ISRN SICS-R--92/09--SE

GCLA II

A Definitional Approach to Control

by
Per Kreuger

SICS research report
R92:09
ISSN 0283-3638

SICS R92:09

GCLA II
A Definitional Approach to Control

by

Per Kreuger

piak@sics.se
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

April 1992

Abstract:

This paper describes the logic programming language GCLA II, its operational semantics and parts of its theoretical foundations. GCLA II is a generalization of the language GCLA (Generalized *H*orn Clause *L*anguage) augmented by a method to guide and constrain proof search. The method is based on specification of strategies in a meta language that is a sub language of GCLA itself.

A GCLA II program is partitioned into two distinct parts. One is used to express the declarative content of the program, while the other is used to define the possible inferences made from this declarative knowledge. Although the intended use of the declarative part and the procedural parts are quite different, they can both be described in the formalism of partial inductive definitions. Thus we preserve a declarative reading of the program as a whole. In particular, given certain syntactical constraints on the program, the heuristics associated with proof search does not affect the declarative reading of the declarative part of the program at all.

Several experimental interpreters and a compiler from GCLA II to prolog have been implemented.

Table of Contents

1 Introduction

1.1 GCLA

GCLA (Generalized *H*orn Clause *L*anguage) is a programming language built on a generalization of Horn clauses. Although the language allows both functional and relational style of programming the language is perhaps best described as a logic programming language. The execution mechanism of the language is built around a set of inference rules and proof search heuristics. Search is conducted depth first and indeterminism is handled by backtracking. The computation is initiated with a query to the interpreter in the form of a sequent "$A \vdash C$" and continues with a search for a proof of some instance "$A\sigma \vdash C\sigma$"of the original sequent. The substitution σ is generally regarded as the result of the computation.

We generalize Horn clauses by allowing iteration of the arrow constructor. This means e.g. that $A \rightarrow B \rightarrow C$ can be seen as a clause. The semantics of this generalization is not given in the traditional way as a somewhat larger subset (than Horn clauses) of first order predicate logic, but rather in terms of a more general (low level) theory, i.e. the theory of partial inductive definitions [Hal 91].

The interpretation of a program in this theory do not give us a fixed logical reading of the program, rather the program itself (in a well defined way) generates a logic in which inferences can be made. This means e.g. that the arrow constructor does *not* always correspond to logical implication.

On the other hand the resulting formalism is strong enough to give us a very useful form of negation which is sound for all programs that *do* allow a logical reading and gives bindings for free variables in negative queries. The language also allows us to pose hypothetical questions to a database, to do default reasoning and functional programming in a very natural way.

Although not all programs in this formalism can be understood as a set of axioms in first order predicate logic, various classes of programs *do* correspond to theories expressible in different first order logics e.g. horn clause theories and minimal logic theories.

The formal system of partial inductive definitions makes a distinction between assumptions (occurring in the antecedent of the sequent) and clauses of the definition (the program). The definition (program) parametrizes a set of inference rules that defines a calculus for the system and thus effectively defines the logic in which inferences are made. The antecedent of a sequent, in contrast, play the more traditional role of assumption; similar to that of antecedents in sequent calculus.

A number of other differences make the interpretation of clauses and queries in GCLA quite different from the ones given to various extensions of horn clauses within the context of first order logic. Some of these will be pointed out in part 2 of this paper, while others may be found in one of the papers describing the theory of partial inductive definitions itself (e.g. [Hal 91, HSH 90])

Related work includes [PS 89] which describes an attempt to give a unified count of relational (logic) and functional programing in terms of ordinary (monotone) inductive definitions. The authors of [PS 89] have also designed and implemented a programming language based on these ideas.

As already mentioned the language GCLA is built around a set of inference rules (some of which depend on a given definition/program) that together with some proof search heuristics defines its operational semantics. The heuristics of GCLA make a choice of the order in which to try rules of the calculus when searching for proofs of a given query. This gives the inference mechanism a greater inherent complexity than systems that are based on a single rule (e.g. resolution). In our practical experiments with GCLA as a programing language we have encountered many problems where the lack of an explicit mechanism for guiding search has turned out to be a major problem.

GCLA II is an attempt to construct a general framework where the programmer can define local heuristics for proof search. Part 3 of this paper describes some results of these efforts. Preliminary results of this research were reported as a part of the survey article [GCLA 91]. The main difference from the approach taken in [GCLA 91] is the introduction of strategies, the possibility to modularize and reuse the procedural information as well as a theoretically more coherent and well understood system. For earlier published versions of this paper see [Kre 91].

1.2 The Need for Control

The generality and expressive power of GCLA gives the programmer the freedom to express his problems in a multitude of ways. He may, e.g., choose a functional approach, a relational (logic programming) approach or any combination of these when formalizing his domain. He may want to treat assumptions in the antecedent of sequents as representations of process states, and application of the axiom rule as process communication etc. For examples on how GCLA can be used in a variety of domains see [GCLA 90, Aro 91, Aro 92a].

This freedom and generality has a price, however, and that is the efficiency of programs expressed in the language. In fact many intuitively correct GCLA programs will not even terminate, lacking a heuristic in the way search for proofs is conducted. Other programs may terminate with trivial, redundant or otherwise unwanted answers. Still others may get caught enumerating infinite sets of unwanted solutions and thus never produce the desired solutions.

From this trade-off between expressive power of the language, and complexity of the execution mechanism rises the issue of control. With a language like GCLA control issues becomes central. Some problems that arise in connection with control in GCLA are:

- How do we direct the search in an optimal manner?
- How do we exclude unwanted solutions from our answer substitutions?
- How do we manage exponential growth in our search spaces, and infinite branches in our search trees?

In GCLA I (see e.g.[GCLA 90]), we had ad hoc solutions to some of these problems. GCLA I used a fixed set of rules and a default search behavior that could only be modified at a global level. We could manage the global behavior of the interpreter by various parameters. However, this compromised the simplicity and elegance inherent in the theoretical foundations of GCLA programming. GCLA II is an attempt to construct a uniform framework for managing control issues in a definitional approach to programming.

1.3 Organization of the paper

The rest of the paper is organized as follows. Section 2 gives some background material, including a short description of the theory of partial inductive definitions, a finite version

of the theory with some novel material and a result on computing substitutions in the finite calculus. Section 2 is concluded with a description of GCLA I. Section 3 gives a detailed account of GCLA II.

2 Some background material

2.1 The Theory of Partial Inductive Definitions (PID)

A declarative semantics of GCLA is generally given in terms of the theory of partial inductive definitions (PID). We give a short summary of the theory here, but refer the interested reader to the more comprehensive presentation in [Hal 91]. Note that this theory is infinitary and have no concept of variables. In the context of various theories of partial inductive definitions this theory serves as a basis. – A theory in which most of the central intuitions are developed and to which most other theories of partial indicative definitions can be reduced.

Partial inductive definitions is an extension of ordinary or monotone inductive definitions (see e.g. [Acz 77]). The extension consists of allowing the definiens of a definition to depend on some (possibly defined) condition. This is expressed syntactically by the arrow construct i.e. $A \Leftarrow B \rightarrow C$ should be interpreted to mean that A is a member of the defined set or notion if C can be shown to be so under the assumption that B is. What follows is a description of the formal system of partial inductive definitions.

2.1.1 The language of PID

i) Universe

Let $\mathcal{U}$ be a universe (set) of atoms. Let $a, b, c, \ldots$ denote atoms.

ii) Conditions (over $\mathcal{U}$)

Define a set $Cond(\mathcal{U})$, of conditions (over $\mathcal{U}$):

- $\top$ (truth) is a condition (over $\mathcal{U}$)
- $\bot$ (falsity) is a condition (over $\mathcal{U}$)
- Each atom in $\mathcal{U}$ is a condition (over $\mathcal{U}$)
- If $\forall (i \in I)\, C_i \in Cond(\mathcal{U})$ then the vector $(C_i)_{i \in I} \in Cond(\mathcal{U})$
- If C_1 and C_2 are conditions (over $\mathcal{U}$), then the conditional $C_1 \rightarrow C_2$ is a condition (over $\mathcal{U}$).

Conditions are denoted by $A, B, C, A_1, B_1, C_1, A_2, B_2, C_2, \ldots$ Finite sets of conditions will be denoted by $\Gamma, \Delta, \Gamma_1, \Delta_1, \Gamma_2, \Delta_2, \ldots$

iii) Definitional Clauses (over $\mathcal{U}$)

If a is an atom (in $\mathcal{U}$) and C is a condition (over $\mathcal{U}$), then $a \Leftarrow C$ is a definitional clause (over $\mathcal{U}$). The atom a is called the head of the clause and the condition C its body.

iv) **(Partial Inductive) Definitions (over $\mathcal{U}$)**

A definition is a possibly empty set of definitional clauses, that defines atoms in terms of conditions.

$$\left\{\begin{array}{l} a \Leftarrow A \\ b \Leftarrow B \\ c \Leftarrow C \\ \quad \ldots \end{array}\right.$$

We denote definitions by $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{D}_2$, ... Note that an atom may be the head of several clauses in a definition.

2.1.2 Completion

Let the completion of a definition $\mathcal{D}$ be the set:

$$\hat{\mathcal{D}} = \mathcal{D} \cup \{ (a \Leftarrow \bot) \mid a \in \mathcal{U} \backslash Dom(\mathcal{D}) \}$$

where $Dom(\mathcal{D})$ is the domain of $\mathcal{D}$, i.e. the set of atoms occurring to the left of "$\Leftarrow$" in some definitional clause in $\mathcal{D}$.

2.1.3 Definiens

Let the definiens of an atom a with respect to a definition $\mathcal{D}$ be the following set of conditions:

$$\mathcal{D}(a) = \{ A \mid (a \Leftarrow A) \in \hat{\mathcal{D}} \}$$

Intuitively the definiens is the set of conditions that defines an atom in a given definition.

2.1.4 A notion of logical consequence

GCLA can properly be regarded as a logical programming language only if the definitions constituting its programs are given a logical reading. There is no primitive notion of logical consequence in the theory of partial inductive definitions but we can define a (logical) consequence relation that is local to a definition $\mathcal{D}$. It is local to $\mathcal{D}$ in the sense that it depends on $\mathcal{D}$. The logical interpretation of a definition is then given in terms of the derived notion of local $\mathcal{D}$-consequence: $\vdash_{\mathcal{D}}$.

i) Condition relations

First we formalize the concept of a condition relation, i.e. a relation between (finite) sets of conditions and conditions. A relation $\vdash$ is a condition relation if it has the following properties:

$$\Gamma \vdash \top \qquad (\top)$$

I.e. if truth is always derivable,

$$\Gamma, \bot \vdash C \qquad (\bot)$$

anything is derivable from falsity,

$$\frac{\Gamma \vdash C_i \text{ for all } (i \in I)}{\Gamma \vdash (C_i)_{i \in I}} \qquad (\vdash ())$$

vectors are derivable if *all* their elements are (c.f. conjunctions in sequent calculus),

$$\frac{\Gamma, A_i \vdash C}{\Gamma, (A_i)_{i \in I} \vdash C} \text{ for some } (i \in I) \qquad (() \vdash)$$

if something is derivable from a certain condition then it is also derivable from *any* vector containing that condition (c.f. conjunctions in sequent calculus),

$$\frac{\Gamma, A \vdash C}{\Gamma \vdash (A \to C)} \qquad (\vdash \to)$$

a conditional is derivable if its consequent is derivable from its antecedent and

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, (A \to B) \vdash C} \qquad (\to \vdash)$$

something is a derivable from a conditional if it is derivable from its consequent and its antecedent is derivable.

Each of these properties can also be seen as introduction or elimination rules for the logical constants (connectives), although the meaning these condition constructors also depend on the following two properties.

ii) **The $\mathcal{D}$-closure property**

A condition relation $\vdash$ is $\mathcal{D}$-closed if it satisfies the additional criteria:

$$\frac{\Gamma \vdash C}{\Gamma \vdash c} \text{ for some } C \in \mathcal{D}(c) \qquad (\vdash \mathcal{D})$$

I.e if *any* condition in the definiens of an atom is derivable then the atom is itself derivable and

$$\frac{\Gamma, A \vdash C \text{ for all } A \in \mathcal{D}(a)}{\Gamma, a \vdash C} \qquad (\mathcal{D} \vdash)$$

if something is derivable from *all* the conditions in the definiens of an atom it is also derivable from the atom itself.

These conditions can be seen as introduction and elimination rules for the *non logical* (*sic*!) constants.

We see here exactly how a consequence relation is defined in terms of the definition $\mathcal{D}$. The $(\vdash \mathcal{D})$ condition is analogue to the resolution rule of clause logics, while $(\mathcal{D} \vdash)$ is its dual. The $(\mathcal{D} \vdash)$ condition is the source of most of the unique features of GCLA. These conditions are the only ones that relate the calculus of PID to the definition, thus they can be seen as the parametric part of the calculus in the sense that through these rules the calculus is parametrized by the definition.

iii) **Reflexivity**

A condition relation $\vdash$ is reflexive if for all Γ and a, it satisfies:

$$\Gamma, a \vdash a \qquad (I)$$

I.e if any atom is derivable from itself.

iv) **Local $\mathcal{D}$-consequence $\vdash_{\mathcal{D}}$**

Local $\mathcal{D}$-consequence is our derived notion of logical consequence. Let $\vdash_{\mathcal{D}}$ be the smallest $\mathcal{D}$-closed reflexive condition relation and let the property $Def(\mathcal{D})$ be defined by:

$$Def(\mathcal{D}) = \{ a \in \mathcal{U} \mid (\vdash_{\mathcal{D}} a) \}$$

$Def(\mathcal{D})$ defines the atoms that are true in the local logic generated by $\mathcal{D}$.

Let

$$\overline{Def}(\mathcal{D}) = \{ a \in \mathcal{U} \mid (a \vdash_{\mathcal{D}} \bot) \}$$

$\overline{Def}(\mathcal{D})$ defines the atoms that are false in the local logic generated by $\mathcal{D}$. Atoms in $\mathcal{U} \setminus (Def(\mathcal{D}) \cup \overline{Def}(\mathcal{D}))$ have an undefined truth-value.

We generalize these notions to cover conditions as well as atoms. Let

$$Cov(\mathcal{D}) = \{C \in Cond(\mathcal{U}) \mid (\vdash_{\mathcal{D}} C)\}$$

and

$$\overline{Cov}(\mathcal{D}) = \{A \in Cond(\mathcal{U}) \mid (A \vdash_{\mathcal{D}} \bot)\}$$

then a condition C is true if $C \in \overline{Cov}(\mathcal{D})$, false if $C \in \overline{Cov}(\mathcal{D})$ and have an undefined truth-value otherwise.

Note that both atoms and conditions may also belong to the intersection of these sets. Such atoms are said to be locally inconsistent. It is an interesting property of the theory of PID that we may have $\vdash_{\mathcal{D}} (a, a \rightarrow \bot)$ for a particular atom a in a given definition $\mathcal{D}$ but still not $\vdash_{\mathcal{D}} \bot$. This is of course related to the fact that the calculus does not contain a cut rule. For a discussion of these and related issues see [S-H 91, S-H 92].

2.2 $OL\mathcal{D}$ - a finite, linear and ordered calculus of PID

As already mentioned the calculus of PID is infinitary in several respects. Both the set of clauses in a definition and the length of vectors may be infinite. This means that the rules of the calculus may have an infinite number of premises. In order to represent a class of infinite sets of clauses and a corresponding class of proofs we introduce variables in definitions and proofs. The approach taken here is based on the work of Lars Hallnäs and Peter Schroeder-Heister in [HSH 90]. We give an ordered version of their calculus $L\mathcal{D}$ (called $OL\mathcal{D}$) extended with two explicit (dual) pairing constructors "," and ";", a variable binding operator "Π" and a limited form of contraction. A similar linear calculus for GCLA I was presented in [GCLA 90].

The subset of PID's expressible in $OL\mathcal{D}$ is not the largest that we can finitely represent, but seems to give a reasonable compromise between expressiveness and computational complexity. For a discussion of stronger finite calculi of PID's see [Eri 91].

Familiarity with standard notation for application and composition of substitutions is assumed. Similarly for definitions of the notions of unifier and most general unifier and their use in Logic Programming refer to e.g. [Rob 65, Ll 87].

2.2.1 The language of $OL\mathcal{D}$

i) Signature Σ

Let Σ be a finite signature. Let $f^i, g^i, h^i, p^i, q^i, r^i, \ldots$ denote term constructors in Σ where i is the arity of a constructor and f, g, etc. is its name. Term constructors with an arity of 0 are called constants. The arity of term constructors will be left out whenever clear from the context.

ii) Variables $\mathcal{V}$

Let $\mathcal{V}$ be a denumerable set of variables. Let $x, y, z, \ldots$ denote variables.

iii) Terms $\mathcal{T}$

The set $\mathcal{T}$ of terms in $OL\mathcal{D}$ is defined inductively:

- Constants and variables are terms.
- if $t_1, \ldots, t_n$ are terms and f^n is a term constructor in Σ with arity n, then $f(t_1, \ldots, t_n)$ is a term.

Let $s, t, u, s_1, t_1, u_1, s_2, t_2, u_2, \ldots$ denote arbitrary terms. A non variable term is called an atom. Let $a, b, c, \ldots$ denote atoms. A ground term is a term that does not contain any variables. The set of all ground terms that can be generated from the term constructors in Σ is called the (Herbrand) universe $\mathcal{U}$ of the language. Define $\mathcal{V}(t)$ to be the set of variables that occur in the term t.

iv) Conditions

Define a set $Cond(\mathcal{T})$, of conditions over $\mathcal{T}$):

- Each term t in $\mathcal{T}$ is a condition.
- T (truth) is a condition.
- $\bot$ (falsity) is a condition.
- If A and B are conditions, then their product (A, B) is also a condition. The parenthesis may be omitted by regarding "," as a right associative operator.
- If A and B are conditions, then their sum $(A; B)$ is also a condition. The parenthesis may be omitted by regarding ";" as a right associative operator. The precedence of ";" is higher than that of ",".
- If A and B are conditions, then the conditional $(A \rightarrow B)$ is a condition. The parenthesis may be omitted by regarding "$\rightarrow$" as a left associative operator. The precedence of "$\rightarrow$" is higher than that of "," but lower than that of ";".
- If C is a condition and x is a variable in $\mathcal{V}$, then the universal $(\Pi x \cdot (C))$ is a condition. The parenthesis may be omitted by regarding "Π" an unary associative prefix operator and "$\cdot$" as a right associative operator with precedence higher than each of the operators defined above but lower than that of "Π".

A variable x is said to be bound by Π in the condition $\Pi x \cdot C$. C is said to be the scope of Πx in $\Pi x \cdot C$. A variable x is said to be free in a condition C if it occurs outside the scope of every Πx in C. Define $\mathcal{V}(C)$ to be the set of free variables in a condition C.

Lower precedence should be interpreted here to mean that an operator binds its arguments harder, e.g the expression

$$\Pi x \cdot \Pi x \cdot A, B, C \rightarrow D \rightarrow E; F; G$$

should be interpreted as

$$\Pi\,(x \cdot\ (\Pi\,(x \cdot\ (((((A,\ (B, C))\ \rightarrow D)\ \rightarrow E);\ (F; G)))))$$

Conditions are denoted by $A, B, C, A_1, B_1, C_1, \ldots$. Finite *ordered* sequences of conditions are denoted by $\Gamma, \Delta, \Gamma_1, \Delta_1, \Gamma_2, \Delta_2, \ldots$.

v) **Definitional Clauses**

If a is an atom (a non variable term in $\mathcal{T}$) and A is a condition (over $\mathcal{T}$), then $a \Leftarrow A$ is a (definitional) clause (over $\mathcal{T}$). Again, the atom a is the head of the clause and the condition C its body.

vi) **(Partial Inductive) Definitions (over $\mathcal{T}$)**

A definition $\mathcal{D}$ is a (possibly empty) finite sequence of definitional clauses.

$$\begin{cases} a_1 \Leftarrow A_1 \cdot \\ a_2 \Leftarrow A_2 \cdot \\ \quad \ldots \\ a_n \Leftarrow A_n \cdot \varnothing \end{cases}$$

Let $\mathcal{D}, \mathcal{D}_1, \mathcal{D}_2, \ldots$ denote definitions.

2.2.2 Definiens

As in the theory of partial inductive definitions the definiens operation is used to determine the defining conditions of an atom w.r.t. a definition. The difference lies in the presence of variables. In this ordered version we also wish to make significant the order in which the clauses are given in the program. We consider a clause as part of the definition of an atom if its head can be instantiated to become identical to the atom. If so the corresponding instance of the body of the clause should be made part of the definiens of the atom. Instead of generating a set, or simply an ordered sequence we generate a sum (as defined above) which will give us a very convenient formulation of the $(\mathcal{D} \vdash)_{\mathcal{L}}$ rule of $OL\mathcal{D}$. The following algorithm gives us a simple and natural ordering of the defining conditions in the sum.

Note that for the purpose of this construction we can regard the empty sum as identical to $\bot$. Thus $C;\bot$ is considered as identical to C. Similarly C is considered to be a singleton sum (and singleton product).

Define the (ordered) definiens operation $\mathcal{D}_{\mathcal{L}}$ over a definition $\mathcal{D}$ by

$$\mathcal{D}_{\mathcal{L}}(a) = def(a, \mathcal{D}) \text{ where}$$

$$def(a, \mathcal{D}) = \begin{cases} \bot \text{ if } \mathcal{D} \text{ is the empty definition} \\ def(a, \mathcal{D}') \\ \quad \text{if } \mathcal{D} = (a_i \Leftarrow C_i) \cdot \mathcal{D}' \text{ and } \neg\exists\rho\,(a_i\rho = a) \\ (C_i\xi;def(a, \mathcal{D}')) \\ \quad \text{if } \mathcal{D} = (a_i \Leftarrow C_i) \cdot \mathcal{D}' \text{ and } a_i\xi = a \end{cases}$$

where ρ and ξ are substitutions, and ξ in the last clause is a *most general* substitution such that $a_i\xi = a$. Note that $\mathcal{D}_{\mathcal{L}}(a)$ is $\perp$ if a is not an instance of a defined atom in $\mathcal{D}$.

As an example consider the following definition:

$$\begin{cases} p(a, y) \Leftarrow c(1, a, y) \cdot \\ p(x, b) \Leftarrow c(2, x, b) \cdot \\ p(z, z) \Leftarrow c(3, z, z) \cdot \varnothing \end{cases}$$

then

$$\mathcal{D}_{\mathcal{L}}(p(a, b)) = c(1, a, b); c(2, a, b),$$

$$\mathcal{D}_{\mathcal{L}}(p(a, a)) = c(1, a, a); c(3, a, a),$$

$$\mathcal{D}_{\mathcal{L}}(p(b, b)) = c(2, b, b); c(3, b, b)$$

and

$$\mathcal{D}_{\mathcal{L}}(p(z_0, z_0)) = c(3, z_0, z_0)$$

while

$$\mathcal{D}_{\mathcal{L}}(p(b, a)) = \perp$$

and

$$\mathcal{D}_{\mathcal{L}}(p(x_0, y_0)) = \perp.$$

2.2.3 *a*-sufficient substitutions

The definiens operation captures the case where we wish to determine if a particular atom a is an instance of some defined atom in the definition. An even more useful operation would be to determine if an atom a can be instantiated to some defined atom in a definition. However not all such instantiations are of use to us. We capture a property of substitutions that is fulfilled for all substitution that makes the $(\mathcal{D} \vdash)_{\mathcal{L}}$ rule sound (with respect to the underlying infinitary theory of PID) with the concept of an a-sufficient substitution.

A substitution σ is a-sufficient w.r.t. a definition $\mathcal{D}$ provided that

$$\forall\zeta\ (\mathcal{D}_{\mathcal{L}}(a\sigma\zeta) = (\mathcal{D}_{\mathcal{L}}(a\sigma))\ \zeta)$$

i.e. if for all substitutions ζ it is irrelevant whether we apply it to the instance $a\sigma$ and then perform the definiens operation or if we first apply the definiens operation and then apply ζ to the result.

There may be several a-sufficient substitutions w.r.t. a certain $\mathcal{D}$ for a given a.

A sequence of a-sufficient substitutions may be computed as follows:

Define

$$suff'(a, \mathcal{H}, C, \xi) = \begin{cases} \varnothing \text{ if } \mathcal{H} = \varnothing \wedge \exists \rho\, (a\rho \in C\rho) \\ (\xi \cdot \varnothing) \text{ if } \mathcal{H} = \varnothing \wedge \neg\exists \rho\, (a\rho \in C\rho) \\ suff'(a, \mathcal{H}', (b \cup C), \xi) \\ \qquad \text{if } \mathcal{H} = (b \cdot \mathcal{H}') \wedge \neg\exists \rho\, (a\rho = b\rho) \\ suff'(a\sigma, \mathcal{H}', C, \xi\sigma) \mathbin{@} suff'(a, \mathcal{H}', (b \cup C), \xi) \\ \qquad \text{if } \mathcal{H} = (b \cdot \mathcal{H}') \wedge \sigma = mgu(a, b) \end{cases}$$

where @ is sequence concatenation (append) operator. Let

$$suff(a, \mathcal{D}) = suff'(a, \mathcal{H}(\mathcal{D}), \varnothing, \{\})$$

where

$$\mathcal{H}(\mathcal{D}) = \begin{cases} \varnothing \text{ if } \mathcal{D} = \varnothing \\ a_i \cdot \mathcal{H}(\mathcal{D}') \text{ if } \mathcal{D} = (a_i \Leftarrow A_i) \cdot \mathcal{D}' \end{cases}$$

and $\{\}$ is the empty substitution.

Then each σ in *suff*$(a, \mathcal{D})$ is an a-sufficient substitution w.r.t. the definition $\mathcal{D}$, provided that the *no-extra-variable-condition*:

$$\forall\, ((a_i \Leftarrow A_i) \in \mathcal{D})\ (\exists \xi\, (a_i\xi = a\xi)) \rightarrow \mathcal{V}(A_i) \subseteq \mathcal{V}(a_i)$$

is fulfilled for a w.r.t. $\mathcal{D}$. This condition is exactly what is needed to ensure that the substitutions generated by the algorithm are a-sufficient.

This result is proved for a somewhat simpler algorithm in [HSH 90]. Each substitution in *suff*$(a, \mathcal{D})$ is computed in exactly the same way as in [HSH 90]. The difference lies in the substitutions excluded by the redundancy check of the first two clauses of the above definition.

The algorithm given in [HSH 90] also involves permuting the clauses of the definition $\mathcal{D}$ to compute a certain class of a-sufficient substitutions. We conjecture that this is not really necessary, and that the algorithm given here in fact computes the same class of substitutions as the one in [HSH 90].

The complexity of the algorithm published in [HSH 90] is exponential (because of the permutation) while the worst case complexity (the number of attempted unifications) for finding all solutions with this algorithm is $n^2 + \frac{n(n-1)}{2}$, where n is the number of clauses involved. What seems even more promising is the fact that most of this complexity can be moved to the compiler whereas before it was performed in runtime (see [Aro 92b]).

Note that this algorithm also gives a very natural ordering of the a-sufficient substitutions based on the order of the clauses in the definition $\mathcal{D}$.

Consider again the definition:

$$\mathcal{D} = \left\{ \begin{array}{l} p(a, y) \Leftarrow c(1, a, y) \cdot \\ p(x, b) \Leftarrow c(2, x, b) \cdot \\ p(z, z) \Leftarrow c(3, z, z) \cdot \varnothing \end{array} \right.$$

then

$$\begin{array}{c} suff(p(x_0, y_0), \mathcal{D}) = \\ \{\langle x_0, a\rangle, \langle y_0, b\rangle, \langle x, a\rangle, \langle y, b\rangle\} \cdot \\ \{\langle x_0, a\rangle, \langle y_0, a\rangle, \langle z, a\rangle\} \cdot \\ \{\langle x_0, b\rangle, \langle y_0, b\rangle, \langle z, b\rangle\} \cdot \\ \{\langle x_0, z\rangle, \langle y_0, z\rangle\} \cdot \varnothing \end{array}$$

The reader can verify that these substitutions are all $p(x_0, y_0)$-sufficient. Note that if we modified the first clause in the definition to be:

$$p(a, y) \Leftarrow c(1, a, z)$$

the definition would not fulfill the *no-extra-variable-condition* and only the last two substitutions produced by the algorithm would be $p(x_0, y_0)$-sufficient.

2.2.4 Sequents

A *sequent* (over $\mathcal{L}$) is a pair $(\Gamma \vdash C)$ of a finite sequence of conditions Γ and a condition C. We call the first element of the pair its *antecedent* and the second its *consequent*. The notation Γ_1, A, Γ_2 will be used to denote the sequence of conditions that results from concatenating the sequence Γ_1 with the sequence $A \cdot \Gamma_2$.

2.2.5 Goals

A goal of the calculus $OL\mathcal{D}$ is a pair. Its first element is a finite ordered sequence of *sequents*. The second element of the pair is a substitution that acts as a witness of the truth of all instances of the sequents in the infinitary calculus PID. In the rules given below the first element of a goal is represented as a sequence with "$\cdot$" as a sequence constructor. $\varnothing$ is the empty sequence.

2.2.6 Inference rules

Note that the inference rules are defined on goals rather than on sequents. As goals contain sequences of sequents the proofs of this calculus are linear in the sense that each rule has at most one premise. Note also that all rules apply to the first query in the goal and that this implies a depth first search for proofs in the calculus.

i) Initial context

$$\langle \varnothing, \quad \{\} \rangle \qquad (\varnothing)_{\mathcal{L}}$$

where $\{\}$ is the empty substitution.

The $(\varnothing)_{\mathcal{L}}$ is the only axiom of $OL\mathcal{D}$. A goal with an empty sequence of queries as its first element is solved, i.e. we have found a proof.

The name initial context refers to its role in a proof, i.e. as something that does not have to proved. In the context of proof search perhaps a term like final context would be more appropriate, as when the computation reaches this configuration the search procedure terminates.

ii) Truth

$$\frac{\langle \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash \top) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash \top)_{\mathcal{L}}$$

A sequent with $\top$ as its consequent is proved.

iii) Falsity

$$\frac{\langle \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, \bot, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (\bot \vdash)_{\mathcal{L}}$$

A sequent where $\bot$ occurs in the antecedent is also proved.

Note that the rule contains an implicit arbitrary permutation of the antecedent i.e. there is a choice of which condition in the antecedent to operate on. This comment applies to all rules operating on the antecedent.

iv) Initial sequent

$$\frac{\langle \Sigma\sigma, \quad \theta \rangle}{\langle (\Gamma_1, a, \Gamma_2 \vdash c) \cdot \Sigma, \quad \theta\sigma \rangle} \qquad (I)_{\mathcal{L}}$$

if $\sigma = mgu(a, c)$.

If a term a is in the antecedent of a sequent, c is the consequent of that sequent and if these two are unifiable, we have found a proof of that particular sequent. This corresponds to the reflexivity of the calculus of PID. We apply the most general unifying substitution σ to Σ and construct a witness for the conclusion by composing the *mgu* σ with the θ obtained from the premise. Sometimes we call this rule *axiom*, as it is similar to a common axiom rule in various logical calculi.

v) Arrow-right

$$\frac{\langle (A, \Gamma \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash (A \rightarrow C)) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash \rightarrow)_{\mathcal{L}}$$

The standard sequent calculus rule for implication to the right.

vi) Arrow-left

$$\frac{\langle (\Gamma_1, \Gamma_2 \vdash B) \cdot (\Gamma_1, A, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (B \rightarrow A), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (\rightarrow \vdash)_{1\mathcal{L}}$$

$$\frac{\langle (\Gamma_1, A, \Gamma_2 \vdash C) \cdot (\Gamma_1, \Gamma_2 \vdash B) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (B \rightarrow A), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (\rightarrow \vdash)_{2\mathcal{L}}$$

Two variants of an other standard sequent calculus rule.

vii) Product-right

$$\frac{\langle (\Gamma \vdash B) \cdot (\Gamma \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash (B, C)) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash ,)_{1\mathcal{L}}$$

$$\frac{\langle (\Gamma \vdash C) \cdot (\Gamma \vdash B) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash (B, C)) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash ,)_{2\mathcal{L}}$$

In $OL\mathcal{D}$ products are pairs but arbitrary finite vectors can be represented as nested product constructs. Products behave as conjunctions in sequent calculus.

viii) Product-left

$$\frac{\langle (\Gamma_1, A, B, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (, \vdash)_{\mathcal{L}}$$

This is the dual of the $(\vdash ;)_{\mathcal{L}}$ rule (defined below) with a built in contraction*. In the theory of partial inductive definitions we avoid the problem of contraction by regarding the ante-

---

* In their simplest form rules for product to the left (duals of the rules for sum to the right) would look like:

$$\frac{\langle (\Gamma_1, A, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \text{ and } \frac{\langle (\Gamma_1, B, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}$$

The rule as actually defined in the calculus could be thought of as composed of a contraction:

$$\frac{\langle (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A, B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}$$

followed by applications of each of the simple rules above.

cedents as sets. Of course we could introduce a general contraction rule but that would be computationally intractable. The rule as formulated here gives us back some of the strength lost by forbidding general contraction, but not all. E.g. the sequent

$$\Pi x.p(x) \vdash (p(a), p(b))$$

is not provable in this calculus, but would be in a calculus with contraction. For more material on the role of contraction an other structural rules in this context see [HSH 90].

ix) Sum-right

$$\frac{\langle (\Gamma \vdash B) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash (B;C)) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash ;)_{1\mathcal{L}}$$

$$\frac{\langle (\Gamma \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma \vdash (B;C)) \cdot \Sigma, \quad \theta \rangle} \qquad (\vdash ;)_{2\mathcal{L}}$$

The sum constructor ";" here occurring to the right behaves as disjunction in sequent calculus. This constructor does not occur in the calculus of PID, and is not strictly necessary. It is included because it gives a convenient formulation of the $(\mathcal{D} \vdash)_{\mathcal{L}}$ rule below. In addition it is quite useful in practical programming, and actually part of most present GCLA implementations. Note that the rules would be completely dual to the $(, \vdash)_{\mathcal{L}}$ if that did not contain the implicit contraction.

x) Sum-left

$$\frac{\langle (\Gamma_1, A, \Gamma_2 \vdash C) \cdot (\Gamma_1, B, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A;B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (; \vdash)_{1\mathcal{L}}$$

$$\frac{\langle (\Gamma_1, B, \Gamma_2 \vdash C) \cdot (\Gamma_1, A, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (A;B), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (; \vdash)_{2\mathcal{L}}$$

The sum constructor ";" occurring in the antecedent is again analogue to disjunction. The rules are dual to the $(\vdash ,)_{i\mathcal{L}}$ rules

xi) Π*-left*

$$\frac{\langle (\Gamma_1, A\sigma, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle}{\langle (\Gamma_1, (\Pi x \cdot A), \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta \rangle} \qquad (\Pi \vdash)_{\mathcal{L}}$$

where σ substitutes a new unique variable for all free occurrences of x in C. Note that we do not state a rule for Π to the right. Doing that in a correct way would strengthen the calculus considerably but would also introduce complexities that we currently feel do not belong in a programming language. See [Eri 91] for details of such a calculus.

xii) Definition-right

$$\frac{\langle (\Gamma\sigma \vdash C\sigma\xi) \cdot \Sigma\sigma, \quad \theta\rangle}{\langle (\Gamma \vdash c) \cdot \Sigma, \quad \theta\sigma\rangle} \qquad (\vdash \mathcal{D})_{\mathcal{L}}$$

if $(b \Leftarrow C) \in \mathcal{D}$, $\sigma = mgu(b, c)$ and ξ is a substitution that assign unique new variables to the free variables in C that are not also free in b.

The$(\vdash \mathcal{D})_{\mathcal{L}}$ rule corresponds to the resolution rule in horn logics. Note that we have one instance of this rule for each clause in the definitions whose head is unifiable with c.

C.f. the definiens operation where *each* such clause contributes an element of the sum $\mathcal{D}_{\mathcal{L}}(c\xi)$ for some c-sufficient substitution ξ.

xiii) Definition-left

$$\frac{\langle (\Gamma_1\sigma, \mathcal{D}_{\mathcal{L}}(a\sigma), \Gamma_2\sigma \vdash C\sigma) \cdot \Sigma\sigma, \quad \theta\rangle}{\langle (\Gamma_1, a, \Gamma_2 \vdash C) \cdot \Sigma, \quad \theta\sigma\rangle} \qquad (\mathcal{D} \vdash)_{\mathcal{L}}$$

if σ is a a-sufficient substitution w.r.t. $\mathcal{D}$.

Note that we have one instance of this rule for every a-sufficient substitution σ in $suff(a, \mathcal{D})$. Using a-sufficient substitutions in this rule is necessary for the soundness of this rule with respect to the theory of PID.

2.2.7 Semantics and soundness

A semantics of this calculus can be given in terms of the infinitary calculus of PID. The details of such an interpretation and a soundness proof is carried out for a similar but somewhat stronger calculus in [Eri 91]. We conjecture that the calculus $OL\mathcal{D}$ is also sound with respect to the theory of partial inductive definitions.

2.3 GCLA

GCLA is a logic programming language that is based on the theory of partial inductive definitions. As stated briefly in section 1.1 GCLA constitutes a generalization of languages based on pure horn clause logic.

The role of a program in traditional logic programming languages (e.g. Prolog) is in GCLA played by the definition. A set of clauses is regarded as a definition of a logic in which inferences are made as responses to *queries* (in the form of *sequents*) posed to the system.

GCLA as it existed up to 1989 (see e.g. [GCLA 90]) is now called GCLA I to distinguish it from GCLA II, the language as it exists today.

2.3.1 GCLA I

An operational semantics of GCLA I is obtained by giving a set of inference rules, e.g. some variant of the linear calculus $OL\mathcal{D}$ defined above and some additional search heuristics. One such semantics is defined in [GCLA 90] and we will not repeat it here. Note how-

ever, that $OL\mathcal{D}$ does not determine a unique search procedure. In many cases we have a choice between the axiom rule, a rule operating on one of the conditions in the antecedent, and one operating on the consequent. Both the initial sequent and all rules operating on the antecedent makes an indeterministic choice of a condition in the antecedent. GCLA II provides a formalism in which these choices are made explicit

The search behavior of GCLA I could be modified globally by modifying certain parameters of the system. This was a rather unsatisfactory solution, and its weaknesses were one of the main motivations behind the development of GCLA II.

A simplified way of thinking about GCLA II is as a system that allows formal definition of alternative finite ordered calculi and associated search heuristics. In fact, the default set of rule definitions in a GCLA II system is generally very close to $OL\mathcal{D}$

3 The issue of control - GCLA II

This section contains a description and an analysis of GCLA II, a version of GCLA that gives the user explicit control over various choices made in the execution of GCLA programs. Section 3.1 describes the general ideas used in GCLA II while section 3.2 gives an informal example of how it can be used. In section 3.3 the different components of a GCLA II program are described in detail and section 3.4 gives an operational semantics of the language.

3.1 Concepts & Intuitions

Control issues in GCLA arise mainly in four types of situations:

- In the choice of a clause from the definition in ($\vdash \mathcal{D}$)
- In the choice of an a-sufficient substitution in ($\mathcal{D} \vdash$)
- In the choice of inference rule used to prove a certain sequent
- In the choice of a condition in the antecedent for the rule to operate on

The first of these choices is similar to the situation in Prolog, and GCLA uses essentially the same approach as does Prolog i.e. utilize the order in which the clauses occur in the definition. The second issue GCLA also handles by using the order of the clauses in the program. The algorithm used to compute a-sufficient substitutions given in section 2.2.3 gives the substitutions in a natural order based on the order of the clauses in the definition. The third and forth of the above issues are the domain of the control language of GCLA II proper, and will be the main subject of this section.

The last three control issues sets GCLA apart from resolution based systems that use only one rule of inference, e.g. Prolog. The Gödel system ([HL 91]) addresses some of the same problems in the context of SLDNF resolution with a flexible computation rule based on a generalized delay mechanism, and a commit operator. This approach does not handle the problems involved with choosing between several inference rules, nor the choice of condition from the antecedent or consequent, nor, of course the choice of a substitution as that is always deterministic in SLDNF. The strict separation of object and meta level variables used in GCLA II is similar in concept to the ground representation of object level variables enforced by Gödel.

The situation in GCLA is similar to the one in automatic theorem proving, and it turns out that we encounter many of the same problems that LCF is designed to solve (see e.g. [GMW 79, Pau 83]). However our approach is quite different. The theoretical background is completely different, and our goal has been a programming language, rather than a system for theorem proving.

Of course GCLA could also be used as a basis for a system for theorem proving, but as such it has three major drawbacks: 1) it has a limited concept of variable quantification, 2) it is essentially a first order system and 3) it has no general contraction.

Stronger finite representations (with more powerful quantification) *do* exist and these allow us to do proof search in larger subsets of valid PID proofs (see [Eri 91]). These formalisms have been used as a basis for a semiautomatic proof editor [Eri 90].

In GCLA the order in which rules are tried may affect both termination and the order in which answers are presented. In addition certain programming methodologies (notably

functional programming) require repeated application of certain sequences of rules to become efficient.

3.1.1 Rules

An inference rule can be regarded as a partial function from a set of derivations to the conclusion sequent. Note that we do *not* regard the inference rule as a function from its premises to its conclusion but from the *proofs* of the premises to the conclusion, i.e. a sequent.

Such functions can (obviously) be inductively defined. In fact, using this idea the inference rules of GCLA can be expressed in a subset of GCLA. Moreover these functions can be computed in a suitable sub-calculus of $OL\mathcal{D}$.

With each rule function we associate a particular set of conditions for the rule to be applicable, sometimes called *provisos* or side conditions. If the provisos are not fulfilled, we may regard the function as undefined for that particular set of arguments (proofs).

As proof search is conducted "backwards", from conclusion to premises, it is actually the inverses of these rule functions that are interesting in the context of proof search. Proof search then, can be regarded as the process of constructing a functional expression consisting of repeated applications of rule functions, where each function symbol in the expression is the name of a rule and the arguments are also expressions of this same kind.

Such an expression is a representation of a (set of) proofs. We will call such a functional expression a proof term. If the rule functions occurring in a proof term are deterministic and the proof term is ground it represents a derivation in the calculus defined by the rule definitions. Otherwise the proof term represents a set or class of derivations.

Now, how can proof terms be utilized to exercise control over the possible inferences in a proof search? If we allow provisos to relate the structure of a partially instantiated proof term with the parts of the resulting sequent, we achieve explicit control over the proof search. But how is this possibility to be utilized in an efficient manner?

In general, we do not want to write a new set of rules for each new problem. This would amount to writing a specialized interpreter for each particular problem. In most cases we will instead like to associate classes of sequents with classes of proofs.

This can be accomplished in a variety of ways. One, used in our first approach to solve these problems (see [GCLA 91]) is to let certain provisos instantiate variables occurring in the proof terms according to the structure of the corresponding conclusion sequent. These provisos were provided by the user and were specific to each problem domain. In this way the inference rules were parametrized by these provisos.

This was a quite adequate solution, but the resulting formalism was rather awkward and verbose. Furthermore we lacked a concept of modularity. In this paper (and in current implementations) we have choosen a different approach.

The approach we now pursue instead focuses on constraining the search behavior of the system using strategy rules similar in concept to the inference rules mentioned above.

Of course if we allow the user to define arbitrary inference rules we cannot guarantee soundness of the system. Instead we suggest a partitioning of the rules into:

- *Inference rules* that actually map proofs of premises into conclusions in a nontrivial manner and

- *Strategy rules* that constrain the search behavior in a well defined way but never actually manipulate the structure of the sequents. Rules of this second kind will not affect the soundness of the system and can thus be used even by a naive user.

With this approach we can provide a sound basic set of inference rules and standard strategies implementing common search behaviors. The user can then define his or her own strategies suitable to his problem domain in terms of the provided rules and strategies. If used in this manner the system is always sound. If the user so chooses he can still define specialized inference rules but he would then, himself, have to guarantee the soundness of these rules.

As already mentioned GCLA can be used as a functional programming language. The methodology used in the meta-language of GCLA II is a generalization of the one developed to write functional programs (see e.g. [Aro 91]). Note that we can compute both the value of a functional expression and its inverse as a (possibly indeterministic) function.

If we have a function definition:

```
plus(0,Y) <= Y.
plus(s(X),Y) <= succ(plus(X,Y)).

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

we can compute the value of the functional expression "`plus(s(0),s(0))`" (as a substitution for the variable `Z`) by querying GCLA with the following sequent

```
plus(s(0),s(0) \- Z
```

but we can also compute its inverse by instead starting out with the sequent

```
plus(X,Y) \- s(s(0))
```

and expecting instantiations of the variables `X` and `Y` as a result i.e. a value of the inverse function. Of course this may not be the only answer if the function is not injective (as in this case), so we need to be able to handle indeterministic functions as well. This is accomplished in a very natural way by backtracking in GCLA. Given the above program and the second sequent GCLA will give the following answers:

```
x = 0
y = s(s(0));

x = s(0)
y = s(0);

x = s(s(0))
y = 0;

no.
```

where "`;`" is a command to the interpreter to search for more answers and "`no`" means no more answers could be found.

Now, given a proof term

$$RuleName(Pt_1, \ldots, Pt_n)$$

and a sequent: