):-
     on(X,Y),
     clear(X),
     clear(Z),
     not(X = Z).

perform(move(X,Y,Z),sit(S)):-
   rem(on(X,Y),
    rem(clear(Z),
     def(clear(Y),
      def(on(X,Z),sit(s(S)))))).
```

In addition, the predicate `sit` must be declared "total".

To generate a plan for how to reach the state where C is on top of B (see Fig. 2) , we ask the following question, "is there a plan with three actions that leads to the state where C is on top of B?"

```
action(X, action(Y, action(Z, sit(0)))) |- on(c,b).
```

In this goal, the assumption is a functional expression with undetermined actions (the variables) that is evaluated to the desired state. During this process, the actions are instantiated. `sit(0)` is a representation of the initial state of the program.

Evaluation of the goal results in the answer substitution

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

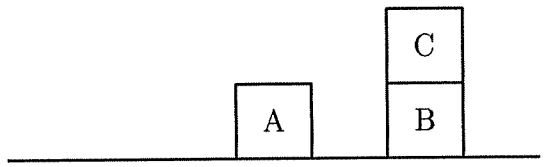

Figure 2: One goal state in the Blocks world example.

Backtracking produces another solution:

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

This example illustrates how GCLA can be used in implementing STRIPS-like planning [13].

4.5 Simulation

Simulation of a task or an action means showing what happens in a real world situation without actually performing the task. Here we look at simulations in the context of proofs. This can be illustrated by using the same example as above, the blocks world. In GCLA we achieve the effect of simulation by posing a certain kind of questions to the program. In contrast to the planning example, the actions are instantiated and we ask about the conclusion. In the planning example the actions were uninstantiated, meaning unknown, and in the question we gave the conclusion, with instantiated arguments, that we wanted to achieve.

A question concerning the state of the world after the occurrence of some actions, i.e. simulating some actions and see the effect. "If we move A from B to the table, and we move B from C to A , what is standing on what?" We assume that the same initial state as in the planning example (see Fig. 1). Posing the query

```
action(move(b,c,a),action(unstack(a,b),sit(0)))
 |- and(sit(s(s(0))), on(X,Y)).
```

to the system gives the answer substitution `X=b,Y=a`.

It should be noted that a query with simply `on(X,Y)` as conclusion could give bindings for `X` and `Y` in any situation encountered by the program, not just the situation after performing the two actions. In particular, the initial state where A is on B could give an answer. To ensure that we get bindings concerning the situation after the two actions only, we require that the second situation, expressed by the term `sit(s(s(0)))` holds, thus the conclusion of the query is a conjunction of these two terms.

When comparing the planning example to the simulation example, GCLA's strength lies in the fact that the same program (definitions) can be used for two different tasks; planning and simulation. The difference is how the questions are instantiated, the instantiation patterns.

5 Conclusions and future work

In conclusion, we have found that several important techniques can be implemented in a natural way using GCLA. However, for certain applications it is necessary to reduce the search space in order to gain sufficient efficiency. We are working on general control primitives to achieve this, and expect GCLA to become an elegant language combining expressiveness with efficiency.

An implementation of GCLA, written in SICStus/Quintus Prolog is available from SICS [1].

We are investigating further various aspects of GCLA. Among these are the ability to execute functional programs and the suitability of GCLA for parallel execution and partial evaluation. Of particular importance is gaining experience with developing production size programs in GCLA.

On the theoretical side, work has been done on using GCLA-like formalisms as type systems for functional programs [4] and as a basis for a framework for formal program development (program calculi) [3].

References

[1] M. Aronsson. The GCLA User's Manual. Technical Report SICS T89012, Swedish Institute of Computer Science, 1989.

[2] M. Aronsson. The Instruction Set for the GCLA Abstract Machine. Research Report SICS R89002, Swedish Institute of Computer Science, 1989.

[3] L.-H. Eriksson and L. Hallnäs. A Programming Calculus Based on Partial Inductive Definitions. Research Report SICS R88013, Swedish Institute of Computer Science, 1988.

[4] D. Fredholm and S. Serafimovski. Partial Inductive Definitions as Type Systems for λ-terms. Master's thesis, Department of Mathematics, University of Stockholm, 1988. Unpublished.

[5] D.M. Gabbay and U. Reyle. N-PROLOG: An Extension of Prolog with Hypothetical Implications: I. *Journal of Logic Programming*, 1(4), 1984.

[6] M. Hagiya and T. Sakurai. Foundation of Logic Programming Based on Inductive Definition. *New Generation Computing*, 2(1):59–77, 1984.

[7] L. Hallnäs. Partial Inductive Definitions. In A. Avron et.al., editor, *Workshop on General Logic, Report ECS-LFCS-88-52.* Department of

Computer Science, University of Edinburgh, 1987. Also published as Research Report SICS R86005C by the Swedish Institute of Computer Science, 1988. A revised version to appear in Theoretical Computer Science.

[8] L. Hallnäs and P. Schroeder-Heister. A Proof–Theoretic Approach to Logic Programming. I. Generalized Horn Clauses. Research Report SICS R88005, Swedish Institute of Computer Science, 1987.

[9] F. Hayes-Roth, D.A. Waterman, and D.B. Lenat. *Building Expert Systems.* Addison-Wesley Publishing Company Inc., 1983.

[10] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[11] D. Miller. A Theory of Modules for Logic Programming. In *Proceedings of the 1986 Symposium on Logic Programming.* IEEE Computer Society Press, 1986.

[12] L. Naish. Negation and Quantifiers in NU-Prolog. In *Proceedings of the Third International Conference on Logic Programming*, pages 624–634, Berlin, 1986. Springer Verlag.

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

[14] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, October 1983.