ISRN SICS-R--93/02--SE

**Implementational Issues in GCLA:
A-Sufficiency and the Definiens Operation.**

by

Martin Aronsson

Implementational Issues in GCLA: A-Sufficiency and the Definiens Operation

by

Martin Aronsson

Knowledge Based Systems Laboratory; 1993-03-04

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

Abstract

We present algorithms for computing *A*-sufficient substitutions and constraint sets together with the definiens operation. These operations are primitive operations in the language GCLA. The paper first defines those primitives, which together form a dual rule to SLD resolution, and then describes the different algorithms and some of their properties together with examples. One of the algorithms shows how a definition can be compiled into a representation holding all possible *A*-sufficient substitutions/constraint sets together with their corresponding definiens. This representation makes the computation at runtime of a definiens and an *A*-sufficient substitution/constraint set have the same complexity as the table lookup operation `clause/2` in Prolog. The paper also describes the generalisation from unification (sets of equalities) to constraint sets and satisfiability of systems of equalities and inequalities.

Keywords: GCLA, inequality, constraints, negation

This is an extended version of a paper first published in Springer Verlags Lecture Notes in Artificial Intelligence 1993: Extensions of Logic Programming, proceedings of a workshop held in Bologna, Italy, 1992.

1. Introduction

GCLA is a programming system developed at SICS for some years [Aro90, Aro92a, Kre92, HS-H90, HS-H91]. It is best regarded as a logic programming language, although it does not have the same theoretical foundation as most other logic programming languages. While traditional logic programming languages are based on logic, GCLA is based on a more general concept of inductive definitions, called *Partial Inductive Definitions* [Hal91].

One of the basic ideas in GCLA is that the program, called the *definition*, forms a partial inductive definition, and to that definition a consequence relation is associated, denoted by $\vdash_{\mathcal{D}}$, where $\mathcal{D}$ denotes a particular definition. A GCLA goal consists of a *sequent*, $\Gamma \vdash_{\mathcal{D}} C$, where Γ is called the *antecedent* and C the *consequent*. Assuming something is the same as adding that element to the antecedent. It is here that GCLA differs from most other logic programming languages, since that is not the same as adding something to the definition.

An atom in the antecedent can be replaced by its defining conditions, which are called its *definiens*. The corresponding inference rule is called the definition-left inference rule ($\mathcal{D}\vdash$), and it is this operation that gives GCLA most of its power compared to ordinary Horn clause logic programming systems (e.g. Prolog). The operation described in [HS-H91] is complex and laborious to perform, and it is divided into two suboperations; the *definiens operation* and the calculation of *A-sufficient* substitutions. It is in the latter one much of the execution time is spent, and therefore it is of great interest to develop better algorithms for this operation.

Section 2 gives the background and the necessary definitions. Section 3 of the paper presents three algorithms; the 'original' one presented in [HS-H91], and two others of which one compiles the definition into a new representation that is used to compute a definiens and an A-sufficient substitution. We also present some test values and comparisons of various data, such as execution time, number of mgu's calculated, number of mgu's needed etc.

By replacing unification of terms with systems of (syntactic) equalities and inequalities, an interesting and more expressive language is defined. Unification guards in the head of the clauses are introduced, and the algorithms are generalised to handle the new satisfiability conditions. The constraint solving system can be further generalised, although the paper does not present any material on this. Section 4 presents the changes and additional terminology relative to section 2 as well as the generalised algorithms.

Section 5 contains a brief discussion of related work, and section 6 contains a brief discussion and some references to future work.

The GCLA system is implemented on top of Sicstus Prolog, and thus the presentation is influenced by some Prolog operations, notably the use of *cannot prove*, '\+', at some places, and we assume that the reader is familiar with Prolog's general operational behaviour.

2. Background and Definitions

Although in GCLAII [Kre92, Aro92a] the programmer is free to write any inference rule he wants, we will stick to the original rules presented in [HS-H90, HS-H91]. The definiens operation and the computation of A-sufficient substitutions are primitive operations of GCLAII, which the user can utilize in his own inference rules, and thus these primitives must have an efficient implementation.

2.1 Background

A GCLA goal is a sequent of the form $\Gamma \vdash_{\mathcal{D}} C$, where the consequence relation $\vdash_{\mathcal{D}}$ is defined by a particular definition $\mathcal{D}$, together with some rules for handling non-atomic conditions (we will often omit the $\mathcal{D}$ -subscript in $\vdash_{\mathcal{D}}$ when it is clear what the definition is). Rules handling non-atomic conditions are those rules that do not use the definition, for example arrow right, which adds an element to the antecedent. Those rules do not use the definition, and therefore we will not further discuss the implementation of the structural rules in this paper (see [Aro92b]), but look at algorithms and representational ideas for definitions, in particular together with the *definiens* operation and *A*-sufficient substitutions.

Before going into detail, we define the syntax of a GCLA definition. Since we talk about inductive definitions, we have no predicates or functions, just terms and conditions.

A *constant* is a *term*, and so is a *variable*. Constants begin with a lowercase letter, while variables begin with an uppercase letter, or `'_'`. The single symbol `'_'` denotes an anonymous variable. If $A_1, \dots, A_n$ are terms and f is a functor (term constructor) of arity n , then $f(A_1, \dots, A_n)$ is a *term*. All terms are *conditions*, and if C_1 and C_2 are conditions, then so are $(C_1 \rightarrow C_2)$, (C_1, C_2) , $(C_1; C_2)$. `true` and `false` are conditions, and if x is a variable and C a condition, then $(\text{pi } x \backslash C)$ is a *condition*, where x occurs bound in the condition C .

An *atom* is a term which is not a variable. If A is an atom, C a condition, then $A \leq C$ is a *clause*. We will refer to A as the *head* of the clause and C as the *body* of the clause. Often we will use H to denote a head and B to denote a body of a clause.

An ordered set of clauses forms a *definition* $\mathcal{D}$.

It is obvious that the universe of constants and atoms is infinite, as well as the number of variables, but that each term is finite.

An example definition is

```
p(X, 1) <= q(X) .
p(X, Y) <= r(X, Y) .

q(2) .
```

There are two GCLA inference rules that use the definition: $\vdash_{\mathcal{D}}$ and $\mathcal{D}\vdash$. The inference rule $\vdash_{\mathcal{D}}$ has the definition

$$\frac{\Gamma\sigma \vdash B\sigma}{\Gamma \vdash A} \vdash_{\mathcal{D}}$$

if $(H \leq B) \in \mathcal{D}$ and $\sigma = \text{mgu}(H, A)$.

This rule corresponds to SLD resolution. Thus we can use the same representation technique as in Prolog's clause, which gives good performance.

The $\mathcal{D}\vdash$ rule is the dual to the $\vdash_{\mathcal{D}}$ rule. While the $\vdash_{\mathcal{D}}$ rule operates on one clause at a time, the $\mathcal{D}\vdash$ rule considers all clauses in a definition. The operation that collects all bodies of the clauses considered is called the *definiens operation* and is denoted by $\mathcal{D}(A)$, where A is an atom, and is explained in section 2.2.2. The rule $\mathcal{D}\vdash$ is defined as

$$\frac{\{\Gamma_1\sigma, B, \Gamma_2\sigma \vdash C\sigma \mid B \in \mathcal{D}(A\sigma)\}}{(\Gamma_1, A, \Gamma_2 \vdash C)} \mathcal{D}\vdash$$

if σ is an A -sufficient substitution (explained in section 2.2.1) with respect to $\mathcal{D}$.

Note that there is one instance of this rule for every σ , and if there are no members in $\mathcal{D}(A\sigma)$ there are no premises in the $\mathcal{D}\vdash$ rule, which means that it holds unconditionally. In this way, the negation of a condition C , ' $\text{not}(C)$ ', is accomplished by posing the query $C \vdash_{\text{false}}$, and if C is not defined in $\mathcal{D}$ the query holds unconditionally.

This rule is much more interesting from an implementational point of view, since it is a "new" rule, i.e. it has little or no correspondence with any other operation in any other language. We will concentrate on the new operations, i.e. on the definiens operation and the generation of A -sufficient substitutions.

2.2 Definitions

We will now give the definitions of the definiens operation and A -sufficient substitutions.

2.2.1 Notations

We will denote substitutions by the letters θ, σ and τ , possibly with subscripts. The empty substitution will be denoted by ε . A set of items will be denoted by $\{\dots\}$, a conjunctive vector will be denoted by $(e_1, \dots, e_n)$, and a disjunctive sum will be denoted by $(e_1; \dots; e_n)$. The concatenation of an element e to a vector L is denoted by (e, L) , and the concatenation of an element e to a sum L is denoted by $(e; L)$. Application of a substitution σ to a term T (sequent, set etc) will be written as $T\sigma$.

2.2.2 The Definiens Operation

The definiens operation is defined formally as: The definiens $\mathcal{D}(A)$ of an atom A , is the set of all instances $B\tau$ of all bodies B such that the instances of their corresponding heads $H\tau$ are equal to A , i.e.

$$\mathcal{D}(A) = \{B\tau \mid A = H\tau \text{ and } H \leq B \in \mathcal{D}\}$$

2.2.3 A -Sufficient Substitutions

Given an atom A , a substitution σ is called *A -sufficient* if for $A\sigma$ the condition $\mathcal{D}(A\sigma\tau) = (\mathcal{D}(A\sigma))\tau$ is fulfilled, for all τ . The reason for this condition is to ensure that $\mathcal{D}\vdash$ is closed under substitution, i.e. that derivability of $\Gamma \vdash C$ implies that $\Gamma\sigma \vdash C\sigma$, for any substitution σ .

The definiens operation and the calculation of an A -sufficient substitution are closely connected, as we shall see later. It is the case that given a definition and an A -sufficient substitution, the definiens is completely determined. The computation of these two can be combined into one operation, which we will call *$\text{succ}_{\mathcal{D}}$* .

2.2.4 Variable-Check

A definition clause $H \leq B$ is said to satisfy the *no-extra-variable condition* if every free variable occurring in B also occurs in H . Suppose now that T is fixed. Then we say that σ passes the *variable-check* if all definition clauses $H \leq B$ for which $H\tau = T\sigma$ holds for some τ , satisfy the no-extra-variable condition. Intuitively, the variable-check for σ means that every clause from which $T\sigma$ can be obtained by substitution, satisfies the no-extra-variable condition.

The variable check must be carried out to ensure that the algorithms presented here produce A -sufficient substitutions. Once explicit quantifiers are introduced into the language, as in [Eri92], the variable check could be replaced by explicit quantifiers. In the rest of this paper we will assume that the σ considered passes the variable check.

3. Algorithms without Constraints

We will present three algorithms. The first one is the original one described in [HS-H91]. Algorithm 2 is a refined version of the first, while the third one generates a new representation of the definition $\mathcal{D}$ to hold the possible defining conditions and A -sufficient substitutions. One can regard the third algorithm as a precomputation, or partial evaluation, of the possible definiens for every possible term.

We will represent an empty definiens by the symbol `false`, and when $\mathcal{D}(A)$ contains more than one element, it will be written as a sum $(B_1 ; \dots ; B_n)$, which is close to how it is represented in the GCLA system. We will identify a sum $(B_1 ; \dots ; B_n ; \text{false})$ with $(B_1 ; \dots ; B_n)$. For further details on how sums and other constructs are handled the reader should consult [Kre92].

3.1 Algorithm 1

The first algorithm was presented in [HS-H91], and could be very inefficient. The complexity is $O(n!)$, where n is the number of clauses considered, and $!$ is the factorial operation. The number of answers produced by this algorithm (before making the answer a set, where all redundancies are removed) is in the worst case equal to the factorial of the number of the clauses considered.

3.1.1 Definition

Let the clauses be ordered in some way. The heads of the definition $\mathcal{D}$ that should be considered are those that are unifiable with the atom A considered:

$$\{H \mid (H \leq B) \in \mathcal{D} \text{ and } mgu(A, H) \text{ exists}\}$$

Call this set $\mathcal{L}$. Let k denote the cardinality of $\mathcal{L}$. Now consider all the permutations $H_1, \dots, H_k$ of $\mathcal{L}$ and define the following algorithm for generating an A -sufficient substitution:

Let

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

Then define

$$\begin{aligned} \sigma_0 &= \varepsilon \\ \sigma_{m+1} &= \sigma_m mgu'(A\sigma_m, H_{m+1}) \\ \sigma &= \sigma_k \end{aligned}$$

The factorial complexity comes from the permutation of the set $\mathcal{L}$, where all permutations are considered. The algorithm depends on the permutation but is deterministic once the permutation is fixed.

It is easy to see how the algorithm can be extended to also compute the definiens operation. Just associate with every head H_i where there exists an mgu the corresponding body of H_i , and collect them in a sum in parallel with σ_i , which we call $\mathcal{B}$. Thus, define the operation $suff_{\mathcal{D}}$ to return both an A -sufficient substitution σ and the sum $\mathcal{B}$:

$$suff_{\mathcal{D}}(A) = \langle \sigma, \mathcal{B} \rangle$$

and the corresponding algorithm is

$$\begin{aligned}
\langle \sigma_0, \mathcal{B}_0 \rangle &= \langle \varepsilon, \text{false} \rangle \\
\langle \sigma_{m+1}, \mathcal{B}_{m+1} \rangle &= \begin{cases} \langle \sigma_m \tau, (\mathcal{B}_{m+1} \tau; \mathcal{B}) \rangle & \text{if } \tau = \text{mgu}'(A\sigma_m, H_{m+1}) \\ \langle \sigma_m, \mathcal{B} \rangle & \text{otherwise} \end{cases} \\
\langle \sigma, \mathcal{B} \rangle &= \langle \sigma_k, \text{reverse}(\mathcal{B}_k) \rangle
\end{aligned}$$

where $\mathcal{B}_{m+1}$ is the body of clause $m+1$, and *reverse* reverses its argument. The reverse operation is performed in order to get the bodies in the order of the permutation considered.

Note that there could be several sums returned by the algorithm corresponding to one substitution, but if we regard the sums as sets, they will be equal (see example 3.1.2 below). Also note that if the sum is not empty ($\mathcal{B} \neq \text{false}$), we identify the sum $(B_1 ; \dots ; B_n ; \text{false})$ with $(B_1 ; \dots ; B_n)$.

3.1.2 Example

1) Consider the definition

$$\begin{aligned}
p(1) &\leq b_1. \\
p(X) &\leq b_2(X).
\end{aligned}$$

where b_1 and b_2 stand for arbitrary bodies. To the term $p(Z)$ the algorithm returns two equal substitutions, $\{Z/1\}$; $\text{suff}_{\mathcal{D}}(p(Z)) = \langle \{Z/1\}, (b_1 ; b_2(1)) \rangle$ and $\text{suff}_{\mathcal{D}}(p(Z)) = \langle \{Z/1\}, (b_2(1) ; b_1) \rangle$. Note that the second argument of the pair is permuted in the two solutions, but that the $p(Z)$ -sufficient substitution is the same. These two solutions stem from the fact that all permutations of the heads are considered. Those two solutions would be regarded as the same, if we regard the second argument of the pair as a set.

3.1.3 Soundness

That the algorithm above produces an A -sufficient substitution σ is proved in [HS-H91], provided that the variable-check for σ is satisfied. That it is not complete is shown by the following example: Consider the program

$$p(1) \leq b1.$$

and let $A = p(X)$. Then $\sigma = \{X/3\}$ is an A -sufficient substitution, since $\mathcal{D}(p(3)) = \emptyset$ and $\mathcal{D}(p(X) \sigma \tau) = \mathcal{D}(p(X) \sigma) \tau$, for all τ , but $\{X/3\}$ cannot be generated by the algorithm.

3.2 Algorithm 2

Algorithm 2 is a refinement of algorithm 1. The refinement lies in how the generation of the permutations is performed. Instead of considering all permutations of the heads of the set $\mathcal{L}$, just some are considered. If the clauses are numbered from top to bottom, it is easy to see that if a clause's head at position i is unifiable with a clause's head at position j , it does not matter if we first take i and then j , or if we first take j and then i . This means that an ordering condition can be imposed on the permutation algorithm: do not consider permutations which result in an ordered set of heads (bodies) whose clause numbers are not in ascending order. This makes this algorithm have a complexity of $O(2^{n+1})$.

A version of this algorithm is also described in [Kre92], but without step 3) below.

3.2.1 Definition

The heads that should be considered are those that are unifiable with the atom A considered:

$$\{H \mid (H \leq B) \in \mathcal{D} \text{ and } mgu(H, A) \text{ exists}\}$$

Call this set $\mathcal{L}$. Let k denote the cardinality of $\mathcal{L}$, and consider some ordering $H_1, \dots, H_k$ of $\mathcal{L}$. Define a set $\mathcal{S}$, which collects the heads that should not contribute to the A -sufficient substitution. Define a set $\mathcal{B}$ which collects those bodies that are part of the definiens. Define the following algorithm for calculating $suff_{\mathcal{D}}$:

- 0) $\langle \sigma_0, \mathcal{S}_0, \mathcal{B}_0 \rangle = \langle \epsilon, \emptyset, \text{false} \rangle$
- 1) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \tau, \mathcal{S}_m, (B_{m+1}; \mathcal{B}_m) \rangle$ if $\tau = mgu(A\sigma_m, H_{m+1})$
- 2) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m, \{H_{m+1}, \mathcal{S}_m\}, \mathcal{B}_m \rangle$ if $\tau = mgu(A\sigma_m, H_{m+1})$
- 3) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m, \mathcal{S}_m, \mathcal{B}_m \rangle$ if $\neg \exists \tau (\tau = mgu(A\sigma_m, H_{m+1}))$
- 4) $\langle \sigma, \mathcal{B} \rangle = \langle \sigma_k, \mathcal{B}_k \rangle$ provided that $\forall H \in \mathcal{S}_k: (\neg \exists \tau (\tau = mgu(A\sigma_k, H)))$

Instead of considering all permutations of $\mathcal{L}$ as in the first algorithm, this algorithm is centred around the fact that either a head H_i contributes to the A -sufficient substitution, or not. This is reflected in the three clauses 1), 2) and 3), together with the provided-check in 4). Instead of always adding H_i to $\mathcal{S}$ when there does not exist a unifier of H_i and A , clause 3) is introduced to keep the set $\mathcal{S}$ as small as possible. There is a nondeterministic choice between 1) and 2), whenever there exists an mgu of A and H_i . If the mgu τ is added to σ_m , then the head H_i is not collected in the set $\mathcal{S}$. If τ is not added to σ_m , then H_i is added to $\mathcal{S}$, and when all k elements of $\mathcal{L}$ have been considered, the heads in $\mathcal{S}$ are checked to see that they are not unifiable with $A\sigma_k$. This last check checks that the elements cannot contribute to σ , which was assumed in clause 2) at some step $j < k$.

By always trying clause 1) first, we get a natural order of the solutions. To get all possible pairs of A -sufficient substitutions and defining conditions $\mathcal{B}$, a complete search is performed.

3.2.3 Examples

1) Consider again the first example of the previous section,

$$\begin{aligned} p(1) &\leq b_1. \\ p(X) &\leq b_2(X). \end{aligned}$$

We get one answer substitution $\{z/1\}$ to the term $p(z)$, since the only permutation that will be considered is the permutation: first $\sigma_1 = mgu(p(z), p(1))$, then $\sigma_2 = mgu(p(z)\sigma_1, p(X))$. The permutation $\tau_1 = mgu(p(z), p(X))$, $\tau_2 = mgu(p(z)\tau_1, p(1))$ will not be considered. Thus $suff_{\mathcal{D}}(p(z)) = \langle \{z/1\}, (b_1; b_2(1)) \rangle$, to be compared with the result in algorithm 1.

2) Consider the definition

$$\begin{aligned} p(1) &\leq b_1. \\ p(2) &\leq b_2. \end{aligned}$$

We get two answer substitutions $\{z/1\}$ and $\{z/2\}$ to the term $p(z)$, since the substitution depends on which of the clauses 1) or 2) in the algorithm that is chosen first. $suff_{\mathcal{D}}(p(z)) = \langle \{z/1\}, b_1 \rangle$ or $suff_{\mathcal{D}}(p(z)) = \langle \{z/2\}, b_2 \rangle$, and the definiens $\mathcal{D}(p(z))$

equals $\{b_1\}$ or $\{b_2\}$. A trace of the working algorithm is presented below for the term $A = p(x)$.

Answer 1:

0) $\langle \epsilon, \emptyset, \text{false} \rangle$	Initial state
1) $\langle \{x/1\}, \emptyset, (b1 ; \text{false}) \rangle$	Clause 1, indeterministic, i.e. there is a possibility to take clause 2 instead
2) $\langle \{x/1\}, \emptyset, (b1 ; \text{false}) \rangle$	Clause 3
3) $\langle \{x/1\}, (b1 ; \text{false}) \rangle$	Clause 4, which, after identifying $(b1 ; \text{false})$ with $b1$, gives the final solution $b1$.

Answer 2:

0) $\langle \epsilon, \emptyset, \text{false} \rangle$	Initial state
1) $\langle \epsilon, \{p(1)\}, \text{false} \rangle$	Clause 2, indeterministic, i.e. there is a possibility to take clause 1 instead
2) $\langle \{x/2\}, \{p(1)\}, (b2 ; \text{false}) \rangle$	Clause 1
3) $\langle \{x/2\}, (b2 ; \text{false}) \rangle$	Clause 4, which gives the final solution, where the check $\neg \exists \tau (\tau = mgu(p(2), p(1)))$ is satisfied.

3.2.2 Soundness

That the substitutions that this algorithm computes are also computed by the first algorithm is easy to see. There are just some permutations that are cut off. That all substitutions that are computed by the first algorithm are also computed by this algorithm is not as clear. We just give an informal comment, where the details are left out.

For every two terms T_1 and T_2 that are unifiable, there are two possibilities considered by the first algorithm: $\tau = mgu(T_1, T_2)$ and $\sigma = mgu(T_2, T_1)$. τ and σ will be considered equal since $T_1\tau = T_1\sigma$ and $T_2\tau = T_2\sigma$ (modulo variable renaming). Therefore, it suffices to consider just one case of the two possibilities. By ordering the heads, we can impose the condition that just one of the two possibilities is considered, which is done implicitly by the accumulating set S combined with the redundancy check performed in the last clause of the algorithm.

3.3 Algorithm 3

Algorithm 1 and 2 were performed at runtime, i.e. given an atom A , they calculate an A -sufficient substitution. It is possible to calculate all possible A -sufficient substitutions for all possible A 's given the definition beforehand. This can be seen as a compilation of the definition, from the definition itself to a new representation of the definition, which holds the possible defining conditions and A -sufficient substitutions. It can also be seen as partially evaluating algorithm 2 with respect to a particular definition $\mathcal{D}$.

The algorithm that uses the calculated representation has complexity $O(2^n)$, but as shown in section 3.4, the actual numbers of mgu 's computed in runtime is less than the other two algorithms.

3.3.1 Definition

Compared to algorithm 2, the first pass of algorithm 3 delays the fourth clause of algorithm 2 until runtime. A variable is given as input instead of an atom A , and all possible A -sufficient substitutions are generated combined with their delayed redundancy-checks.

We will use ' $\backslash +$ ' as cannot-prove, i.e. negation as failure. It will only be used together with unifiability, i.e. '=', and is used to express "not unifiable with", so $\backslash + (T_1 = T_2)$ is equal to $\neg \exists \tau (\tau = mgu(T_1, T_2))$. If T_2 does not share variables with any other term considered, which is the case when T_2 is the head of some clause in the definition, $\backslash + (T_1 = T_2)$ is the same as $\neg \exists y_1, \dots, y_n (T_1 = T_2)$ where $y_1, \dots, y_n$ are the variables in T_2 , i.e. T_1 should not be an instance of T_2 . $\mathcal{S}$ is defined to be a comma-separated vector instead of a set, as it was in algorithm 2. This is to reflect that the check of the elements in $\mathcal{S}$ is postponed until runtime.

Start by partially evaluating algorithm 2 with respect to a given definition $\mathcal{D}$. The heads that should be considered are all heads of the definition $\mathcal{D}$. Call this set $\mathcal{L}$. Let k denote the cardinality of $\mathcal{L}$. Now consider the ordered list $H_1, \dots, H_k$ of $\mathcal{L}$ and define the following algorithm for generating all possible A -sufficient substitutions

- 0) $\langle \sigma_0, \mathcal{S}_0, \mathcal{B}_0 \rangle = \langle \epsilon, \emptyset, \text{false} \rangle$
- 1) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \tau, \mathcal{S}_m, (B\tau; \mathcal{B}_m) \rangle$ if $\tau = mgu(A\sigma_m, H_{m+1})$
- 2) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m, (\backslash + (A = H_{m+1}), \mathcal{S}_m), \mathcal{B}_m \rangle$ if $\tau = mgu(A\sigma_m, H_{m+1})$
- 3) $\langle \sigma_{m+1}, \mathcal{S}_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m, \mathcal{S}_m, \mathcal{B}_m \rangle$ if $\neg \exists \tau (\tau = mgu(A\sigma_m, H_{m+1}))$
- 4) $\langle \sigma, \mathcal{R}, \mathcal{B} \rangle = \langle \sigma_k, \mathcal{S}_k \sigma_k, \mathcal{B}_k \rangle$

Do a complete search over all the possible A -sufficient substitutions, number all the possible triples $\langle \sigma, \mathcal{R}, \mathcal{B} \rangle$ from 1 to n and let

$$\mathcal{A} = \{ \mathcal{D}(A\sigma_i, \mathcal{B}_i \sigma_i) :- \mathcal{R}_i \mid 1 \leq i \leq n \}$$

where $\mathcal{B}_i$ is the definiens of $A\sigma_i$, and the (Prolog) relation $\mathcal{D}$ holds the possible instances of A together with its corresponding $\mathcal{B}$'s. Thus $\mathcal{D}$ is a representation of $\text{succ}_{\mathcal{D}}(A)$, where $\mathcal{R}$ contains the restrictions on when this particular $\text{succ}_{\mathcal{D}}$ is applicable. $\mathcal{A}$ as a whole contains all possible defining conditions and A -sufficient substitutions for all possible A 's.

3.3.2 Simplifications

$\mathcal{A}$ is perhaps not the least set of clauses that represents the set of defining conditions, there could be clauses whose bodies, $\mathcal{R}$, can never be fulfilled. Such clauses are clauses where there exists a restriction $\backslash + (A = H)$ in $\mathcal{R}$ where A is an instance of H . Since A and H will always be unifiable, $\backslash + (A = H)$ will always fail, and such clauses can be removed safely. Furthermore, there could be redundant restrictions in $\mathcal{R}$. The restriction $\backslash + (A = H)$ in $\mathcal{R}$ where there does not exist a unifier of A and H will always fail, and therefore $\backslash + (A = H)$ will always succeed, and such restrictions can therefore be removed safely.

3.3.3 Example

1) Consider again the definition

$$\begin{aligned} p(1) &\leq b_1. \\ p(X) &\leq b_2(X). \end{aligned}$$

The new representation is

$$\begin{aligned} \mathcal{D}(p(1), (b_1; b_2)). \\ \mathcal{D}(p(X), b_2) :- \backslash + (p(X) = p(1)). \end{aligned}$$

3.3.4 Soundness

Compared to algorithm 2 the possible A -sufficient substitutions have been computed in advance. The heads that were unifiable with A in algorithm 2 are represented in the new $A\sigma_i$ term in the new representation, and those heads that should not be unifiable with the term A are represented in the body $R_i\sigma_i$ of the new representation. Thus there is a one-to-one correspondence between the new representation and the answers that are computed by algorithm 2.

3.4 Comparison of the Algorithms

Below is a table showing the number of times the mgu operation is entered (both successful and failed tries) at runtime in computing all possible A -sufficient substitutions in the three different algorithms. n is the number of clauses considered.

n	Algorithm 1	Algorithm 2 Best case	Algorithm 2 Worst case	Algorithm 3 Best case	Algorithm 3 Worst case
1	1	2	2	2	2
2	4	6	6	3	4
3	18	13	14	4	8
4	96	23	30	5	16
5	600	36	62	6	32
n	$n*n!$	$n(3n-1)/2+1$	$2^{n+1}-2$	$n+1$	2^n

Best case is when no head is unifiable with any other head, in which case there are as many possible A -sufficient substitutions as there are clauses, plus one for the case when no head was unifiable with A .

Worst case is when all heads are unifiable with each other, but differ from each other at some place, which gives 2^n possible A -sufficient substitutions, depending on A . For example, the definition

$$\begin{aligned} p(1, Y, Z) &\leq b1. \\ p(X, 2, Z) &\leq b2. \\ p(X, Y, 3) &\leq b3. \end{aligned}$$

is such a case for $n = 3$. One can note that the third algorithm computes exactly the needed 2^n cases.

Note that more mgu's are calculated in algorithm 2 than in algorithm 1 when n is less than 3, but when n equals 2 algorithm 1 often computes two equal substitutions and thus gives the GCLA system less efficient overall behaviour.

4. Algorithms Incorporating Guards and Constraints

In section 3 we dealt with A -sufficient substitutions without constraints. Since clauses in GCLA can have a guard putting constraints on variables, the algorithms must cope with those too. Thus, we are now going to generalise the algorithms to deal with *constraint sets* instead of substitutions. We will look upon a constraint set as a specification of a possible substitution, which means that a set of constraints should always have at least one solution, i.e. it should represent at least one grounding substitution. We will only deal with one type of constraint, 'not unifiable with' or more precisely 'not instance of', which is denoted by the relation ' $\backslash=$ '. However, one can think of other kinds of constraints that can be added, such as type conditions (for example x must be a number), ordering conditions (for example x must be greater than some y given an order relation)

etc. But when adding a new type of constraint, satisfiability in the new constraint system must be decidable.

4.1 Guards and Constraints

We will use the same terminology as described in section 2, with the following additional definitions and changes.

4.1.1 Unification Constraints and Guards

The constraint $x \neq \text{Template}$ is a restriction on x meaning that x should not be an instance of Template . We define $x \neq T$ as $\forall y_1, \dots, y_n (x \neq T)$, where $y_1, \dots, y_n$ are the variables in T . x may be any term. For example

- 1) $(x \neq s(y))\sigma$, where $\sigma = \{x/s(z)\}$, is not satisfiable, since we are looking for an x such that x is not an instance of $s(y)$, for all y , but x is $s(z)$, and $s(z)$ is an instance of $s(y)$, for all values of z .
- 2) $(x \neq s(y))\sigma$, where $\sigma = \{x/0\}$, is satisfiable, since $0 \neq s(y)$, for all y .
- 3) $(x \neq s(0))\sigma$ where $\sigma = \{x/s(y)\}$ is satisfiable provided that y is constrained to $(y \neq 0)$.
- 4) $(x \neq s(y))\sigma$ where $\sigma = \{x = s(0)\}$ is not satisfiable, since we are looking for an x such that x is not an instance of $s(y)$, for all y , and $s(0)$ is an instance of $s(y)$.
- 5) $(x \neq f(1, 2))\sigma$ where $\sigma = \{x = f(y, z)\}$ is satisfiable provided that at least one of y and z are constrained to $(y \neq 1)$ and $(z \neq 2)$ respectively, i.e. $((y \neq 1) \vee (z \neq 2))$.

If $g_1, \dots, g_n$ are constraints, then $\{g_1, \dots, g_n\}$ is a *guard*. We will use the letter G to denote a guard.

If A is an atom, C is a condition, and G is a guard, then $A \# G \leq C$ is a *guarded clause*. We will as before use H to denote the head of a clause and B to denote the body of a clause. We will often call a guarded clause just a clause.

A set of equalities and inequalities will be called a *constraint set*, and will be denoted by $\theta, \zeta, \sigma, \tau$ or γ , possibly with subscripts or superscripts. These correspond to substitutions in section 2 and 3.

4.1.2 The Definiens Operation and A -sufficient Constraint Sets

We need a new operation, $\bullet$, for combining constraint sets, corresponding to combining substitutions.

Define $\bullet$ as:

$$\sigma \bullet \tau = \begin{cases} \sigma \cup \tau & \text{if } \sigma \cup \tau \text{ is satisfiable} \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ is the unsatisfiable constraint. That $\bullet$ is associative is easy to show. We can note that when two substitutions σ and τ are combined, if σ and τ replace a variable X with t_1 and t_2 respectively, and assume $t_1 \neq t_2$, the combined substitution $\sigma\tau$ defines X to be replaced with t_1 . In the constraint set analogue $\sigma \bullet \tau$, $\sigma \bullet \tau$ is defined to be $\perp$.

Furthermore, define $\text{comp}(A = H \# G)$ as

$$\text{comp}(A = H \# G) = \{ \{A \neq H\}, \{A = H, X_i \neq T_i \mid X_i \neq T_i \in G\} \}$$

$comp$ is the complement of $\{A = H\} \cup G$, i.e. the negation of $A=H \ \& \ g_1 \ \& \dots \ \& \ g_n$, where $g_i \in G$, which is $(A \neq H \vee A = H \ \& \ (g_1 \vee \dots \vee g_n))$. To see this, consider the following informal discussion with the guard $\{X \neq T\}$.

$\{A = H, X \neq T\}$ represents at least one of all grounding substitutions σ such that $\{A = H, X \neq T\}\sigma$ holds. Let the function $subst$ of a constraint set have the following definition:

$$subst(\mathcal{M}) = \{\sigma \mid \mathcal{M}\sigma \text{ holds and } \mathcal{M}\sigma \text{ is ground}\}$$

i.e. $subst(\mathcal{M})$ is the set of all possible grounding substitutions such that $\mathcal{M}\sigma$ holds.

Then $subst(\{A = H, X \neq T\}) = \{\sigma \mid A\sigma = H\sigma, X\sigma \neq T\sigma\}$. The complement to $subst(\{A = H, X \neq T\})$ is then $\{\tau \mid \tau \notin subst(\{A = H, X \neq T\})\} = \{\tau \mid \tau \in subst(\{A \neq H\}) \cup subst(\{X = T\})\} = \{\tau \mid A\tau \neq H\tau\} \cup \{\tau \mid X\tau = T\tau\}$, and, without loss of generality, we get $\{\tau \mid A\tau \neq H\tau\} \cup \{\tau \mid A\tau = H\tau, X\tau = T\tau\}$, which is represented as $comp(A = H \# \{X \neq T\})$.

We define the definiens operation for constraint sets as:

$$\mathcal{D}(A, \theta) = \{\langle B, \{A = H\} \cup G \rangle \mid H \# G \leq B \in \mathcal{D} \text{ and } \{H = A\} \bullet G \bullet \theta \neq \perp\}$$

or in words

$\mathcal{D}(A, \theta)$ is the set of all pairs of bodies B and sets $\{A = H\} \cup G$ such that $\theta \cup \{A = H\} \cup G$ is satisfiable, where θ is a (satisfiable) constraint set. If θ is unsatisfiable, $\mathcal{D}(A, \theta)$ is undefined.

We will need a set consisting of only the bodies B of a definiens $\mathcal{D}(A)$. Define $\mathcal{D}^b(A, \theta)$ as

$$\mathcal{D}^b(A, \theta) = \{B \mid \langle B, \sigma \rangle \in \mathcal{D}(A, \theta)\}$$

where σ is a constraint set.

In the proof we assume that the bodies of the clauses do not introduce new, free variables (i.e. that variables in the body also occur in the head of a clause), in which case the proofs below do not hold, i.e. we assume that the no-extra-variable condition is fulfilled. We also assume that variables in clauses can be freely renamed.

We have to prove that the new definiens operation $\mathcal{D}$ is closed under $\bullet$, i.e. the condition corresponding to $\mathcal{D}(A\sigma)\tau = \mathcal{D}(A\sigma\tau)$ in the substitution case.

$$\mathcal{D}^b(A, \theta) = \mathcal{D}^b(A, \theta \bullet \tau), \text{ for all } \tau \text{ such that } \theta \bullet \tau \neq \perp$$

and for the calculated A -sufficient constraint set σ ;

$$(*) \quad \mathcal{D}^b(A, \theta \bullet \sigma) = \mathcal{D}^b(A, \theta \bullet \sigma \bullet \tau), \text{ for all } \tau \text{ such that } \theta \bullet \sigma \bullet \tau \neq \perp.$$

We will show that $(*)$ can be replaced by another condition, namely that the calculated constraint set σ is total, where total is defined as:

A constraint set σ is *total* if for every clause $H \# G \leq B$, either $\{A = H\} \cup G \cup \sigma$ is satisfiable or $\tau \cup \sigma$ is satisfiable, where $\tau \in comp(A = H \# G)$.

Proposition: If σ is total, then the condition (*) is satisfied.

Proof: $\mathcal{D}^b(A, \theta \cdot \sigma) = \{B \mid H \# G \leq B \in \mathcal{D} \text{ and } \{H = A\} \cdot G \cdot \theta \cdot \sigma \neq \perp\}$, and since σ is total (hypothesis), either $\{A = H\} \cup G$ is in σ or an element of $\text{comp}(A = H \# G)$ is in σ , for all heads H and guards G such that $H \# G \leq B \in \mathcal{D}$. This means that the only sets $\{A = H\} \cup G$ for which $\{H = A\} \cdot G \cdot \theta \cdot \sigma \neq \perp$ holds are those terms where $\{A = H\} \cdot G \subseteq \sigma$. Since $\theta \cdot \sigma \cdot \tau \neq \perp$ holds (see (*)), we can conclude that $\{H = A\} \cdot G \cdot \tau \neq \perp$, for all sets $\{H = A\} \cup G \subseteq \sigma$. Therefore $\{H = A\} \cdot G \cdot \theta \cdot \sigma \cdot \tau \neq \perp$ for all sets $\{H = A\} \cdot G \cdot \theta \cdot \sigma \neq \perp$, and therefore $\mathcal{D}^b(A, \theta \cdot \sigma) \subseteq \mathcal{D}^b(A, \theta \cdot \sigma \cdot \tau)$.

That the cardinality of $\mathcal{D}(A, \theta \cdot \sigma \cdot \tau)$, denoted by $\#(\mathcal{D}(A, \theta \cdot \sigma \cdot \tau))$, is not greater than $\#(\mathcal{D}(A, \theta \cdot \sigma))$ is easy to show. Since σ is total, it holds that for all H, G, B such that $H \# G \leq B \in \mathcal{D}$, if $\sigma \cdot \{H = A\} \cdot G = \perp$, then $\sigma \cdot \{H = A\} \cdot G \cdot \tau = \perp$ (intuitively, τ cannot make $\mathcal{D}(A, \theta \cdot \sigma)$ larger). Thus $\#(\mathcal{D}(A, \theta \cdot \sigma \cdot \tau)) \leq \#(\mathcal{D}(A, \theta \cdot \sigma))$. Together with $\mathcal{D}^b(A, \theta \cdot \sigma) \subseteq \mathcal{D}^b(A, \theta \cdot \sigma \cdot \tau)$ we have $\mathcal{D}^b(A, \theta \cdot \sigma) = \mathcal{D}^b(A, \theta \cdot \sigma \cdot \tau)$.

Q.E.D.

We have defined and proved the condition (*) for a constraint system where the only condition on the algorithm is that it produces a total constraint set σ , i.e. that it considers all clauses of a definition $\mathcal{D}$.

With the new definitions of the definiens operation and A -sufficient constraint sets, the two inference rules $\vdash \mathcal{D}$ and $\mathcal{D} \vdash$ are changed. Instead of applying the substitutions on the conditions directly, we have an accumulating set of constraints, that must be satisfiable at each step:

$$\frac{\Gamma \vdash B \quad \{H = A\} \cup G \cup \gamma}{\Gamma \vdash A \quad \gamma} \vdash \mathcal{D}$$

if $(H \# G \leq B) \in \mathcal{D}$, and $\{H = A\} \cup G \cup \gamma$ is satisfiable.

and

$$\frac{\{\Gamma_1, B, \Gamma_2 \vdash C \quad \zeta \cup \sigma \cup \gamma \mid \langle B, \zeta \rangle \in \mathcal{D}(A, \sigma \cup \gamma)\}}{\Gamma_1, A, \Gamma_2 \vdash C \quad \gamma} \mathcal{D} \vdash$$

if σ is an A -sufficient constraint set with respect to $\mathcal{D}$, $\mathcal{D}(A, \sigma \cup \gamma)$ is the definiens operation and $\sigma \cup \gamma$ is satisfiable.

4.1.3 Satisfiability

The equality theory we will use has syntactic equality, $T_1 = T_2$, and syntactic inequality, $T_1 \neq T_2$, or more precisely, T_1 is not an instance of T_2 . To see that satisfiability of a set of equalities and inequalities is computable and terminating, consider the algorithm below, based on Herbrands algorithm [LMM88] for computing a unifier of a set of equalities. For other algorithms see [Wal87] and [Smi91]. Especially [Smi91], where the notion of U-constraints (universally quantified disequalities) is described, is very close to the satisfiability discussed here, although he considers inequalities where arbitrary arguments of terms can be universally declared variables (see section 5).

Non-deterministically choose an equation from the equation set to which a numbered step applies. The action taken by the algorithm is determined by the form of the equation:

- 1) $f(t_1, \dots, t_n) = f(s_1, \dots, s_n)$
replace by the equations $t_1 = s_1, \dots, t_n = s_n$.
- 2) $f(t_1, \dots, t_n) = g(s_1, \dots, s_m)$ where $f \neq g$
halt with failure
- 3) $x = x$ delete the equation
- 4) $t = x$ where t is not a variable
replace by the equation $x = t$
- 5) $x = t$ where $t \neq x$ and x has another occurrence in the set of equations
if x appears in t then halt with failure
otherwise replace x by t in every other term of the set.

The algorithm terminates when no step can be applied or when failure has been returned.

An equation set (possibly empty) is in *solved form* if it has the form $\{v_1 = t_1, \dots, v_n = t_n\}$ and the v_i 's are distinct variables which do not occur in the right hand side of any equation (see [LMM88] for further details). That this algorithm terminates and leaves the set in solved form is proved in [LMM88].

Our algorithm for determining satisfiability of a set of equalities and inequalities has two passes. The first pass performs Herbrand's unification algorithm with a sixth clause:

- 6) $x \neq t$ Skip it

and a slightly changed terminating clause:

The algorithm terminates when no one of the clauses 1 to 5 can be applied or when failure has been returned.

The second pass tests the inequalities for satisfiability. We assume some ordering $\{T_1, \dots, T_n\}$ of the elements in the set:

Traverse the set of equations and constraints. For every element in the set, one of the clauses below should be applied:

- 7) $x = t$ If it is an equation, skip it.
- 8) $t_1 \neq t_2$ If it is a constraint, check it:
 - If t_1 is an instance of t_2 , then halt with failure.
 - If $t_1 \neq t_2$ is satisfied, i.e. they are not unifiable, then t_1 will always differ from t_2 , and $t_1 \neq t_2$ can be removed. To check unifiability, use Herbrand's algorithm.
 - Otherwise, keep $t_1 \neq t_2$ and continue.

To check if a term t is an instance of t' , the following algorithm can be used:

Replace all variables in t with new constants. Then use Herbrand's algorithm to check if t and t' are unifiable. If so, t is an instance of t' , otherwise not.

It is easy to show that the second pass terminates since it is a one pass traversing and the instance procedure uses Herbrand's algorithm, which is proved to terminate.

This shows that checking satisfiability of a set of syntactic equalities and inequalities is computable and terminating. Of course there are other, more efficient algorithms. One

such is to use unification combined with corouting for checking inequalities in the same manner as corouting can be used to implement $\text{dif}/2$ in many Prologs, in particular [Smi91] discusses a constraint system which is similar to ours. [LMM88] also treats the solving of systems of equalities and inequalities.

4.2 Algorithm 2 with Constraints

To generalize algorithm 2 to handle constraints the function mgu is replaced with a check for satisfiability, and the algorithm starts with an initial constraint set instead of the empty substitution.

4.2.1 Definition

Let the order of the clauses be determined by a list (an ordered set). The heads H and guards G that should be considered are those for which the set $\{H = A\} \cup G \cup \gamma$ is satisfiable, for a given A and a given initial constraint set γ :

$$\{\langle H, G \rangle \mid (H \# G \leq B) \in \mathcal{D} \text{ and } \{H = A\} \cup G \cup \gamma \text{ is satisfiable}\}$$

Call this set $\mathcal{L}$. Let k denote the cardinality of $\mathcal{L}$, and let γ be the initial set of constraints. Now consider the ordered list $\langle H_1, G_1 \rangle \dots, \langle H_k, G_k \rangle$ of $\mathcal{L}$. Let $\mathcal{B}$ be a sum which collects those bodies that are part of the definiens, and define the following algorithm for calculating $\text{suff}\mathcal{D}$:

- 0) $\langle \sigma_0, \mathcal{B}_0 \rangle = \langle \emptyset, \text{false} \rangle$
- 1) $\langle \sigma_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \cup \{A = H_{m+1}\} \cup G_{m+1}, (B_{m+1} ; \mathcal{B}_m) \rangle$
if $\gamma \cup \sigma_m \cup \{A = H_{m+1}\} \cup G_{m+1}$ is satisfiable
- 2) $\langle \sigma_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \cup \tau, \mathcal{B}_m \rangle$
if $\gamma \cup \sigma_m \cup \tau$ is satisfiable
where $\tau \in \text{comp}(A = H_{m+1} \# G_{m+1})$
- 4) $\langle \sigma, \mathcal{B} \rangle = \langle \sigma_k, \mathcal{B}_k \rangle$

Note that the step when choosing between 1) and 2) could be nondeterministic, and by always trying 1) first, we get a natural order of the solutions. Also note the use of comp in clause 2), which is indeterministic if the guard is not empty (see the definition of comp , section 4.1.2), and one of the possible negations is chosen. By performing a complete search, all possible A -sufficient constraint sets can be generated.

If one compares this algorithm with the corresponding one for substitutions, one could note that the set $\mathcal{S}$ and clause 3) have disappeared. This is due to the fact that the constraint system now handles the case when a head H_i is not to contribute positively (i.e. $A = H_i$) in a solution, which was solved with the accumulating set $\mathcal{S}$ and clause 3) before. This also means that other clauses whose bodies are not part of the definiens contribute (negatively) to the A -sufficient constraint set. In other words, the algorithm is total in the sense of section 4.1.

4.2.2 Example

1) Consider the definition

$$\begin{aligned} p(1) &\leq b_1. \\ p(x) \# \{x \neq 1\} &\leq b_2(x). \end{aligned}$$

There are two possible $p(z)$ -sufficient constraint sets $\{z = 1\}$ and $\{z = x, x \neq 1\}$, and corresponding defining conditions $\langle b_1, \{p(z) = p(1)\} \rangle$ and $\langle b_2(x), \{p(z) = p(x), x \neq 1\} \rangle$ respectively.

4.3 Algorithm 3 with Constraints

As for the second algorithm, the largest difference between algorithm 3 in section 3.3 and this algorithm is that the set S , which incrementally collected the heads which should not be unifiable with A , is removed, and instead of adding heads to S in the second clause of the algorithm, their negation is put in the constraint set.

4.3.1 Definition

Partially evaluate algorithm 2 with respect to a given definition $\mathcal{D}$. The heads and guards that should be considered are all heads and guards in the clauses of the definition. Call this set $\mathcal{L}$. Let k denote the cardinality of $\mathcal{L}$. Now consider the ordered list of pairs $\langle H_1, G_1 \rangle, \dots, \langle H_k, G_k \rangle$ of $\mathcal{L}$ and define the following algorithm for generating all possible A -sufficient constraint sets:

- 0) $\langle \sigma_0, \mathcal{B}_0 \rangle = \langle \emptyset, \text{false} \rangle$
- 1) $\langle \sigma_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \cup \{A = H_{m+1}\} \cup G_{m+1}, (\mathcal{B}_{m+1}; \mathcal{B}_m) \rangle$
if $\sigma_m \cup \{A = H_{m+1}\} \cup G_{m+1}$ is satisfiable
- 2) $\langle \sigma_{m+1}, \mathcal{B}_{m+1} \rangle = \langle \sigma_m \cup \tau, \mathcal{B}_m \rangle$
if $\sigma_m \cup \tau$ is satisfiable
where $\tau \in \text{comp}(A = H_{m+1})$
- 3) $\langle \sigma, \mathcal{B} \rangle = \langle \sigma_k, \mathcal{B}_k \rangle$

Number all possible constraint sets σ from 1 to n and let

$$\mathcal{A} = \{D(A, \langle \mathcal{B}_i, \sigma_i \rangle) \mid 1 \leq i \leq n\}$$

where $\mathcal{B}_i$ is the definiens of A and σ_i the A -sufficient constraint set.

That the algorithm computes a total σ is showed by an induction on the heads of the definition. Either $\{A = H\} \cup G$ is in σ , or $\{A = H\} \cdot G \cdot \sigma = \perp$, for all H, G such that $H \# G \leq B \in \mathcal{D}$.

4.3.2 Examples

All the examples show the constraint sets and the new representation without any simplifications.

1) Now it is possible to get answers to negated atoms by a constraint set, which specifies when there is no clause applicable. For example, the definition

$$p(1) \leq b_1.$$

is compiled into

$$\begin{aligned} D(p(X), \langle b_1, \{p(X) = p(1)\} \rangle) . \\ D(p(X), \langle \text{false}, \{p(X) \neq p(1)\} \rangle) . \end{aligned}$$

and to the query $p(Y) \setminus \text{false}$ (which is posing a query 'not $p(Y)$ ' in GCLA), we get as an answer constraint $p(Y) \neq p(1)$ (which equals $Y \neq 1$).

2) Consider the definition:

$$x = x.$$

This is compiled into

```
D(Z=Y, <true, {(Z=Y) = (X=X)}>).
D(Z=Y, <false, {(Z=Y) \= (X=X)}>).
```

i.e. $z=y$ is false if $z=y$ is not an instance of $x=x$, i.e. z is not equal y . This is the same as $\text{dif}(X, Y)$, a primitive in some Prologs, which constrains x and y not to be equal.

3) Another example is the relation `member/2`,

```
member(E, [E|_]).
member(E, [F|R]) # { (F=E) \= (X=X) } <= member(E, R).
```

which is compiled into

```
D(member(X,Y), <true, {member(X,Y) = member(E, [E|_])}>).
D(member(X,Y),
  <member(E,R), {member(X,Y) = member(E, [F|R]),
    (F=E) \= (Z=Z)}>).
D(member(X,Y),
  <false, {member(X,Y) \= member(E, [E|_]),
    member(X,Y) \= member(E, [F|R])}>).
D(member(X,Y),
  <false, {member(X,Y) \= member(E, [E|_]),
    member(X,Y) = member(E, [F|R]),
    (F=E) \= (Z=Z)}>).
```

4) Consider the small database

```
elephant(dumbo).
elephant(jumbo).
```

Compiling this gives the definiens representation

```
D(elephant(X), <true, {elephant(X) = elephant(dumbo)}>).
D(elephant(X), <true, {elephant(X) = elephant(jumbo)}>).
D(elephant(X),
  <false, {elephant(X) \= elephant(dumbo),
    elephant(X) \= elephant(jumbo)}>).
```

4.4. A Note on Negation

Negation is one application of the $\mathcal{D}^+$ rule, where $\text{succ}_{\mathcal{D}}$ is used. There are other cases where this possibility of answering by constraints can be utilized, but negation is one of the most important we have discovered so far. The examples of the previous section showed how definitions were completed with clauses defining the false cases.

It is our conviction that the definiens operation together with not-equal constraints gives a dual inference rule to SLD resolution. When the true-right rule and the d-right rule, which is GCLA's analogue to SLD resolution, are the only rules used, the proofs must end with the symbol `true` to the right. When performing proof-search for a goal $\vdash C$, we try to end the proofs by finding an applicable clause whose body is the condition `true`. The substitution produced during the proof search is the constraint on the variables for the proof to be valid.

When a term is negated, there should be a dual concept to SLD resolution, and we should try to end the proof of a goal $C \vdash \text{false}$ with the dual condition to `true`, `false`, to the left of $\vdash$. In GCLA the dual concept is realized by the rules false-left and d-left. The dual proof search is then trying to find a substitution (or a constraint which represents at least one substitution) which makes C false with respect to the definition. If, then, the definition is incomplete, the proof can be ended if a constraint set is produced which

makes C have no definition, which is the dual to the d-right rule where the proof search tries to find a definition of C that makes it true. What makes this differ from negation by failure is that we actually try to find (a representation of) a substitution that makes C false with respect to the definition. This is what the definiens operation is capable of together with the algorithms presented above for solving satisfiability of constraint sets consisting of equalities and inequalities.

What also makes it a complete dual is that if the definition of a term is complete without any clause with a body `false`, corresponding to the dual case to the right of $\vdash$ where there is no clause with body `true`, the proof search continues by replacing the term by its definition.

An example which illustrates the discussion above is given by considering the definition of `member` in the previous section. The query

$$\text{true} \vdash \text{member}(A, B)$$

has as the first answer $B = [A]$. By backtracking, the next answer $B = [_, A]$ is found. The third answer is $B = [_, _, A]$. This generation could then go on further. The dual query is

$$\text{member}(A, B) \vdash \text{false}.$$

which as first answer gives $B \setminus = [_ | _]$, which means that A is not a member of for example the empty list or a term. The next answer is $B = [C | D]$, $A \setminus = C$, $D \setminus = [_ | _]$. Then this generation could also go on further, generating a representation of the dual to the substitution of the first query.

The two corresponding proofs of the second generated answer shows the duality:

$\frac{}{\text{true} \vdash \text{true}}$	$\frac{}{\text{false} \vdash \text{false}}$
$\frac{}{\text{true} \vdash \text{member}(A, [A])}$	$\frac{}{\text{member}(A, D) \vdash \text{false}}$
$\frac{}{\text{true} \vdash \text{member}(A, [_ A])}$	$\frac{}{\text{member}(A, [C D]) \vdash \text{false}}$

We believe that the definiens operation together with a satisfiability algorithm for systems of equality and inequality gives a dual concept to SLD resolution for finding representations of substitutions that makes the negated term hold to a given definition.

5. Related Work

Since the definiens operation and the notion of A -sufficiency are local to GCLA and Partial inductive definitions (to our knowledge), there is not very much work done elsewhere. However, when incorporating satisfiability of equalities and inequalities, there is a lot of related work in the field of constructive negation (c.f. [MN89, Dra91, Kun87, Wal87, Har91], among others). In our case this is the generation of constraints which correspond to an empty definiens, i.e. the A -sufficient constraint set to which the definiens operation is empty.

[LMM88] gives a very thorough treatment of substitutions and unifiers of set of equalities and inequalities.

[Smi91] defines U -constraints as universally quantified disequalities, denoted by $T_1 <> T_2$ where T_1 and/or T_2 can contain universally quantified variables, which is similar to our not-instance-relation $\setminus =$. The difference lies in that in the operation

$T_1 <> T_2$, both T_1 and T_2 can have universally quantified variables, while in the operation $T_1 \setminus = T_2$, all variables in T_2 are universally quantified while all variables in T_1 are existentially quantified. This makes us unable to express the U-constraint $s(? , x) <> z$, where $?$ denotes a universally quantified variable, but makes $\setminus =$ easier to implement, and $\setminus =$ suffices for our purposes. However, there is a possibility to generalize our relation $\setminus =$ to a three-argument relation $\setminus = (Vars, X, Y)$, where $Vars$ holds the universally quantified variables, and X and Y should not be unifiable. The same basic algorithm as in section 4.1.3 can be used, with the following clause replacing 8):

- 8) $\setminus = (V, t_1, t_2)$ If it is a constraint, check it:
- If $t_1 \setminus = t_2$ is satisfied, i.e. they are not unifiable, then t_1 will always differ from t_2 , and $t_1 \setminus = t_2$ can be removed. To check unifiability, use Herbrands algorithm.
 - Replace all variables of t_1 , which are in V , with new constants. Call the new term t_1' . Do the same with t_2 forming t_2' . If t_1' and t_2' are unifiable, then t_1 and t_2 will always be unifiable, and the constraint $\setminus = (V, t_1, t_2)$ can never be fulfilled. Halt with failure.
 - Otherwise, keep $\setminus = (V, t_1, t_2)$ and continue.

We are then able to express that two terms t_1 and t_2 are unequal, where arbitrary variables in t_1 or t_2 could be universally quantified. For example, the Prolog predicate `diff/2` could be expressed either as $\setminus = ([], X, Y)$, or as $\setminus = ([Z], p(X, Y), p(Z, Z))$.

[Dra91] gives the notion of SLDFA resolution, which uses constraints of equalities and inequalities. A negated literal L in a goal is solved by finding a set of constraints for which L is finitely failed.

[Wal87] gives a treatment of negation in logic programming with constraints which in much is very similar with [Dra91].

[Har91] defines the completion of clauses on hereditary Harrop (HH) form, and discusses their properties, in particular that the completion is a HH formula itself, and thus is in the language. The completion is defined as:

Let the clauses defining p be the universal closure of

$$\begin{aligned} p(t_{11}, t_{12}, \dots, t_{1n}) &\Leftarrow B_1. \\ &\dots \\ p(t_{k1}, t_{k2}, \dots, t_{kn}) &\Leftarrow B_k. \end{aligned}$$

Let p^+ be the clause

$$\forall x_1, \dots, x_n p(x_1, \dots, x_n) \Leftarrow \bigvee_{i=1}^k \exists (E_i \ \& \ \text{contr}(B_i))$$

where E_i is $x_1 = t_{i1} \ \& \ \dots \ \& \ x_n = t_{in}$, $\exists(E_i \ \& \ G_i)$ is the existential closure of all free variables of $E_i \ \& \ G_i$, $1 \leq i \leq k$, and $x_1, \dots, x_n$ are new variables not occurring in any G_i . $\text{contr}(C)$ replaces all occurrences of $\neg A$ with $A \rightarrow \perp$.

p^- is the clause

$$\forall x_1, \dots, x_n \perp \Leftarrow p(x_1, \dots, x_n) \ \& \ \bigwedge_{i=1}^k \forall (\neg E_i \vee \text{contr}(\text{fails}(B_i)))$$

where $fails(c)$ negates the formula c . If there is no clause in the program whose head's name is p , then p^+ is empty and p^- is the clause

$$\forall x_1, \dots, x_n \perp \Leftarrow p(x_1, \dots, x_n).$$

The completion of a predicate p is $\{p^+, p^-\}$.

This completion bears close resemblance to the $\mathcal{D}$ -representation defined in section 4.3, in that $(\neg E_1 \vee contr(fails(B_1)))$ is a corresponding disjunction to the nondeterministic choice between clauses 1) and 2) in section 4.2 and 4.3.

6. Remarks and Future Work

Note that the second algorithm has the advantage that it can handle dynamic assert in and retract from the definition, and is thus justified for such cases. But for definitions that are not dynamic, the representation calculated by the third algorithm has much better performance.

It should be possible to develop better indexing algorithms for the $\mathcal{D}$ -representation than just indexing on the first argument's principal functor, as in most Prolog implementations today. If the ' $\backslash =$ ' - constraints could be used for indexing purposes, it would reduce the number of choice points generated, since it is often the case that these constraints make the clauses mutually exclusive.

Another interesting generalisation is to develop a constraint language in which the programmer can define his own constraint system. To do this, the properties discussed in section 4 must be satisfied to guarantee that the definiens operation and the overall GCLA system behaves correctly, but in principle it should be possible. Then the GCLA system should be indexed with the constraint system CS in mind, i.e. in the same way as other constraint logic programming languages, which are denoted by $CLP(CS)$, and the programmer could define a constraint system that is most suitable for his purposes.

Acknowledgement

The author would like to thank Per Kreuger, Lars-Henrik Eriksson and Lars Hallnäs of the FOPKBS group at SICS for many valuable discussions. The elaborate version $\backslash =/3$ of $\backslash =/2$ was especially due to discussions with Per Kreuger. Thanks also to Peter Schroeder-Heister for valuable comments on a draft of this paper, and to Torkel Franzen for improving the language of the paper.

References

- [Aro90] M. Aronsson, L-H Eriksson, A. Gärödal, L. Hallnäs, P. Olin, The Programming Language GCLA: A Definitional Approach to Logic Programming, *New Generation Computing* 7(4), pp 381 - 404, 1990
- [Aro91] M. Aronsson, *GCLA User's manual*, Technical Report SICS T91:21, 1991
- [Aro92a] M. Aronsson, *Methodology and Programming Techniques in GCLAI*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, January 1991, in Springer Lecture Notes in Artificial Intelligence. Also available as SICS Research Report R92:05.

- [Aro92b] M. Aronsson, *Implementation Issues in GCLA II*, forthcoming SICS technical report.
- [Dra91] W. Drabent, *Constructive Negation by Failed Answers*, Research Report LiTH-IDA-R-91-23, Department of Computer and Information Science, Linköping University, 1991.
- [Eri92] Lars-Henrik Eriksson, *A Finitary Version of the Calculus of Partial Inductive Definitions*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, January 1991, in Springer Lecture Notes in Artificial Intelligence. Also available as SICS Research Report R92:08.
- [Hal91] L. Hallnäs, *Partial Inductive Definitions*, Theoretical Computer Science 87, pp 115 - 142, 1991.
- [HS-H90] L. Hallnäs, P. Schroeder-Heister, *A Proof-Theoretic Approach to Logic Programming. I, Clauses as Rules*, Journal of Logic and Computation vol. 1 no. 2, pp 261 - 283, 1990.
- [HS-H91] L. Hallnäs, P. Schroeder-Heister, *A Proof-Theoretic Approach to Logic Programming. II, Programs as Definitions*, Journal of Logic and Computation vol. 1 no. 5, pp 635 - 660, 1991.
- [Har91] J. Harland, *A Clausal Form for the Completion of Logic Programs*, Proc. of the International Conference on Logic Programming no. 8, pp 711 - 725, 1991.
- [Kre92] P. Kreuger, *GCLAII, A Definitional Approach to Control*, Extensions of Logic Programming: Proceedings of a workshop held at SICS, January 1991, in Springer Lecture Notes in Artificial Intelligence. Also available as SICS Research Report R92:09.
- [Kun87] K. Kunen, *Answer Sets and Negation as Failure*, Proc. of the International Conference on Logic Programming no. 4, pp 219 - 228, MIT Press, 1987.
- [LMM88] J-L. Lassez, M.J. Maher, K. Marriott, *Unification Revisited*, in J. Minker, Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, 1988.
- [MN89] J. Maluszynski, T. Näslund, *Fail Substitutions for Negation as Failure*, in Proceedings of the North American Conference on Logic Programming, MIT Press, 1989.
- [Smi91] D.A. Smith, *Constraint Operations for CLP(FT)*, in Proc. of ICLP 8, MIT Press, 1991.
- [Wal87] M. Wallace, *Negation by Constraints: A Sound and Efficient Implementation of Negation in Deductive Databases*, in Proc. of 1987 Symposium on Logic Programming, San Francisco, pp 253 - 2633, 1987.