AUTOMATED REASONING WITH A CONSTRAINT-BASED METAINTERPRETER

HENNING CHRISTIANSEN

▷ Using constraint logic techniques, it is made possible to use a well-known metainterpreter backwards as a device for generating programs. A metainterpreter is developed, which provides a sound and complete implementation of the binary demo predicate. Based on it, a general methodology for automated reasoning is proposed and it turns out that a wide range of reasoning tasks, normally requiring different systems, can be defined in a concise manner in this framework. Examples are shown of abductive and inductive reasoning in the usual first-order setting as well as in contexts of default reasoning and linear logic. Furthermore, examples of diagnosis and natural language analysis are shown. ◁

1. INTRODUCTION

We propose a general methodology for automated reasoning in the setting of metalogical programming. The central component is a fully declarative metainterpreter, reversible in the sense that it is equally well suited for generating programs as for executing them in the normal way. Constraint logic programming techniques are central in order to provide an implementation which is practically relevant and which can be proved to be sound and complete.

Where other systems for automated reasoning usually support a single form of reasoning, our system appears as a highly generic environment in which existing and new techniques can be implemented and combined with each other. We illustrate this by examples of abductive and inductive reasoning in the usual first-order setting as well as in contexts of default reasoning and linear logic. We show also applications in diagnosis and natural language analysis. The definitions of such tasks in our framework appear in a quite concise and declarative manner. We believe the system to be a tool well suited for research and teaching in automated reasoning and logic programming, providing a short turn-around time from the

Address correspondence to Henning Christiansen, Department of Computer Science, Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark. E-mail: `henning@dat.ruc.dk`

PRELIMINARY VERSION, SEPTEMBER 1996

ALL KINDS OF COMMENTS ARE WELLCOME!

conception of an idea to a running prototype. For large and really difficult problems, however, specialized systems with more intelligent search strategies will still be needed. — The implemented system is available electronically at the address `http://www.dat.ruc.dk/software/demo.html`.

In the rest of this introduction we give an overview of the approach and put it into a historical perspective; finally we present a brief outline of the paper.

1.1. The idea

Normally, a metainterpreter is thought of as a predicate that executes object programs given at a metalevel. What we suggest is to use it *backwards* as a way to generate new object programs. Our metainterpreter is a realization of the binary proof predicate demo, which is specified as follows.

$\text{demo}(P', Q')$ iff P' and Q' are ground names of object program and query, P and Q, such that there exists substitution σ with

$$P \vdash Q\sigma$$

A metavariable, say X, in P' will thus stand for a piece of program text and a logically satisfactory implementation, such as our constraint-based version, will produce program fragments which make Q provable. By means of additional side-conditions, demo can be instructed to produce useful programs as illustrated by the following pattern.

$$\text{useful}(X) \wedge \text{demo}(\cdots X \cdots, \cdots)$$

The 'useful' predicate may specify syntactic requirements to the program fragments sought, perhaps extended with additional calls to demo to express integrity constraints. In this way, a choice of 'useful' can define a reasoning method, e.g., abduction or a class of inductive problems, and the remaining parts of the arguments to demo set up the specific problem to be solved.

The object language for our demo predicate consists of positive Horn clauses extended with equality and inequality constraints. Technically, this a minor extension, but inequalities make it possible to express exceptions to a predicate without introducing the problem sphere associated with mechanisms such as negation as failure.

1.2. Getting the thing to work

A straightforward implementation of the demo predicate, nick-named Instance-demo, has been studied by several other authors recently [17, 18, 25, 5] (our version of it is shown in section 2.3). It replicates SLD resolution (see [38]) using the primitive operations specified as follows.

$\text{instance}(S', T', \sigma')$ iff S' and T' are names of terms S and T, σ' name of a substitution σ with $S\sigma = T$.

$\text{member}(C', P')$ iff C' and P' are names of clause and program C and P where C is a member of P.

In the referenced works, programs are represented as list structures with 'member' being the usual list operation; 'instance' is implemented in a similar straightfor-

ward way in traditional logic programming languages such as Prolog or Gödel with substitutions represented as lists of variable-value pairs.

While sufficient when the program argument to demo is fully given (i.e., ground), such naive implementation methods impliy severe problems in the general case we have in mind:

1. Uninstantiated variables in the first argument to an instance constraint will lead the Gödel programs of [25, 5] into floundering states, so no answer is provided. This arises in the maintenance of substitution arguments.
2. With a straightforward implementation in Prolog, a subgoal instance(X, Y, Z), for metavariables X, Y, Z may initiate on backtracking a generation process of all possible names of terms in the object language until one that meets the subsequent subgoals is reached. Obviously, this can easily cause loops.
3. The representation of programs as lists results in infinitely many equivalent solutions produced for essentially the same object program by permutation and duplication of clauses and by insertion of arbitrary numbers of new variables. (To see this, consider the solutions to the Prolog calls `member(a,L), member(b,L)`). Backtracking on failure is thus condemned to loop.

It is obvious to suggest constraint techniques applied for the first two related problems: When not enough information is present in the arguments to 'instance', it should delay and some additional machinery needs to be developed in order to check for satisfiability.

To solve the third problem, we consider programs as *sets* of clauses rather than lists, and viewing also 'member' as a constraint makes it possible to provide a detailed control of which solutions are produced.

Another problem arises when arbitrary patterns with uninstantiated variables are possible in the arguments to 'instance'. As we will show in section 3.3, satisfiability turns out to be closely related to the undecidable semiunification problem. Fortunately, we have identified an invariant property, called safeness, which hold for the constraint sets that actually can occur, and under which our constraint solver is guaranteed to terminate. Roughly, safeness means that no variable can occur in a first as well as in a second argument to 'instance' constraints.

We can show that this constraint solver together with the logic program defining demo comprises a sound and complete implementation of provability, where completeness here is a metalevel statement saying that demo is able to produce *any* program that makes something provable.

The constraint solver is described as a derivation system, which can be mapped into an executable program in Sicstus Prolog [50] using its notion of attributed variables, originally suggested by [30, 31]. Constraints appear as predicates in Prolog with an inherent constraint-behaviour in the sense that they delay and wake up at the right moments together with additional code that takes care of the overall satisfiability.

Soundness and completeness is preserved in our implementation of the constraint solver in Prolog. The overall program structure of the metainterpreter is executed directly by Prolog with the well-known characteristics of general efficiency paired with a lack of completeness due to possible loops. However, in practice this appears not to be a big problem. When demo is applied for a specific task, the defining side-conditions (depicted as the 'useful' predicate above) can behave in a "lazy" way by means of delay mechanisms, which do not affect the declarative nature of

the metalanguage. In this way demo and 'useful' work as two co-routines, and this has proved to be sufficient for all examples we show in this paper.

The main advantage of using the structure of Instance-demo is that we achieve an efficient implementation of object level unification, virtually mapping it into metalevel unification, i.e., in the end into Prolog unification. Operationally, this is very similar to the nonground representation in the Vanilla interpreter, but avoiding the soundness problems with this representation pointed out by [26].

In [5], Instance-demo is compared with another interpreter using an explicit simulation at the metalevel of an object level most-general-unifier operation. Application of object level substitutions to the lists of subgoals waiting to be processed needs also to be simulated at the metalevel. This takes place in each proof step and, as this list can grow arbitrary large, this approach seems quite inefficient. However, [5] points out metaprogramming tasks where such an interpreter may be more appropriate than Instance-demo. In our case demo, when once implemented, serves as a black-box, and for reasons of efficiency Instance-demo seems to be the obvious choice.

It should be stressed that the referenced works consider exclusively the case with fully given program arguments; our use of constraints to extend to the general case seems to be new.

1.3. Related work in a historical perspective

In tracing the principle of formalized proof procedures in logic and computer science, it is relevant to go back to D. Hilbert (1862–1943). He was the most prominent representative of the formalist school around the turn of the century, strongly putting forward that mathematical theorem proving in general could be axiomatized as a way to achieve a complete proof procedure; see, e.g., [23, 24]. With Gödel's first incompleteness theorem in 1931 [20], this optimistic view had to be adjusted.[1]The theorem states that in any logical system capable of expressing number theory there will be true sentences that cannot be proved. The more modest goal pursued in computer science nowadays is to search for special cases where completeness can be obtained, while still allowing for interesting applications (or perhaps searching for incomplete proof procedures for stronger systems, where we are willing to accept a loop now and then).

The notion of a ground representation of an object language at the metalevel was apparent in the proof of Gödel's theorem. He represented formulas of the given logic by numbers. Translating this into computer science terms, it means a representation of programs as data which, thus, can be transformed and elaborated. However, when the purpose is to write interesting programs, a structural representation such as a tree is more convenient than a monolithic number. Programs as data do not appear only in recent work on metaprogramming in logic (as described by [26]), but has been central in computer science throughout its history, consider, e.g., the von Neumann computer architecture [43] with its notion of the stored program, or any traditional compiler, which is a program transforming other programs from one language to another, or we can refer to reflective architectures (see [39] for an overview and references) where Lisp [40] is the historically most important one.

[1] A nontechnical and vivid introduction to Hilbert's work and the period up to Gödel is given by [46]; see [21] for a collection of important original papers.

We now take a jump to Robinson's [48] introduction in 1965 of resolution, which is a complete proof procedure for the subset of first-order logic called Horn clause logic. Later resolution has lead to efficient implementations of logic programming languages such as Prolog [51], where, however, a depth-first control structure is used for reasons of efficiency, but with the consequence of reducing completeness to a hypothetical property. Another important issue of resolution is the emphasis on the logical variable. A query may contain free variables, and completeness implies that the procedure can produce all values for these variables that turn the query into a true statement.

When the principle of the logical variable is combined with formalized provability based on a ground representation of logic programs, the potentiality appears of generating programs automatically from metalevel specifications expressed in terms of provability. Completeness becomes a statement that any program satisfying the metalevel specification can be produced.

The demo predicate, which is a formalization of provability for logic programs, was introduced by Kowalski in his book [36] of 1979. The potentiality for generation of programs as described above was not noticed, but many other practically relevant applications of demo were mentioned; see also later work [4, 3].

A fundamental result concerned with formalized provability was given by Sato [49] in 1992. He gave an axiomatization of provability for full first order logic in itself and proved it to be complete with respect to a three-valued semantics. A proof procedure was obtained by means of a breadth-first interpreter at the metalevel, preserving the completeness in the three-valued setting. However, this approach does not seem suited for generating programs in practice, because the interpreter in this case will perform an effective enumeration of all possible programs until the right one is found. Also in 1992, we presented a resolution method [9] for a language with a construct similar to the demo predicate. This method involved rewriting of equations with function symbols expressing roughly the same thing as the instance constraints, we use in the present paper. However, no practical implementation was given and termination in the rewriting process was not guaranteed as it essentially was approaching an undecidable problem [8]. The implementation method for demo presented in the present paper can be seen as a reformulation of this, but now using efficient constraint techniques and in a context where the safeness condition described above holds, effectively excluding the undecidable cases from the constraint satisfaction problem.

We notice another early suggestion (1990) for using the demo predicate as a device for program synthesis [44], but it seems clear to us that the field of constraint logic programming had to mature and be applied in order to provide a relevant implementation; see [32] for background on constraint logic programming. — An early version of our constraint solver was described in [12] and applications presented in [10, 11, 13, 14].

When the problem is restricted to have demo to synthesize programs from a specified, finite collection of clauses, a straighforward implementation in Prolog will be sufficient; we can mention [6] (also 1990) where new programs are synthesized by set operations such as intersection and union applied to a collection of object programs given in advance.

Now we will consider another thread, studies of reasoning in a logical context; we will skip over early history and go directly to C.S. Peirce (1839–1914). His work [45]

has attracted new interest recently among philosophers and computer scientists, among other reasons because he appears to be the first to postulate deduction, abduction, and induction as being *the* fundamental ways of reasoning.[2]

We can use the demo predicate to give a simplistic characterization of these notions. Assume a metalevel predicate rules($\cdots$) defining the shape of rules for describing general knowledge in Horn clause logic, similarly cases($\cdots$) and observations($\cdots$) and consider the following formula.

(*) rules(X) $\wedge$ cases(Y) $\wedge$ observations(Z) $\wedge$ demo(X & Y, Z)

Purely deductive reasoning is when X and Y are fully specified, i.e., they constitute an established theory in which observations Z have to be verified or predicted. This corresponds to the functionality in a Prolog interpreter and most earlier work on demo [36, 4, 3, 1, 2]. When X and Z are fixed, abduction is obtained, an explanation Y is sought for the observations Z. Induction, which is the process of identifying general rules from a set of examples, is obtained by fixing Y and Z, and perhaps a part of X (so-called background knowledge). In this perspective, demo appears, at least at the level of specification, as a general reasoning device which can be used for all three reasoning forms and actually smoothes out the distinction between them.

In the last decade, new areas called inductive and abductive logic programming have emerged. Most techniques for abduction appear as extensions of SLD or SLDNF resolution (see [38]) with the possibility of deriving new facts, when otherwise a given atom does not match a clause in the program; see [33] for an overview. Within induction, a technique called inverse resolution has been developed. It is defined by a number of derivation rules which together capture an effect similar to using the resolution rule backwards; an overview of this and other techniques in inductive logic programming is given by [42].

In our implementation of demo, resolution appears as a metalevel relation and depending on the given side-conditions and the degree of instantiation of its arguments, it may operationally behave similarly to the modified resolution for abduction or the inverse resolution applied for induction. For the abduction case, consider again the specification (*) above with X and Z ground. If demo runs out of clauses that unify with a give atom A, it will automatically commit the metavariable Y to represent a program with a fact which unifies with A and also satisfies the metalevel predicate cases($\cdots$). Analogously in case of induction, a proof step may introduce a pattern for a new clause whose head is unifiable with the actual atom and constraints about its body will be derived in the subsequent proof steps.

In [33], a notion of abductive frameworks is defined which also includes integrity constraints and we show, section 4.2, how this can be described in our framework. To indicate the generality in the use of demo, we have combined abductive frameworks with induction and default reasoning so that we automatically get produced defaults-with-exceptions from examples, section 4.5. It should be noticed, however, that we provide no support for negation, so negative examples have to be encoded in a suitable way.

[2] Peirce had two different theories concerned with these notions, an early version, by P. Flach [16] called the *syllogistic* theory, which he abandoned later in favour of a new formulation, by Flach called the *inferential* theory, with slightly different meanings associated with the three terms. Here, we refer to the syllogistic version, which is the most common one to assume in computer science literature.

In inductive and abductive reasoning, it may also be important to make a priority among the generated answers. For diagnosis [47], which is a special case of abduction, minimal explanations are preferred, i.e., collections of ground facts such that if one fact is removed, the observed symptoms cannot be explained anymore. Our system does not include a mechanism to express such priorities in general, but we can show, how the minimality requirement in diagnosis can be expressed in a straightforward way by control at the metalevel, section 4.8.

For natural language analysis, [29] has developed a technique called weighted abduction which associates a quality measurement to each possible explanation in order to chose the best one and in induction, various kinds of statistics has been applied in order to chose the best rules. We have not considered adding such facilities to demo, but it may be interesting to apply fuzzy logic at the metalevel for this purpose.

Finally we will mention that there is a long tradition in machine learning working with related problems which we did not include in this review.

1.4. Overview

Section 2 describes the theoretical setting for achieving the complete implementation of demo. We introduce a class of typed constraint logic languages which includes our object and metalanguages $\mathcal{HCL}$ and Clp($\mathcal{HCL}$), the latter in which the demo predicate is programmed. We end this section by the derivation system $\mathcal{DS}$ which serves a constraint solver for Clp($\mathcal{HCL}$).

Section 3 gives the proofs of soundness and completeness of $\mathcal{DS}$ and hence of our implementation of the demo predicate.

The relation between constraint satisfaction and the undecidable semiunification problem is described in more details in appendix A.

Section 4 entitled "Automated reasoning with the demo predicate" explains firstly the implementation in Prolog of $\mathcal{DS}$ and Clp($\mathcal{HCL}$). Then follows a series of examples of reasoning problems defined in the demo system:

- Abductive frameworks.
- Default reasoning.
- Abduction in default reasoning.
- Induction of defaults-with-exceptions from examples.
- Abduction in a fragment of linear logic.
- A natural language example.
- Diagnosis.

2. A CONSTRAINT-BASED, REVERSIBLE METAINTERPRETER

2.1. Typed constraint logic languages

The presence of a naming relation at the metalevel induces a natural classification of metalevel terms, namely those that stand for programs, those that stand for clauses, etc. This makes it obvious to suggest the use of a *typed* metalanguage. Our experience is that the introduction of types leads to a considerably simpler derivation system for solving the sort of constraints that are relevant for metaprogramming tasks. In the following, we introduce a framework for typed constraint logic languages and constraint solving.

We consider a class of constraint logic languages CLP($\mathcal{X}$) similarly to [32], but here adapted for typed languages. In our case, the parameter $\mathcal{X}$ refers to some domain of constraints over terms with no interpreted function symbols.

Each constraint logic language is characterized by a *structure* consisting of

- a finite set of *type identifiers* or *types* for short (called monotypes by [28]),
- a collection of *function symbols* each with an arity n and rank $f\colon \tau_1 * \cdots * \tau_n \to \tau$, where $\tau_1, \ldots \tau_n, \tau$ are types; function symbols of arity 0 are called *constants*.

Furthermore, a language includes

- for each type τ, an infinite collection of *variables* of type τ, and
- disjoint collections of *predicate symbols* and of *constraints symbols*, each of which has an arity n and rank $p\colon \tau_1 * \cdots * \tau_n$, where $\tau_1, \ldots \tau_n$ are types,

Capital letters such as X and Y are used for concrete variables; the underline character '_' is used as an anonymous variable in the sense that each occurrence of it stands for a variable that does not occur elsewhere. A *program* is a finite set of *clauses* of the form $h \leftarrow b_1 \wedge \ldots \wedge b_n$ with h being an atom, each b_i an atom or a constraint, composed in the usual way respecting the ranks of each symbol; a *query* is similar to the body of a clause. Queries and bodies of clauses are collectively called *formulas* and we assume two inclusion operators, both denoted $\uparrow$, from atoms and from constraints to formulas;[3] to simplify the notation, we leave out these operators except in a few, essential cases. Finally, the truth constant *true* is used to indicate the empty body of a fact.

The meaning of the constraints in a given language is assumed given by a set of ground constraints referred to as *satisfied* constraints. We assume, for each type τ, constraint symbols '$=\colon \tau * \tau$' and '$\neq\colon \tau * \tau$' with the usual meanings of syntactic identity and non-identity. To cope with the semantics of inequalities, the set of function symbols is required to be such that there exists infinitely many ground terms for each type.

We assume any substitution to be idempotent corresponding to the sort of answers generated by Prolog and for reasons of technical simplicity, we define satisfiers and answer substitutions to be ground substitutions. The logical semantics is given in terms a proof relation defined for ground queries as follows.

Definition 2.1. The *proof relation* for a constraint language $\mathcal{L} = \text{CLP}(\mathcal{X})$, denoted $\vdash_{\mathcal{L}}$, between programs and ground queries is defined inductively as follows.

- $P \vdash_{\mathcal{L}} \mathit{true}$ for any program P.
- Whenever P has a clause with a ground instance $H \leftarrow B$ such that $P \vdash_{\mathcal{L}} B$, we have $P \vdash_{\mathcal{L}} H$.
- Whenever $P \vdash_{\mathcal{L}} A$ and $P \vdash_{\mathcal{L}} B$, we have $P \vdash_{\mathcal{L}} A \wedge B$.
- $P \vdash_{\mathcal{L}} C$ whenever C is a satisfied constraint of $\mathcal{X}$.

A *correct answer* for a query Q with respect to a program P is a substitution σ for the variables of Q such that $P \vdash_{\mathcal{L}} Q\sigma$.

[3] These inclusion operators make it easier to define a naming relation as part of typed meta-language without having to introduce a notion of subtypes.

A constraint set C is said to be *satisfiable* if there exists a ground substitution μ for the variables of C so that $C\mu$ is satisfied; in this case μ is called a *satisfier* for C. The notation $[\![C]\!]$ refers to the set of all satisfiers for C. The notions of satisfiers and satisfiability is extended to sets of constraints and atoms by saying "A is satisfied (wrt. program P)" whenever $P \vdash_{\mathcal{L}} A$ for any atom or constraint A.

As a procedural semantics, we use top-down derivations in the sense of [32], however here adapted for typed languages. Furthermore, as we have in mind implementations using Prolog-like technology, we assume an indivisible (and efficiently implemented) unification operation.

A *derivation system* consists of *transition rules* $S \rightsquigarrow S'$ over *states* of the form

$$\langle C, \alpha \rangle$$

where C is a finite set of literals and α an *accumulated substitution*, which represents the explicit variable bindings made so far.

States are assumed to be *idempotent* in the sense that no variable $x \in \mathrm{dom}(\alpha)$ occurs in C, i.e., α is mapped consistently over the constraint set C. We extend the notions of satisfaction and satisfiers (with respect to program P) to states by viewing the accumulated substitution as a set of term equations. We assume a special state called FAILURE with set of satisfiers $[\![\text{FAILURE}]\!] = \emptyset$.[4]

We use also $\rightsquigarrow$ to denote the *derivation relation* induced in the natural way by a set of derivation rules $\{\rightsquigarrow\}$; $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$. A state is *final* if there is no derivation possible from it, and a derivation is *successful* if it ends in a final state $\langle C, \alpha \rangle$ where C consists of constraints only. A derivation is *failed* if it ends with FAILURE or a final state which includes an atom.

Whenever $\langle Q, \emptyset \rangle \rightsquigarrow^* S$ is a successful derivation for a query Q, any substitution in $[\![S]\!]$ restricted to the variables of Q is called a *computed answer* for Q.

The following transition rules, labelled (Unif), (Res), (Dif1), (Dif2), and (True), constitute the core of any transition system.

The unification rule (Unif) is the only one that changes the accumulated substitution.

(Unif) $\langle C \cup \{s = t\}, \alpha \rangle \rightsquigarrow \langle C, \alpha \rangle \mu$
— where μ is a most general unifier of s and t chosen so that $\alpha\mu$ is defined and the new state becomes idempotent;
however, if s and t have no unifier, the result is FAILURE.

In any other rule, we will leave out the accumulated substitution assuming it to be copied unchanged. Other rules can of course affect it indirectly by setting up one or more equations as is the case in the following resolution rule. Resolution is only meaningful in the context of some program P.

(Res) $C \cup \{A\} \rightsquigarrow C \cup \{B_1, \ldots, B_n, A = H\}$
— whenever A is an atom, $H \leftarrow B_1 \wedge \ldots \wedge B_n$ a variant with new variables of a clause in P.

In this rule, the equation $A = H$ should be understood as an equation between

[4] An explicit failure state gives a way to identify failure immediately from a constraint such as 'a=b'. The framework of [32] does not provide this notion so the only way to identify failure is to continue execution until no derivation rule is applicable. In case of recursive programs, this can lead to infinite derivations made up from obviously unsatisfiable states.

terms similar to A and H but with the predicate symbols replaced in a consistent way by function symbols.

Occurrences of the truth constant *true* are removed by the following rule.

(True) $C \cup \{true\} \rightsquigarrow C$

We need the following characterization in order to handle inequality constraints.

Definition 2.2. The notion of two terms being *distinguisable* is defined inductively as follows.

- Two terms $f(t_1, \ldots, t_n)$ and $f'(t'_1, \ldots, t'_{n'})$ are distinguisable if f and f' are different.
- Two terms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ are distinguisable if, for some i, that t_i and t'_i are distinguisable.

The following two rules define a behaviour of inequalities quite similarly to the `dif` predicate of Sicstus Prolog [50].

(Dif1) $C \cup \{t_1 \neq t_2\} \rightsquigarrow C$
— whenever t_1 and t_2 are distinguisable.

(Dif2) $C \cup \{t_1 \neq t_2\} \rightsquigarrow$ FAILURE
— whenever t_1 and t_2 are identical.

So, for example, the inequality f(X,a)≠f(X,b) is reduced away by (Dif1), f(X)≠f(X) reduces to FAILURE by (Dif2), whereas f(X)≠f(Y) and f(X,a)≠f(b,Y) cannot be reduced.

Analogously with [32] we give the following definition.

Definition 2.3. A *derivation system for* a constraint logic language CLP($\mathcal{X}$) is a set of transition rules $\{\rightsquigarrow\}$ which includes (Res), (Unif), (True), and (Dif1–2) such that the conditions (*i*) and (*ii*) hold.

(*i*) (Preservation of satisfiers)
For any state S, let $\{S_i\}_{i \in I}$ be the set of all states with $S \rightsquigarrow S_i$. Then the set of substitutions $\bigcup_{i \in I} [\![S_i]\!]$, each restricted to the variables of S, coincides with $[\![S]\!]$.

(*ii*) (Termination in constraint solving)
For any state of the form $S = \langle C, \alpha \rangle$, with C consisting of constraints only, any derivation from S is finite and any maximal derivation $S \rightsquigarrow^* S'$ ends with either a satisfiable state $S' = \langle C', \alpha' \rangle$ or $S' =$ FAILURE.

By induction, we can prove the following soundness and completeness result.

Proposition 2.1. Let $\{\rightsquigarrow\}$ be a derivation system for a constraint language CLP($\mathcal{X}$). *Then, for any program P and query Q of* CLP($\mathcal{X}$), *a correct answer for Q with respect to P is a computed answer and vice-versa.*

A derivation system satisfying proposition 2.1, however, is only of practical relevance if it can be justified that there exists an efficient proof procedure for it, which preserves the proposition. By a proof procedure we mean an algorithm which, for any correct answer, can produce a representation of it in finite time.

In Lloyd's presentation of SLD-resolution [38], he uses the notion of a computation rule to represent the overall behaviour of a proof procedure. A computation rule is defined as a deterministic strategy for selecting the next atom to be processed (out of the finite set comprising the current goal) and he shows that the completeness result is maintained under any such computation rule. This characterization is in some sense optimal: It explains how a large source of nondeterminism can be eliminated and the remaining nondeterminism in the derivation step is finite and cannot be reduced further in any obvious way. In the SLD case, the nondeterminism in the choice of program clause needs to be preserved.

When defining a notion of computation rule in the context of constraint solving, it should analogously govern a portion of the process as large as possible where alternative choices can be eliminated. In the following definition, we set as the domain of a computation rule "the choice of derivation step and the literals to which it applies". Thus, in order to achieve an optimal behaviour, the nondeterminism within each rule of the derivation system should be restricted to what really is essential for completeness.[5]

Definition 2.4. Given a derivation system $\{\rightsquigarrow_i\}_{i\in I}$, a *computation rule* is a function from finite sets of literals to some $i \in I$ and selected literal(s) to which the rule $\rightsquigarrow_i$ applies. Given a computation rule R, a *R-derivation* is a derivation in which each step $S_n \rightsquigarrow S_{n+1}$ is consistent with R. A computed answer produced in an R-derivation is called an *R-computed answer*.

Condition (ii) in the previous definition anticipates a computation rule which is *fast-solving:*

The resolution rule (Res) cannot be applied if another rule is applicable.

When a fast-solving computation rule is used, condition (ii) implies that $\rightsquigarrow^*$ serves as an effective constraint satisfaction test which in addition may simplify the constraint set. It is easy to show that proposition 2.1 still holds when derivation is restricted by a fast-solving computation rule. We can illustrate the relevance of a fast-solving computation rule by the query 'p' to the program consisting of the clauses 'p $\leftarrow$ a=b $\wedge$ p' and 'p'. A fast-solving computation rule yields one failed and one succesful derivation whereas a "slow-solving" rule in addition would produce one infinite derivation and infinitely many failed ones.

For analogous reasons, it may be relevant to require a computation rule which is *fast-unifying:*

The rules (Unif), (Dif1), and (Dif2) takes precedence over all other rules.

Finally, we have to remark that the conditions (i) and (ii) in the definition above turns out to be too naive in many cases. For the derivation system we present in section 2.4, condition (i) needs to be refined by a notion of equivalence of satisfier sets and an invariant must be imposed on the states in order to verify (ii).

[5] As in (Lloyd, 1987) the process of renaming variables, also called "standardizing apart", is tacitly assumed to be a deterministic operation. Choosing just one out of the infinitely many renamings is sufficient! Analogously for the choice of a most general unifier in the unification step.

2.2. The object and metalanguage

The object language for demo is called $\mathcal{HCL}$ and consists of untyped, positive Horn clauses with equality and inequality constraints allowed in the body of clauses. The precise syntax and semantics is given by considering $\mathcal{HCL}$ as a constraint logic language (as defined above) with only one type and no additional constraints.

The metalanguage in which demo is written is a constraint logic language CLP($\mathcal{HCL}$) with function symbols that reflect the syntax of $\mathcal{HCL}$ and constraints that make it possible to express its proof relation.

CLP($\mathcal{HCL}$) has the following types:

> *program*, *clause*, *formula*, *atom*, *constraint*,*term*, *substitution*,
> and *substitution-pair*.

For each symbol f of $\mathcal{HCL}$, CLP($\mathcal{HCL}$) includes a function symbol $'f$ of arity and rank corresponding to the syntax of $\mathcal{HCL}$, e.g.,

> $'\leftarrow$: *atom* $*$ *formula* $\rightarrow$ *clause*,

and for each $\mathcal{HCL}$ variable, say X, a constant $'\mathrm{X}: \rightarrow$ *term*, etc. For any phrase P of $\mathcal{HCL}$, the notation $\lceil P \rceil$ refers to the ground term that arise when each symbol f occurring in P is replaced by $'f$ and we call $\lceil P \rceil$ a *name* for P. Additionally, the metalanguage includes the following function symbols to represent programs and substitutions,

> empty-program: $\rightarrow$ *program*,
> program-cons: *clause* $*$ *program* $\rightarrow$ *program*,
> empty-substitution: $\rightarrow$ *substitution*,
> program-cons: *substitution-pair* $*$ *substitution* $\rightarrow$ *substitution*,
> pair: *term* $*$ *term* $\rightarrow$ *substitution-pair*.

Prolog's list-notation will be used for terms of type *program* as well as *substitution*; the 'pair' function is written $(-,-)$. We extend the $\lceil \cdots \rceil$ notation to programs and substitutions as follows,

> $\lceil \{C_1, \ldots, C_n\} \rceil = [\lceil C_1 \rceil, \ldots, \lceil C_n \rceil]$,
> — assuming no duplicates among the clauses $C_1, \ldots, C_n$.
> $\lceil \{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\} \rceil = [(\lceil X_1 \rceil, \lceil T_1 \rceil), \ldots, (\lceil X_n \rceil, \lceil T_n \rceil)]$,
> — assuming no duplicates among the variables $X_1, \ldots, X_n$.

Notice that the name of an object program (or substitution) is not unique, so the metalevel constraints "$=$" and "$\neq$" do not correspond to the relations between object programs (substitutions) usually written with the same symbols. For all other syntactic phrases of $\mathcal{HCL}$, the name in CLP($\mathcal{HCL}$) is unique and compositional.

The reverse brackets $\lfloor \cdots \rfloor$ are used inside $\lceil \cdots \rceil$ to indicate the presence of a metavariable. If, for example, Z is a metavariable of type *atom*, we have

$$\lceil \lfloor \mathrm{Z} \rfloor \leftarrow \mathrm{q(f(X,b))} \rceil \quad = \quad \mathrm{Z}\ '\leftarrow\ '\uparrow\ '\mathrm{q}('\mathrm{f}('\mathrm{X},'\mathrm{b})).$$

For simplicity, we have not included the naming brackets' notation in the formal definition of the language, instead we consider it as syntactic sugar. But we want to emphasize, that any implemented system must support such a notation in some way. Otherwise, programming about the ground representation is really impractical.

We have chosen the naming relation as simple as possible in order to simplify the presentation in this paper. In the implemented system, section 4.1, we use a more

detailed naming relation in which names of function and predicate symbols, arities, and argument lists are made explicit so that the name of, say, p(X,a) is the following huge term.

atom(predicate(p, 2),
termlistcons(variable(X),termlistcons(constant(a),emptytermlist))).

For practical metaprogramming, we find such a naming relation relevant because it allows to parameterize over object phrases in a more abstract way, e.g., by binding an item to name a constant or a function symbol of arity three without specifying which symbol.

We believe, however, that our choice of naming relation in the present paper is representative and suited to display the general principles behind the sort of metalanguages and metainterpreters we are interested in. In [12] we use a metalanguage based on the more detailed naming relation illustrated above and give a derivation system for it, which is much more complicated than what we can show here.

For each type $\tau \in \{\mathit{clause}, \mathit{formula}, \mathit{atom}, \mathit{constraint}, \mathit{term}\}$, CLP($\mathcal{HCL}$) has a constraint symbol

$\text{instance}_\tau\colon\ \tau * \tau * \mathit{substitution}$.

The type subscript will be left out when obvious from the context or when a distinction is unnecessary. Additionally, we have the following constraint symbols.

no-duplicates: *program*,
member: $\mathit{clause} * \mathit{program}$
not-member: $\mathit{clause} * \mathit{program}$

Satisfaction is defined by exactly the following constraints recognized as satisfiable:

- any constraint instance($\lceil P_1 \rceil, \lceil P_2 \rceil, \lceil \sigma \rceil$) where P_1, P_2 are phrases of $\mathcal{HCL}$, σ a $\mathcal{HCL}$ substitution with $P_1\sigma = P_2$,[6]
- any constraint of the form member($c, [\ldots, c, \ldots]$),
- any constraint of the form not-member($c, [c_1, \ldots, c_n]$), $n \geq 0$ where c is different from any of $c_1, \ldots, c_n$, and
- any constraint of the form no-duplicates($[c_1, \ldots, c_n]$), $n \geq 0$ where all $c_1, \ldots, c_n$ are different.

Member constraints are used in the usual way for selecting clauses out of the object program. Instance constraints provide a way to express object level unification at the metalevel, which we will make clear in proposition 2.2 and example 2.1 below.

It is obvious that the member constraint could have been specified as an ordinary predicate, but in order to suppress the generation of different representations of the same object program, we need to have a detailed procedural control, which is only possible by explicit derivation rules. The no-duplicates constraints are used to ensure that terms of type *program* really are names of programs; not-member and $\neq$ constraints are used here as auxiliaries. We could also have provided a collection of similar constraints concerned with terms of type *substitution*, but it will turn out that the derivation system and the metaprograms of interest imply the relevant properties.

[6] Notice that σ by convention must an idempotent substitution. For the usage of instance constraints in implementing the demo predicate, satisfaction of instance constraints could equally well have been defined in terms of general or ground object level substitutions.

The following property, that follows from the definition of the constraints, indicates how object level unification can be simulated by instance constraints.

Proposition 2.2. For arbitrary metalevel terms t_1, t_2, t, s_1, s_2, and metalevel substitution θ, the following properties are equivalent.

- $\{\text{instance}(t_1, t, s_1), \text{instance}(t_2, t, s_2)\}$ *is satisfiable with satisfier* θ.
- *There exists phrases* P_1, P_2, P, *and substitution* σ *of* $\mathcal{HCL}$ *such that* $t_1\theta = \lceil P_1 \rceil$, $t_2\theta = \lceil P_2 \rceil$ *and* $P_1\sigma = P_2'\sigma = P$, *where* P_2' *is a copy of* P_2 *whose variables have been renamed apart from those of* P_1.

The inherent renaming of variables in the proposition is very convenient as a our primary goal is to simulate a the proof relation for $\mathcal{HCL}$ given by definition 1. Typically P_1 is a part of the object level query and P_2 the head of an object clause.

The proposition imply a kind of reflection of object language variables to metavariables that we want to make clear by means of an example.

Example 2.1. The constraint

$$\text{instance}(\lceil \text{p(X)} \rceil, \text{Z}, _)$$

is equivalent to the following equation where Z_X is a new variable.

$$\text{Z} = \lceil \text{p}(\lfloor \text{Z}_\text{X} \rfloor) \rceil.$$

An instance constraint can thus be implemented by a translation from the ground representation into terms with 'live' metavariables standing for object variables, quite similar to the non-ground representation used in the classical Vanilla interpreter (see, e.g., [25]).

So referring to proposition 2.2, we can map object level unification to the metalevel in the following way: To unify two object terms given by their ground names, replace variable names consistently with new metavariables and let the metalevel unification do the work. — It will appear that the derivation system $\mathcal{DS}$ to be introduced performs a recursive decomposition of instance constraints which yields exactly this translation, the substitution argument is used to ensure consistency in the variable's replacement.

Consider, for example, the object level condition

$$\text{p(X,a)} = \text{p(b,Y)}.$$

It is equivalent to the metalevel equation,

$$\lceil \text{p}(\lfloor \text{Z}_\text{X} \rfloor, \text{a}) \rceil = \lceil \text{p}(\text{b}, \lfloor \text{Z}_\text{Y} \rfloor) \rceil.$$

2.3. The metainterpreter

Using CLP($\mathcal{HCL}$), we can now write a metainterpreter which simulates $\mathcal{HCL}$'s proof relation. The program, which we will refer to as DEMO, defines the following two predicates.

demo: *program* $*$ *formula*,
demo_1: *program* $*$ *formula*.

We remind that the $\uparrow$ symbol appearing in clause ($d_1 2$) is the inclusion operator of atoms into formulas that is left implicit in most other cases.

(d) demo(P, Q) $\leftarrow$
no-duplicates(P) $\wedge$
instance(Q, Q_1, _) $\wedge$
demo$_1$(P, Q_1).

($d_1 1$) demo$_1$(P, $\lceil$*true*$\rceil$) $\leftarrow$ *true*.

($d_1 2$) demo$_1$(P, $\lceil\uparrow \lfloor$A$\rfloor\rceil$) $\leftarrow$
member(C, P) $\wedge$
instance(C, $\lceil \lfloor$A$\rfloor \leftarrow \lfloor$B$\rfloor \rceil$, _) $\wedge$
demo$_1$(P, B).

($d_1 3$) demo$_1$(P, $\lceil\lfloor T_1\rfloor{=}\lfloor T_2\rfloor\rceil$) $\leftarrow$ $T_1{=}T_2$.

($d_1 4$) demo$_1$(P, $\lceil\lfloor T_1\rfloor{\neq}\lfloor T_2\rfloor\rceil$) $\leftarrow$ $T_1{\neq}T_2$.

($d_1 5$) demo$_1$(P, $\lceil\lfloor$A$\rfloor \wedge \lfloor$B$\rfloor \rceil$) $\leftarrow$
demo$_1$(P, A) $\wedge$
demo$_1$(P, B).

The purpose of the 'no-duplicates' constraint is to impose our convention of programs being sets and not lists of clauses, a property which cannot be expressed with the sort of types normally used in logic programming. This principle, together with a careful implementation, section 2.4 below, of the 'member' constraint prevents the generation of alternative presentations of the same program due to permutation and duplication of clauses.

The soundness and completeness of the DEMO program can be expressed as follows.

Proposition 2.3. Let p and q be terms of type program *and* formula*, and ρ a substitution, all of the language* CLP($\mathcal{HCL}$)*. Then the following two statements are equivalent.*

- *ρ is a correct answer for* demo(p, q) *with respect to the* DEMO *program.*
- *There exist program P, query Q, and substitution σ of $\mathcal{HCL}$, where σ is a correct answer for Q with respect to P and $p\rho = \lceil P\rceil$, $q\rho = \lceil Q\rceil$.*

PROOF. Assume ρ is a correct answer for demo(p, q), i.e.,

(1) DEMO $\vdash_{\text{CLP}(\mathcal{HCL})}$ demo$(p, q)\rho$

By definition of $\vdash$ and referring to clause (d) of the DEMO program, we see that (1) holds if and only if there exist query Q_1 and substitution σ of $\mathcal{HCL}$ such that

(2) DEMO $\vdash_{\text{CLP}(\mathcal{HCL})}$ no-duplicates$(p\rho)\wedge$
instance$(q\rho, \lceil Q_1\rceil, \lceil\sigma\rceil)\wedge$
demo$_1(p\rho, \lceil Q_1\rceil)\rho$.

By the definition of the constraints, (2) holds if and only if $p\rho = \lceil P\rceil$, $q\rho = \lceil Q\rceil$ for some program P and query Q of $\mathcal{HCL}$ with $Q\sigma = Q_1$ and

(3) $P \vdash_{\mathcal{HCL}} Q_1$ (or equiv. $P \vdash_{\mathcal{HCL}} Q\sigma$).

In (2), we can especially choose Q_1 and σ such that Q_1 is ground without affecting the satisfiability of the instance constraints. Under this assumption, we can show

(3) equivalent with

(4) $P \vdash_{\mathcal{HCL}} Q_1$.

This follows by induction over the number of recursive applications of the definition of $\vdash$ needed to verify $P \vdash_{\mathcal{HCL}} Q_1$. In the induction step we use the fact that each clause for demo_1 is equivalent with an instance of a case in the definition of $\vdash$. □

It should be emphasized that the so-called completeness of demo needs to be complemented by an appropriate derivation system in order to be of a more than mathematical interest. The following example investigates the potential expressivity in queries to the demo predicate.

Example 2.2. The set of answers to the query,

demo(X, Y)

characterizes the proof relation $\vdash_{\mathcal{HCL}}$.

The set of answers to the query

demo($\lceil P \rceil$, X)

characterizes the set of formulas which are logical consequences of the $\mathcal{HCL}$ program P.

Assuming a predicate abducible: $\rightarrow$ *clause* programmed in CLP($\mathcal{HCL}$), the set of answers to the query

abducible(X) $\wedge$ demo($\lceil$ [$\lfloor$X$\rfloor$ | P] $\rceil$), $\lceil Obs \rceil$)

characterizes the set of all $\mathcal{HCL}$ clauses whose name satisfies the abducible(–) condition and which together with the clauses of P can explain the 'observations' Obs.

It may seem a little unsatisfactory that the demo predicate does not provide any information about which object level substitution actually made a query succeed. However, we can get the same information using metavariables as part of the query argument, cf. the "reflection principle" indicated in example 2.1.

Example 2.3. Assume an object program P and let O be the set of ground object terms determined by the answer substitutions for the object query p(X). Let in a similar way M be the set of terms given for the metavariable Z_X by the metalevel query

demo($\lceil P \rceil$,$\lceil$p($\lfloor \text{Z}_\text{X} \rfloor$)$\rceil$)

with respect to the DEMO program. Then M consists of names for the terms of O together with of (all!) names for non-ground object terms, each of which is more general than a term in O.

However, a derivation system needs not invent new variable names. Instead it can provide more abstract answers in the shape of answer constraints. An answer constraint such as

instance(Z_X, $\lceil$f(a,b)$\rceil$, [('Y,'c)| _])

captures the names of a large collection of names of object terms, e.g., $\lceil$f(a,b)$\rceil$, $\lceil$f(A,b)$\rceil$, $\lceil$f(a,B)$\rceil$, etc., but not $\lceil$f(Y,b)$\rceil$.

2.4. A derivation system for solving CLP($\mathcal{HCL}$) *constraints*

In the following, we present and explain the rules of a derivation system for solving the constraints of CLP($\mathcal{HCL}$). The system, called $\mathcal{DS}$ consists of the rules below together with the basic rules (Res), (Unif), (True), and (Dif1–2) introduced in section 2.1. The soundness and completeness of $\mathcal{DS}$ is shown in section 3.

We start with the rules concerned with instance constraints. The rule (I) expresses that a given variable can have one and only one instance under a given substitution. It applies also following a unification of two variables $v = v'$.

(I) $C \cup \{\text{instance}(v, t, s), \text{instance}(v, t', s)\} \rightsquigarrow C \cup \{t = t', \text{instance}(v, t', s)\}$
— when v is variable.

The following two rules (It1–2) reduce instance constraints that express bindings to object variables; the first one applies when a binding has been recorded already for the given variable, the second one installs an initial binding in the substitution.

(It1) $C \cup \{\text{instance}_{term}('x, t, s)\} \rightsquigarrow C \cup \{t = t'\}$
— when $'x$ is the name of an $\mathcal{HCL}$ variable and $s = [\cdots('x, t')\cdots]$

(It2) $C \cup \{\text{instance}_{term}('x, t, s)\} \rightsquigarrow C \cup \{w = [('x, t)|w']\}$
— when $'x$ is the name of an $\mathcal{HCL}$ variable, (It1) does not apply, and $s = [\cdots|w]$; w' is a new variable.

Notice that a fast-unifying computation rule is relevant for (It2) in order to avoid different and incompatible expansions of the substitution tail w. The representation of an object level substitution is not unique, and it appears that the rules (It1–2) will produce exactly one representation of a given substitution, the one chosen depends on the order in which the rules are applied.

Instance constraints with names of structured object language terms in the first argument are reduced as follows.

(It3) $C \cup \{\text{instance}_{term}('f(t_1, \ldots, t_n), t', s)\}$
$\rightsquigarrow C \cup \{t' = 'f(v_1, \ldots, v_n),$
$\text{instance}_{term}(t_1, v_1, s), \ldots, \text{instance}_{term}(t_n, v_n, s)\}$
— when $'f$ is the name of a function symbol of $\mathcal{HCL}$, $n \geq 0$; $v_1, \ldots, v_n$ are new variables.

The reduction of a term instance constraint is, thus, triggered by its first argument being non-variable. In case the first argument is a variable, but the second is bound to a structure, the rule (It3) does not apply. To see why, consider the constraint $\text{instance}_{term}(\text{X}, \lceil\text{f(a)}\rceil, s)$. There is no meaningful assignment to make for X because the constraint is satisfiable with different values of X that are not covered by subsumption, e.g., $\lceil\text{f(a)}\rceil$, $\lceil\text{f(A)}\rceil$, $\lceil\text{f(B)}\rceil$, $\lceil\text{A}\rceil$, or $\lceil\text{B}\rceil$, where A and B are object variables.

Rules for all other syntactic constructs in $\mathcal{HCL}$ of categories *clause*, *formula*, *atom*, and *constraint* are defined similarly to (It3) except that they are triggered also by the second argument being non-variable.

(Ic) $C \cup \{\text{instance}_{clause}(t_1, t_2, s)\}$
$\rightsquigarrow C \cup \{t_1 = (u_1 {}'\!\leftarrow u_2), t_2 = (v_1 {}'\!\leftarrow v_2),$
$\text{instance}_{atom}(u_1, v_1, s), \text{instance}_{formula}(u_2, v_2, s)\}$
— when t_1 or t_2 is of the form $(\cdots {}'\!\leftarrow \cdots)$; u_1, u_2, v_1, v_2 are new variables.

(If1) $C \cup \{\text{instance}_{formula}(t_1, t_2, s)\}$
$\rightsquigarrow C \cup \{t_1 = (u_1{}' \wedge u_2), t_2 = (v_1{}' \wedge v_2),$
$\text{instance}_{formula}(u_1, v_1, s), \text{instance}_{formula}(u_2, v_2, s)\}$
— when t_1 or t_2 is of the form $\cdots{}' \wedge \cdots$; u_1, u_2, v_1, v_2 are new variables.

(If2) $C \cup \{\text{instance}_{formula}(t_1, t_2, s)\}$
$\rightsquigarrow C \cup \{t_1 = {}'\mathit{true}, t_2 = {}'\mathit{true}\}$
— when t_1 or t_2 is of the form $'\mathit{true}$.

(If3) $C \cup \{\text{instance}_{formula}(t_1, t_2, s)\}$
$\rightsquigarrow C \cup \{t_1 = {}'\uparrow u, t_2 = {}'\uparrow v, \text{instance}_{\tau}(u, v, s)\}$
— when t_1 or t_2 is of the form $'\uparrow \cdots$; u, v are new variables;
τ refers to type *atom* or *constraint*.

(Iac) $C \cup \{\text{instance}_{\tau}(t_1, t_2, s)\}$
$\rightsquigarrow C \cup \{t_1 = {}'p(u_1, \ldots, u_n), t_2 = {}'p(v_1, \ldots, v_n),$
$\text{instance}_{term}(u_1, v_1, s), \ldots, \text{instance}_{term}(u_n, v_n, s)\}$
— when t_1 or t_2 is of the form $'p(\cdots)$ for some $\mathcal{HCL}$ predicate p;
$u_1, \ldots, u_n, v_1, \ldots, v_n$ are new variables;
τ refers to type *atom* or *constraint*.

In rule (Iac) for type *constraint*, $'p$ may stand for one of the symbols $'=$ or $'\neq$.

Member constraints are reduced by the following rules.

(M1) $C \cup \{\text{member}(c, v)\} \rightsquigarrow C \cup \{v = [c|v']\}$
— when v is a variable; v' is a new variable.

(M2) $C \cup \{\text{member}(c, [c'|p])\} \rightsquigarrow C \cup \{m\}$
— where m is either $c = c'$ or $\text{member}(c, p)$.

(M3) $C \cup \{\text{member}(c, [\,])\} \rightsquigarrow$ FAILURE.

Rule (M1) is the only rule that can apply when a new clause is added to an uninstantiated program or program tail v. The nondeterminism in (M2) provides an iteration through the list of positions for clauses indicated by the list structure in a term of type *program*. To see the overall behaviour of (M1–3), consider the constraint set $\{\text{member}(c_1, v), \text{member}(c_2, v)\}$, assuming c_1, c_2 being two distinguishable terms and v a variable. With a given fast-unifing computation rule, exactly one solution will be found, e.g., $v = [c_1, c_2|_]$ if the computation rule choses $\text{member}(c_1, v)$ first. A fast-unifing computation rule will also prevent applications of (M1) leading to failure due to incompatible assignments to v.

The following transition rules define the behaviour of the no-duplicates constraints and the auxiliary not-member constraints.

(ND1) $C \cup \{\text{no-duplicates}([c|p])\} \rightsquigarrow C \cup \{\text{not-member}(c, p), \text{no-duplicates}(p)\}$.

(ND2) $C \cup \{\text{no-duplicates}([\,])\} \rightsquigarrow C$.

(NM1) $C \cup \{\text{not-member}(c, [c'|p])\} \rightsquigarrow C \cup \{c \neq c', \text{not-member}(c, p)\}$.

(NM2) $C \cup \{\text{not-member}(c, [\,])\} \rightsquigarrow C$.

When no-duplicates is called with an uninstantiated variable, it delays because none of (ND1–2) can apply. We notice that the time complexity is quadratic in the size

of the program.

3. CORRECTNESS: SOUNDNESS AND COMPLETENESS OF DEMO

In section 3.1 we characterize those states that can appear when $\mathcal{DS}$ and the DEMO program is used for executing queries to the demo predicate. We identify important invariants that are necessary in order prove soundness and consistency. The central result in section 3.2 is a proposition stating that the set of satisfiers (qua an equivalence relation) is preserved in the possible derivations from an initial state. Section 3.3 shows the central termination result and formulates the concluding soundness and completeness theorem.

3.1. States and their properties

Satisfiability of instance constraints in general is closely related to the undecidable semiunification problem; we discuss this in appendix A. To get around this problem, we restrict our correctness considerations to the following class of derivations and states.

Definition 3.1. A *demo-derivation* as a derivation in $\mathcal{DS}$ using a fast-solving and fast-unifying computation rule, with resolution made with clauses of the DEMO program, and starting from an initial state of the form $\langle \mathrm{demo}(p, q), \emptyset \rangle$. Any state occurring in a demo-derivation is called a *demo-state*.

The following properties of substititution well-formedness and safeness will serve as invariants, that are essential for correctness proofs.

Definition 3.2. A set of CLP($\mathcal{HCL}$) constraints C (or a state $\langle C, \alpha \rangle$) is *substitution well-formed* if the following conditions hold.

- Any instance constraint in C is of the form
  $$\mathrm{instance}(t, t', [('x_1, t_1), \ldots, ('x_n, t_n)|w]),$$
  where each $'x_i$ is the name of a distinct $\mathcal{HCL}$ variable; and no name of an $\mathcal{HCL}$ variable occurs in $t', t_1, \ldots, t_n$ and the tail w is a variable.
- If a variable w occurs as the tail of two substitution arguments s_1 and s_2, then s_1 and s_2 are identical; the only other way w can occur in C is in at most one equation of the form $w = [('x, t)|w']$ where w' is a variable that do not occcur elsewhere and $'x$ does not occur as part of a pair $('x, t)$ in $s_1 = s_2$.
- Terms of type *substitution* or *substitution-pair* do not occur in other ways in C.

Proposition 3.1. Any demo-state is substitution well-formed.

PROOF. Consider an arbitrary demo derivation $S_0 \rightsquigarrow^* S_n$ starting from a state $S_0 = \langle \mathrm{demo}(p, q), \emptyset \rangle$.

We use induction over n, observing that S_0 is substitution well-formed as there are no terms of type *substitution* or *substitution-pair* in it.

Whether a unification step violates the proposition depends on the equations set up by the derivation step preceeding it, so when considering the other possible steps, we check also the indirect effect given by the equations they may create.

A (Res) step using a clause of the DEMO program does not violate the property as the only terms of type *substitution* that are involved are new variables w appearing as substitution argument of a single instance constraint and nowhere else; no terms of *substitution-pair* are involved. By inspection, we see that no other rule of $\mathcal{DS}$ violates the property. □

Intuitively, substitution well-formedness means that each such substitution tail variable w serves as a fill-pointer to a unique substitution and the equations mentioned in the second condition appears temporarily when a new binding is added. The relevance of the notion is captured by the following proposition, which we state without proof.

Proposition 3.2. Let C be a substitution well-formed set of constraints. Then there exists a ground metalevel substitution ρ such that any term of type substitution appearing in $C\rho$ is the name of some object language substitution.

Furthermore, ρ can be chosen such that for any instance constraint in C, instance(t, t', s), $t'\rho$ is the name of a ground object term, $s\rho$ the name of a ground, object level substitution.

A metavariable in one of the arguments to demo will stand for the ground representation of some unknown phrase of the object language $\mathcal{HCL}$ (such variables are called *external* below). Looking at metavariables in second arguments to instance constraints (called *internal* below), their role is intuitively a bit different. Of course, they can only be bound to names of object language phrases but, as indicated by proposition 2.2 and example 2.1, they serve as "live" placeholders for object variables, effectively providing a non-ground representation. The following notion of safeness specifies that no confusion is possible between the two categories of metavariables and, as we will see later, this removes the hard recursion that may imply undecidability.

Definition 3.3. A substitution well-formed set of CLP($\mathcal{HCL}$) constraints C (or a state $\langle C, \alpha\rangle$) is *safe* if its variables of type different from *substitution* can be separated into two disjoint sets of *external* and *internal* variables such that the following conditions hold.

- For any instance constraint in C

  $$\text{instance}(t, t', [('x_1, t_1), \ldots, ('x_n, t_n)|w]),$$

  the variables occurring in t are external, those in $t', t_1, \ldots, t_n$ internal.
- The arguments to any constraint in C with constraint symbol member, not-member, or no-duplicates contain no internal variables.
- No equation or inequality in C between terms of type different from *substitution* contains both an internal and an external variable, or both an internal variable and the name of an $\mathcal{HCL}$ variabel.

Proposition 3.3. Any demo-state is safe.

PROOF. Consider an arbitrary demo derivation $S_0 \rightsquigarrow^* S_n$ starting from a state $S_0 = \langle \text{demo}(p, q), \emptyset\rangle$.

To show that safeness holds for any demo-state, we extend the characterization of internal and external variables as follows,

- for any call demo(t_1, t_2) in S, any variable in t_1 and t_2 is external,
- for any call demo$_1(t, t')$ in S, any variable in t is external, any variable in t' is internal.

Again, we can use induction over the n, observing that the proposition obviously holds for S_0. By inspection of the DEMO program and $\mathcal{DS}$ it appears that no rule, directly or by means of an equation, can mix up the two classes of variables. □

In the following we define what is understood by a constraint-normalized state and prove that such states are satisfiable. In section 3.3, below, we will show that $\mathcal{DS}$ with a fast-solving computation rule eventually will reach a constraint-normalized state before a possible next resolution step.

Definition 3.4. A safe constraint set C (or safe state $\langle C \cup A, \alpha\rangle$, with C consisting of constraints and A of atoms) is *constraint-normalized* whenever the following conditions hold.

- There are no equations or member constraints in C.
- Any instance constraint is of the form instance$_\tau(v, t, s)$ where v is a variable, and if τ is different from *term*, t is also a variable.
- If, for given v and s that instance$_\tau(v, t, s) \in C$, there is no other constraint instance$_\tau(v, t', s) \in C$.
- For any not-member(c, p) or no-duplicates$(p) \in C$, p is a variable.
- For any $t_1 \neq t_2 \in C$, t_1 and t_2 are different but not distinguisable.

Proposition 3.4. For a constraint-normalized state $\langle C \cup A, \alpha\rangle$, with C consisting of constraints, A of atoms, the state $\langle C, \alpha\rangle$ is satisfiable.

PROOF. Consider a constraint-normalized state $\langle C \cup A, \alpha\rangle$ as above. We show satisfiability by constructing a satisfier ρ for $\langle C, \alpha\rangle$ as follows.

Any inequality in C of the form $t_1 \neq t_2$ where t_1 and t_2 are different; it will be satisfied under any substituition ρ which is such that for any variable v in t_1 or t_2, $v\sigma$ contains a subterm that does not occur elsewhere in C or in $v'\sigma$ for any other variable v'. The decisions for ρ in the following will satisfy this requirement. Notice that the substitution well-formedness condition implies that terms of type *substitution* do not occur in an inequality in C. By inspection of the DEMO program and $\mathcal{DS}$, the same thing can be shown for terms of type *program*.

In any constraint in C of the form not-member(c, p) or no-duplicates(p), p is a variable, and letting $p\rho = []$ for any such variable p, will satisfy these constraints. Such variables p cannot occur in other ways in C so the satisfiability of any other constraint is not affected by this decision.

For each instance$_{term}(v, t, s)$, v is a variable. Let here $v\rho = \lceil X_v \rceil$ where X_v is a unique $\mathcal{HCL}$ variable whose name does not occur elsewhere. To any variable x occurring in t, let $x\rho = \lceil T_x \rceil$ where T_x is a ground term of $\mathcal{HCL}$ whose name does not occur elsewhere. The safeness condition ensures that there is no conflict in these decisions (i.e., no variable can be referred to both as a "v" and as an "x"). The mentioned instance constraints will be satisfied under any further specified ρ for which $s\rho$ is the name of a substitution such that $s\rho = [\cdots(\lceil T_x \rceil, t\rho)\cdots]$.

For each $\text{instance}_{atom}(v, v', s)$, v and v' are variables. Let here $v\rho = \lceil p(X_v) \rceil$ where p is a predicate symbol and X_v a unique $\mathcal{HCL}$ variable whose name do not occur in elsewhere, and let $v'\rho = \lceil p(T_{v'}) \rceil$ where $T_{v'}$ is a ground term of $\mathcal{HCL}$ whose name does not occur elsewhere. The mentioned instance constraints will be satisfied under any further specified ρ for which $s\rho$ is the name of a substitution such that $s\rho = [\cdots([X_v], [T_{v'}])\cdots]$.

Instance constraints concerned with constraints, formulas, and clauses are treated analogously.

The requirements made above concerned with the effect of ρ on substitution arguments can be met, because each of the mentioned object variables X_v can occur only once in relation to a given substitution. This follows from the part of the definition of constraint-normalized state that says that for any $\text{instance}_{\tau}(v, t, s)$, there is no other $\text{instance}_{\tau}(v, t', s)$.

Finally, consider any binding $x \mapsto t$ in α. Due to the idempotent property for states, x does not appear in C so there have not been made any decisions for $x\rho$ above. Thus we can let $x\rho = t\rho$, possibly adding some arbitrary values for any variables in t not mentioned already.

This finishes the construction of a satisfier for S. □

Finally, we have an invariant property, that excludes a subtle recurrence problem that may cause $\mathcal{DS}$ to loop when applied to constraint sets outside the domain of demo-states. In order to motivate proposition 3.5 we give an example showing the phenomenon, which it excludes.

Example 3.1. Consider a state which includes the following constraints.

$$\text{instance}_{formula}('true' \wedge \text{Z}, \text{Y}, S), \text{instance}_{formula}(\text{Z}, \text{Y}, S')$$

Transition rule (If1) can be applied to the first constraint leading to an instantiation of Y, and following steps by (Unif) and (If2), we have the following constraints.

$$\text{instance}_{formula}(\text{Z}, \text{Y}_1, S), \text{instance}_{formula}(\text{Z}, 'true' \wedge \text{Y}_1, S')$$

Now (If1) applies to the second constraint leading to an instantiation of Z, and following steps by (Unif) and (If2), we get the following and a loop is established.

$$\text{instance}_{formula}('true' \wedge \text{Z}_1, \text{Y}_1, S), \text{instance}_{formula}(\text{Z}_1, \text{Y}_1, S')$$

The problem arise because terms of type *formula* can contain proper subterms also of type *formula*; there are no similar problems for the types *atom*, *constraint* and *clause*. For instance_{term} constraints, the problem does not exist because the corresponding transition rule (It3) only propagates structure from the first to the second argument, but not the other way round.

Proposition 3.5. An internal variable of type formula in a demo-state S can occur at most once in some instance constraint in S.

The proposition can be proved by a straightforward but lengthy induction proof that requires a detailed inspection of all transitions rules and clauses of the DEMO program. The proof in itself does not give any interesting new insight, so we have decided to leave it out.

3.2. Preservation of satisfiers

As we have described earlier, $\mathcal{DS}$ suppresses the generation of different representations of the same object programs, one random representation is created, determined by the order in which the constraints are processed. The same holds for representations of object substitutions. This means that some derivations steps will reduce the set of satisfiers — measured at the metalevel — but without changing the possible choices at the object level. In order to characterize soundness, we introduce the following notion of equivalence for sets of satisfier.

Definition 3.5. Two ground metalevel substitutions σ_1 and σ_2 are *equivalent* if, for any variable v that $v\sigma_1$ and $v\sigma_2$ either

- are names for the same object program,
- are names for the same object substitutions, or
- are identical.

Two sets of substitutions Σ_1 and Σ_2 are *equivalent* whenever each member of Σ_1 has an equivalent substitution in Σ_2 and vice versa.

The following proposition implies the soundness of $\mathcal{DS}$ in the sense that no derivation step can introduce satisfiers not entailed by the original query and, thus, that any computed answer also is a correct answer. In addition, it points forward to the completeness result by indicating that no correct answer is "lost" in the derivation process — but proposition 3.6 has to be complemented by the termination result to be given in section 3.3 in order to show completeness.

Proposition 3.6 (Preservation of satisfiers for $\mathcal{DS}$). For any demo-state S, let $\{S_1, \ldots, S_n\}$ be the set of all states with $S \rightsquigarrow S_i$ by a step in a demo-derivation. Then the set of substitutions $\bigcup_{i=1,\ldots,n}[\![S_i]\!]$, each restricted to the variables of S, is equivalent with $[\![S]\!]$.

Proof. Whenever $S \rightsquigarrow S_i$ by one of the rules (Dif1–2), (I), (It1), (It3), (Ic), (If1–3), (Iac), (M2–3), (ND1–2), and (NM1–2), the action that takes places is that a constraint is replaced by other constraints (or failure), which are obviously equivalent by definition of these constraint. In these cases, we have that $[\![S_i]\!]$, each restricted to the variables of S, coincides with $[\![S]\!]$. Similar arguments go for (Unif) and (True).

The rule (M1) replaces a constraint member(c, p), p a variable, by an equation $p = [c|p']$, where p' is a new variable, and this obviously reduces the possible set of satisfiers. In order to show the equivalence between $[\![S]\!]$ and $[\![S_i]\!]$ (suitably restricted), consider firstly an arbitrary satisfier σ for S. It must hold that $p\sigma = [c_1, \ldots, c_n]$ with $c\sigma = c_i$ for some i. By definition, σ will be equivalent with another substitution σ' where $p\sigma' = [c\sigma', c'_1 \ldots, c'_{n-1}]$. We have to verify that σ' is a satisfier for S and that it has an extension σ'' which is a satisfier for S_i. To see this, we have to consider the possible ways p can occur in constraints in S. Due to the fast-unifying computation rule, there are no equations in S involving p, so the possible occurrences are of the forms, no-duplicates$([\cdots|p])$, not-member$(\cdots, [\cdots|p])$, or member$(\cdots, [\cdots|p])$. By definition of these constraints, when they are satisfied under σ, they must also be satisfied under σ'. Letting $\sigma'' = \sigma' \cup \{p' \mapsto [c'_1 \ldots, c'_{n-1}]$, we achieve a satisfier for S_i. — The other way round, any satisfier for S_i is obviously

also a satisfier for S.

The argument is quite similar for (It2), which adds an object variable binding to the representation of an object substitution, in a way that reduces the possible satisfier sets.

Consider, now, for an atom A in S the set of possible states $\{S'_1, \ldots, S'_{k_A}\}$ that can be reached by a (Res) step from S involving A and some clause in the DEMO program. By definition of $\vdash$, it follows that the substitutions $\bigcup_{i=1,\ldots,k_A} [\![S'_i]\!]$, each restricted to the variables of S, coincides with $[\![S]\!]$. □

3.3. Termination in constraint solving

The use of a fast-solving computation rule implies that all rules concerned with constraint solving must finish before a next resolution step will be allowed. In this section, we show that these constraint solving sub-derivations alwyas terminate in a constraint-normalized state.

To begin with, we show that if termination is observed, it will do so with a constraint-normalized state.

Proposition 3.7. The only derivation steps possible in a demo-derivation from a constraint-normalized demo-state are by the rules (Res) and (True).

For any demo-state different from FAILURE *and which is not constraint normalized, there will be possible derivation steps by means of rules different from (Res) and (True).*

If a demo-derivation ends in a final state S which is not failed, then S is constraint-normalized.

PROOF. By inspection of $\mathcal{DS}$, it appears that the conditions that enable each rule (different from (Res) and (True)), implies a state which is not constraint-normalized; hence the first part of the proposition. With respect to the second part, it is easy to see that the possible ways a state can violate the definition of a normalized state imply that some rule different from (Dif) and (True) can apply.

The last part is a direct consequence of what already has been shown. □

Now, finally, comes the central termination result.

Proposition 3.8 (Termination in constraint solving). Consider an arbitrary demo-derivation and an arbitrary state S_n in it,

$$S_0 \leadsto \cdots \leadsto S_n \leadsto \cdots$$

Then there exists a $k \geq n$ such that S_k is constraint normalized or FAILURE *and no derivation step $S_i \leadsto S_{1+1}$, $i = n, \ldots, k-1$ is made by (Res).*

With the strong invariants on demo-states expressed by proposition 3.5 and the safeness proposition 3.3, the following termination proof becomes straightforward although a bit lengthy due to the number of different constraints and types.

PROOF. For any demo state S, we define the following weights, each being an integer number ≥ 0.

- n_1: The number of instance$_{clause}$ and 'member' constraints in S.
- n_2: The number of occurrences in second arguments to instance constraints in S of function symbols with a rank of the form $\cdots \rightarrow$ *formula*.

n_3: The number of occurrences of in first arguments to instance$_{formula}$ constraints in S of function symbols with a rank of the form $\cdots \to$ *formula*.

$n4$: The number $f + a + c + t$ where

- f is the number of instance$_{formula}$ constraints in S,
- a is the number of instance$_{atom}$ constraints in S,
- c is the number of instance$_{constraint}$ constraints in S,
- t is the number of occurrences of the truth constant *true* (of CLP($\mathcal{HCL}$)) in S.

n_5: The number of occurrences of function symbols in first arguments to instance$_{atom}$ and instance$_{term}$ constraints in S.

n_6: The number of instance$_{term}$ constraints in S,

n_7: The number of occurrences of function symbols in arguments to 'no-duplicates' constraints in S.

n_8: The number of occurrences of function symbols in arguments to 'member' and 'not-member' constraints in S.

n_9: The number of '$\neq$' constraints in S.

We use the tuple $\langle n_1, ..., n_9 \rangle$ to measure the complexity of S assuming a lexicographical ordering of such tuples defined as follows.

> $\langle n_1, ..., n_9 \rangle \ < \ \langle n'_1, ..., n'_9 \rangle$ if and only if,
> for some $k = 1, \ldots, 9$ that $n_1 = n'_1, \ldots, n_{k-1} = n'_{k-1}$ and $n_k < n'_k$.

We will show that each possible derivation step (with a transition rule different from (Res)) decreases this measurement. Referring to the fast-unifying computation rule, we consider only states without equations and instead show that the sequence of steps (R)–$(\text{Unif})_1$– $\cdots$ –$(\text{Unif})_n$, viewed as a whole, decreases the measurement, where (R) denotes any rule different from (Unif) and (Res) and $(\text{Unif})_1$– $\cdots$ –$(\text{Unif})_n$, $n \geq 0$ are the unification steps for the possible equations produced by (R). We can ignore those rules that lead to FAILURE and proceed as follows.

- The rule (True) decreases n_4 and leaves the other weights unchanged.
- The rule (Dif1) decreases n_9 and leaves the other weights unchanged.
- The rule (I) replaces constraints $\text{instance}(v, t, s), \text{instance}(v, t', s)$ by $t = t', \text{instance}(v, t', s)$. We consider the possible types of the arguments seperately.
  - For type *clause*, n_1 is decreased.
  - For type *formula*, n_1 is unaffected and proposition 3.5 gives that the unification for $t = t'$ cannot increase n_2. Safeness implies that n_3 is unaffected. One instance$_{formula}$ constraint is removed, so n_4 is decreased.
  - For types *atom* and *constraint*, $n_1, \ldots, n_3$ are unaffected, n_4 is decreased.
  - For type *term*, $n_1, \ldots, n_5$ is unaffected, n_6 is decreased.
- The rules (It1–3) all decreases n_5; $n_1, \ldots, n_4$ are unaffected.
- The rule (Ic) decreases n_1.
- For the rule (If1) we consider two cases.
  - t_2 is of the form ${t_{21}}' \wedge t_{22}$; here n_1 is unaffected, n_2 decreases.
  - t_2 is a variable and t_1 is of the form ${t_{11}}' \wedge t_{12}$; n_1, n_2 are unaffected, for n_2 we need to refer to proposition 3.5, and n_3 decreases.

- For the rules (If2–3), the arguments are analogously to those for (If1).
- For the rule (Iac), $n_1, \ldots, n_3$ are unaffected, n_4 decreases.
- The rule (M1) decreases n_1.
- The rule (M2) replaces member$(c, [c'|p])$ by either
  - an equation $c = c'$ in which case n_1 decreases, or
  - a simpler constraint member(c, p) in which case n_8 is decreased, all other weights are unaffected.
- The rules (ND1–2) leave $n_1, \ldots, n_6$ unchanged and decreases n_7.
- The rule1 (NM1–2) leave $n_1, \ldots, n_7$ unchanged and decreases n_8.

□

We summarize this termination result and propositions 2.3, 3.4, 3.6, and 3.7 in the following, which express the soundness and completeness of demo-derivations considered as an implementation of $\vdash_{\mathcal{HCL}}$.

Theorem 3.1. Let p and q be terms of type program and formula, and ρ a substitution, all of the language CLP($\mathcal{HCL}$). *Then the following two statements are equivalent.*

- *ρ is a computed answer for the query demo(p, q) with respect to the derivation system $\mathcal{DS}$ and the* DEMO *program using a fast-solving and fast-unifying computation rule.*
- *There exists program P, query Q, and substitution σ of $\mathcal{HCL}$, where σ is a correct answer for Q with respect to P and $p\rho = \lceil P \rceil$, $q\rho = \lceil Q \rceil$.*

From this, it follows that the set of all correct answer for demo(p, q) can be found by a procedure that combines the indicated computation rule with a breadth-first management of alternative resolution steps. — A more practical procedure is considered in section 4.1 below.

4. AUTOMATED REASONING WITH THE DEMO PREDICATE

In this section we describe how the metainterpreter developed in section 2 can be made into an efficient Prolog program, which in turn serves as the central tool in a methodology for automated reasoning.

We illustrate the use of the system by a number of examples that are indented to cover a wide range of different reasoning problems as well as showing how unusual and unexpected combinations of ideas can be realized in a surprisingly straightforward fashion. All examples are available in full detail together with the source code for the system at the electronic address given in section 1.

4.1. Outline of an implementation in Prolog

4.1.1. IMPLEMENTING CONSTRAINTS IN PROLOG An efficient implementation of CLP($\mathcal{HCL}$) has been achieved using mechanisms in Sicstus Prolog [50] intended for constraint solving. Constraints are represented as Prolog predicates that delay and reactivate themselves in a suitable way. This eliminates the need for an additional layer of overhead that would arise in a straightforward simulation of the derivation system.

The basic transition rules (Unif), (Res), (True), and (Dif1–2) are inherent in the underlying Prolog system, so with an implementation of the constraints of CLP($\mathcal{HCL}$) as indicated, programs written in CLP($\mathcal{HCL}$) can be executed directly by the Prolog.

Sicstus Prolog supports a notion of *attributed variables* [30, 31] which allows the programmer to set up, inspect and alter named attributes attached to each Prolog variable. In addition, the programmer may supply a hook predicate `verify_attributes`, which is called automatically whenever Prolog unifies a variable with a term (which may appear to be another variable).

We illustrate the principle by the type constraints we have incorporated to correct for the fact that Prolog is not a typed language. For each type τ of CLP($\mathcal{HCL}$), there is a constraint $\tau(t)$ satisfied for exactly all terms t of type τ. These constraints are represented as predicates `program_(...)`, `clause_(...)`, `atom_(...)`, etc., the underline character consistently used to distinguish constraints from a few Prolog built-in's. A delayed constraint, say `atom_(X)`, is attached as an attribute to `X` controlled as follows.

- Whenever `X` is unified with another variable `Y`, it is checked that `Y` does not carry a different type constraints. The type constraint is preserved for the variable `X=Y`.
- Whenever `X` is unified with a structural term, the constraint is reduced (if possible) into other type constraints according to the naming relation.

So if `X` is unified with $\lceil$p(a,$\lfloor$`Z`$\rfloor$)$\rceil$, the variable `X` together with `atom_(X)` vanishes and `term_(Z)` remains as an attribute on `Z`.

Instance constraints are implemented in a similar way except that now several constraints can be delayed on the same variable `X`, e.g., `term_instance_(X,T,S)` and `term_instance_(X,T1,S1)`. For `term_instance_` the attributes are controlled as follows.

- Whenever `X` is unified with another variable `Y`, the two attached constraint sets are compared in order to identify possible applications of transition rule (I), and a new list of delayed constraints is formed for the variable `X=Y`.
- Whenever `X` is unified with a structural term, the constraint is reduced (if possible) as specified by the transition rules (It1–3).

Instance constraints for other types are delayed in case both the first and second argument are variables. A delayed constraint, say `atom_instance_(X,Y,S)` is attached as an attribute to `X` as well as to `Y`, so that a unification to either of the variables can trigger an action.

The 'no-duplicates' and 'not-member' constraints are implemented straightforwardly in accordance with the transition rules (ND1–2, NM1–2), with a unconditional delay when an argument of type *program* are given as a variable. The 'member' constraint is implemented in the following traditional Prolog style.

```
member(C,P):- var(P), !, P=programcons(C,_).
member(C, programcons(C,_)).
member(C, programcons(_,P)):- member(C,P).
```

Declarative delay mechanisms as those found in, e.g., the Gödel language [26] are not powerful enough for implementing the constraints of CLP($\mathcal{HCL}$) as predicates with a suitable execution behaviour. It seems to be essential to have the ability

to compare and alter explicitly the list of calls pending on a given variable as is possible using the low-level and not very declarative notion of attributed variables.

It is difficult to give comparative measurements concerning the efficiency of this implementation of demo as there does not seem to exist other systems with the same functionality.

In order to give some kind of measurement, we will compare with a Prolog interpreter, Prolog excuting an object program completely given, of course, and demo with the same program specified to such a degree that it will go through the same control pattern as Prolog.

Without a formal proof, we estimate the time spent on processing a specific instance constraint as being proportional with the size of its first argument. Instance constraints are applied

- once in the DEMO clause (d) to the name of the object query, and
- each time the DEMO clause (d_2) selects a clause in the object program.

In total, instance constraints slow down demo by a constant factor compared with the Prolog interpreter.

The only other significant contribution is given by the no-duplicates constraint in clause (d), which, as we have noticed already, is quadratic in the size of the program. However, for any program of moderate size this will still be small as it is made up from $n * (n-1)/2$ calls of the built-in and efficiently implemented `dif` predicate. Notice also that the no-duplicates constraints can be removed safely from clause (d) of the DEMO program in cases where other conditions anyhow imply that all clauses are different. This is the case in the diagnosis example in section 4.8 below.

4.1.2. ADDITIONAL FEATURES IN THE SYSTEM As already mentioned, the implemented system applies a more detailed naming relation than the one used in this paper, which makes it possible to implement a subtype relation. To see the use of this, consider the following constraint set.

```
term_(X), constant_(X), term_instance_(X,Y,_)
```

The new constraint `constant_` is satisfied for terms of type *constant*, which is a subtype of *term*. The constraint set is satisfiable and results in a unification of `X` and `Y`.

An extended notation is provided to facilitate the use of the naming relation. A Prolog-like syntax is used for the object language with three different operators representing the naming brackets $\lceil \cdots \rceil$ in order to resolve ambiguity, `\` is used for object programs and clauses, `\\` for formulas, atom and constraints, and `\\\` for terms. So, e.g., `\\p(a,X)` is a way of writing a *ground* term which names the $\mathcal{HCL}$ atom $p(a, X)$. A '`?`' operator represents $\lfloor \cdots \rfloor$, so the expression `\\p(a,?Z)` stands for the name of an $\mathcal{HCL}$ atom whose predicate is p, whose first argument is a and whose second argument is unspecified, indicated by the metavariable `Z`.

This extended syntax is compiled away from queries and program files before any computation is performed and the answers correspondingly decompiled before printed out.

The naming relation has been extended to support a concatenation operator '`&`' for programs and a notion of object program modules. In the following,

```
demo( \ ( m1 & m2 & ?P), ...)
```

`m1` and `m2` must be defined as object program modules; the notation for this is shown in the examples below. The 'member' constraints has been extended accordingly as described by the following additional transition rules.

$C \cup \{\text{member}(c, p_1 \,'\&\, p_2)\} \rightsquigarrow C \cup \{\text{member}(c, p_1)\}$

$C \cup \{\text{member}(c, p_1 \,'\&\, p_2)\} \rightsquigarrow C \cup \{\text{member}(c, p_2)\}$

$C \cup \{\text{member}(c, m)\} \rightsquigarrow C \cup \{\text{member}(c, p)\}$
— when m is defined as an object program module containing the program p.

This means that the demo predicate can interpret these new constructs, but it will never invent occurrences of them when used for generating programs.

We illustrate by an example, how these rules work together with the original ones (M1–3). Assume three metalevel predicates `green(...)`, `red(...)`, and `blue(...)`, each defining a condition on programs, and assume they behave in a lazy way by means of coroutine mechanisms (which we illustrate in the examples to follow). Consider the following query.

```
green(Pg), red(Pr), blue(Pb), demo( \ (?Pg & ?Pr & ?Pb), ···).
```

If, now, the internal matters of demo needs to invent a new clause in order to proceed, it will first to expand `Pg` with one 'green' clause. If this leads to failure, it will try next to expand `Pr` with a 'red' one, and if this also fails, an expansion of `Pb` is tried.

Finally, we will mention an optimization of object program modules, which can be viewed as an application of partial evaluation. Whenever an object program module is declared by the user, its clauses are processed once and for all by instance constraints and whenever demo requires a clause from a module, it will pick a preprocessed version, each time with fresh metavariables.

4.2. Abductive frameworks

Kakas, Kowalski, and Toni [33] define an abductive framework as a triplet $\langle A, T, I\rangle$, where T is a theory of background knowledge, A defines the set of possible hypotheses that can be abduced, and I are the integrity constraints. By means of an example from [33], we show how abductive frameworks can be defined in our system. The background knowledge `kb0` is defined as an object program module in the following way.

```
:- object_module( kb0,

     \[ (sibling(X,Y):- parent(Z,X),parent(Z,Y)),
        (parent(X,Y):- father(X,Y)),
        (parent(X,Y):- mother(X,Y)),
         father(john, mary),
         mother(jane,mary)
      ]).
```

The `object_module` procedure stores a representation of the indicated object program in a global Prolog fact; the symbol `kb0` can now be used as a synonym for the program in calls to demo.

Abducibles in this example are the extensional `father` and `mother` predicates. We formalize what it means for a program to consists of extensional facts as follows.

```
:- block extensionals(-).

extensionals( \ []).

extensionals( \ [ (father(?A, ?B):- true) | ?More ]):-
    constant_(A), constant_(B),
    extensionals( More ).

extensionals( \ [ (mother(?A, ?B):- true) | ?More ]):-
    constant_(A), constant_(B),
    extensionals( More ).
```

The block declaration will hold back the execution of a particular call until some other event (e.g., internal `demo` matters) instantiates the argument. The predicate will wake up each time a new clause is added, make sure it is a `father` or `mother` fact with constant arguments, and delay again on the new program tail.

Integrity constraints are defined by a metalevel predicate as follows; the third condition does not appear in [33], but is needed for getting rid of some irrelevant answers.

```
integrity_check(KB):-

    % You can only have one father:
    for_all(  ( constant_(A), constant_(B), constant_(C),
                demo(KB, \\ (father(?A, ?C), father(?B,?C)))),
              A=B ),

    % You can only have one mother:
    for_all(  ( constant_(A), constant_(B), constant_(C),
                demo(KB, \\ (mother(?A, ?C), mother(?B,?C)))),
              A=B ),

    % A mother cannot be a father:
    for_all(  ( constant_(A), constant_(B),
                demo(KB, \\ (mother(?A, ? _), father(?B, ? _)))),
              dif(A,B) ).
```

The `for_all` predicate is implemented in Prolog as follows; it generates all solutions for the first argument and succeeds if and only if the second argument succeeds in all cases.

```
for_all(P,T):-  \+ ( call(P), (call(T) -> fail ; true) ).
```

It is intended to represent the logical statement $\forall$(`P` $\rightarrow$ `T`), but clearly it suffers from the deficiencies of Prolog's approximation to negation as failure.

We have now what is needed to implement a predicate for updating the database such that new observations can be explained.

```
update(KB, Obs, NewKB):-
    extensionals(UpdateFacts),
    NewKB = \ ( ?KB & ?UpdateFacts ),
    demo( NewKB, Obs ),
    close_constraints( NewKB ),
    integrity_check( NewKB ).
```

Given a knowledge base `KB` and some observed facts `Obs`, a new knowledge base `NewKB` is produced. The `close_constraints` predicate is a facility in the system which instantiate remaining metavariables to prototypical values. In this example, it terminates the open program tail of `UpdateFacts` and provides, thus, a correct behaviour of the naively implemented `for_all` device. The following test query shows the overall behaviour of the `update` predicate.

```
?- update( \kb0, \\sibling(mary,bob), N).

N = \ (kb0 & [(father(john,bob):-true)]) ? ;

N = \ (kb0 & [(mother(jane,bob):-true)]) ?

?- update( \kb0, \\ (sibling(mary,bob), mother(joan,bob)), N).

N = \ (kb0&[(father(john,bob):-true),(mother(joan,bob):-true)]) ?
```

4.3. Default reasoning

In [33] it is shown that default reasoning can be simulated as a special case of abduction. Here we show an example from [33] formulated in our system, although in our framework we find it more direct to generate instances of the default rule without introducing an auxiliary abducible predicate; we compare the two approaches following our example. We will later extend the example with abduction and induction.

We represent the factual knowledge together with exceptions to the defaults as an object program in the following way; the second clause defines an exception to the `fly` predicate given by a default rule below.

```
:- object_module( kb0,

     \[ (bird(X,yes):- penguin(X,yes)),
        (fly(X, no):- penguin(X,yes)),
         penguin(tweety, yes),
         bird(john, yes) ]).
```

Due to the lack of negation in our system, we have encoded a truth value in each predicate. Consistency of such a program is defined as follows.

```
consistent(T):-
    for_all( ( constant_(C), constant_(YN1), constant_(YN2),
               demo(T, \\ ( ?P/2-[?C, ?YN1], ?P/2-[?C, ?YN2]))),
```

```
                    YN1=YN2 ).
```

This shows an alternative notation for atoms that allows to parameterize over the predicate symbol. The `consistent` condition is satisfied whenever, for any predicate p and any individual c we do not have $p(c, \texttt{yes})$ and $p(c, \texttt{no})$ at the same time.

This definition of consistency is suited for grounded program representations, typically after `demo` and `close_constraints` have made their job. For larger applications it may be worthwhile writing a new version of the `consistent` predicate to have it execute co-operatively with `demo`.

A default rule such as

fly(X, yes):- bird(X, yes)

is not used as a normal rule, only ground instances of it that do not violate the overall consistency should appear in object level proofs. Ground instances of the default rule is defined at the metalevel as follows.

```
:- block default_instance(-).

default_instance( \ (fly( ?X, yes):- bird( ?X, yes) )):-
    constant_(X).
```

A first version of a query mechanism for single facts can now be put together as follows.

```
query(KB, Q):-
    default_instance( D ),
    demo( \ ( ?KB & [?D]), Q),
    close_constraints( \ ( ?KB & [?D]) ),
    consistent( \ ( ?KB & [?D]) ).
```

The following test queries show the overall behaviour of the `query` predicate.

```
?- query( \kb0, \\fly(john,yes)).
yes

?- query( \kb0, \\fly(tweety,yes)).
no
```

A test print will show for the first query, that the default instance

```
\ (fly(john,yes):- bird(john,yes)
```

is used. Free metavariables can also be used in the query.

```
?- constant_(I), constant_(YN), query( \kb0, \\fly(?I,?YN)).

I = \\\tweety
YN = \\\no ? ;

I = \\\john
YN = \\\yes ? ;

no
```

Without the constraints `constant_(I)` and `constant_(YN)` we would also get answers saying that `I` and/or `YN` could be names of object languages variables.

In [33] default logic is simulated by an abductive framework by introducing an auxiliary abducible predicate `birds_fly` and moving the exceptions to the integrity constraints. The default rule is made part of the database in the following form; we use our own notation for comparison.

```
fly(X, yes):- bird(X, yes), birds_fly(X, yes).
```

New `birds_fly` facts can be abduced provided they do not violate the following integrity constraint.

$\forall$`X penguin(X, yes)` $\rightarrow$ `fly(X, no).`

This can also be programmed in our system by a suitable `for_all` expression.

4.4. Abduction in default reasoning

The principle for abduction shown in section 4.2 applies unchanged in the context of default logic. Again, we must formalize what kind of abducibles, we will allow `demo` to introduce.

```
:- block extensionals(-).

extensionals( \ []).

extensionals( \ [ (bird(?C, yes):- true) | ?More ]):-
    constant_(C),
    extensionals( More ).

extensionals( \ [ (penguin(?C, yes):- true) | ?More ]):-
    constant_(C),
    extensionals( More ).
```

The procedure for abductive update is similar to the first-order case, except that we replace `demo` by the `query` predicate developed in section 4.3, which represents provability in this particular default setting. No integrity constraints are needed for this problem.

```
update(KB, Obs, NewKB):-
    extensionals(UpdateFacts),
    query( \ (?KB & ?UpdateFacts), Obs),
    NewKB = \ ( ?KB & ?UpdateFacts ).
```

The following query shows that the only possible explanation why `peter` flies is that `peter` is a `bird`.

```
?- update( \kb0, \\fly(peter, yes), N).

N = \ (kb0 & [(bird(peter,yes):-true)]) ? ;

no
```

4.5. Induction of defaults-with-exceptions from examples

Due to the declarative nature of the demo predicate, we can easily extend the previous examples of default reasoning with induction.

In the following example, we must imagine a situation in our early childhood where we have become aware of a number of individuals and the categories to which they belong, and we observe that some of them fly and some do not. The exercise to follow corresponds to the formation of a concept of flying in the concrete form of a default rule with exceptions.

In order to mechanize this, we have to define formally at the metalevel the sorts of rules, we will allow the system to invent. The following predicates define for given object level predicates `P`, `Q`, and `Ex` the shape of a default rule "any `Q` is also a `P`" and an exception rule "... except those that are `Ex`". Note that `X` stands for a particular object language variable.

```
default_rule( P,  \ (?P-[X, yes]:- ?Q-[X, yes] )):-
    predicate_(P), predicate_(Q).

exception_rule(P, \ (?P-[X, no]:- ?Ex-[X, yes] )):-
    predicate_(P), predicate_(Ex).
```

We need the following metalevel predicates in order to characterize object programs consisting of instances of a random default rule and object programs consisting of exception rules. We assume predicates

- `default_instances(`D`, `I`)`,
  I a program of ground instances of D, and
- `exception_rules(`P`, `E`)`,
  E a program of exceptions rules for the predicate P,

The induction problem can be stated by the following predicate `aha`.

```
aha(Facts, Obs, Default, Excs):-
    default_rule(Pred, Default),
    default_instances(Default, DefIs),
    exception_rules(Pred, Excs),

    demo(\ (?Facts & ?Excs & ?DefIs), Obs),

    close_constraints(\ (?Facts & ?Excs & ?DefIs) ),
    consistent(\ (?Facts & ?Excs & ?DefIs) ).
```

We test this using the following knowledge base describing some individuals and their categories, but with no knowledge about `flying`.

```
:- object_module( facts,
     \[ (bird(X,yes):- penguin(X,yes)),
        (bird(X,yes):- cock(X,yes)),
         penguin(tweety, yes),
         bird(john, yes),
         bird(peter,yes),
         cock(andrew,yes)  ]).
```

The following queries show the discovery of default-with-exceptions from two dif-

ferent sets of observations.

```
?- aha(\facts, \\ (fly(john,yes), fly(peter,yes), fly(tweety,no),
                   fly(andrew,no)),
       D,E).

D = \ (fly(X,yes):-bird(X,yes))
E = \[(fly(X,no):-penguin(X,yes)),(fly(X,no):-cock(X,yes))]

?- aha(\facts, \\ (fly(john,yes), fly(peter,yes), fly(tweety,no),
                   fly(andrew,yes)),
       D,E).

D = \ (fly(X,yes):-bird(X,yes))
E = \[(fly(X,no):-penguin(X,yes))]
```

In both cases, the same default rule appears, and one or two exceptions are needed depending on whether or not the `cock andrew` is observed to `fly`.

The `aha` predicate can be extended so that more than one default rule can be generated at the same time and we can allow more conditions in their bodies, positive as well as negative; this is a matter of writing more general versions of the metalevel predicates `default_rules` and `exception_rules`.

4.6. Abduction in a fragment of linear logic

Linear logic [19] makes it possible to reason about aspects of process and time in a logical setting. It differs from first order logic in the way that some formulas are considered as resources in the sense that they are consumed when used in a proof. This can implemented in our framework as follows.

For simplicity, we assume here that any clause is a resource in this way. To implement this, we extend the demo predicate with an argument representing the proof defined as a list of names of the clauses that are applied. As a side condition on the proof, we use a predicate `no_dups_list(···)` accepting only lists without duplicates; it can be implemented exactly as the no-duplicates constraint, cf. section 2.4. With this we can define a new demo predicate which implements this little fragment of linear logic as follows.

```
demo_lin(P,Q):- no_dups_list(Proof), demo(P, Q, Proof).
```

The proof predicate `demo_lin` can be used for abduction in exactly the same way as `demo`. Let, e.g., `abducible(···)` describe programs of facts of the sort `drink(tuborg)`, `drink(another_tuborg)`, etc. The following query,

```
?- abducible(D),
   demo_lin(\ [(drunk:- drink(X), drink(Y), drink(Z))] & D, \\drunk).
```

will generate answers where `D` contains at least three facts.

4.7. A natural language example

This example is concerned with the relation between simple still-life scenes and sentences about them. The example illustrates also how integrity constraints can

imply new abducibles to be generated.

Let T be a $\mathcal{HCL}$ program describing a number of things in the world together with some of their properties, e.g., `thing(the_flower)`, `thing(the_vase)`, `thing(the_table)`, `container(the_vase)`. An actual scene is described by another program of facts about the immediate physical relation between the objects, e.g., `in(the_flower, the_vase)`, `on(the_vase, the_table)`. Utterances about a scene are defined by an $\mathcal{HCL}$ program, declared as a module `grammar` in the following way.

```
:- object_module( grammar,

           \  [ (sentence(S):- simple(S)),
                (sentence(S):- folded(S)),
                (simple([X, is, on, Y]):-
                    thing(X),
                    thing(Y),
                    on(X,Y)   ),
                (simple([X, is, in, Y]):-
                    thing(X),
                    thing(Y),
                    in(X,Y)   ),
                (folded([X, is, PREP, Y]):-
                    simple([X, is, _, Z]),
                    simple([Z, is, PREP, Y])  )
              ]).
```

The folded sentence allows us to say 'the flower is on the table' instead of the longer 'the flower is in the vase, the vase is on the table'. Assuming also modules `things` and `scene` defining a particular scene as above, we can use the metainterpreter to execute queries in the normal deductive way, e.g., for testing the correctness of a given sentence.

```
?- demo( \  (grammar & things & scene),
         \\ sentence([the_flower, is, on, the_table]).
```

This model can be extended with abduction so that the program component `scene` can be generated "backwards" from sentences about it. In other words, the problem to be solved is to construct explanations in terms of '`in`' and '`on`' facts which can explain the stated sentences. Any such explanation must satisfy some integrity constraints with respect to the actual `things` theory; an `in` fact, for example, must satisfy the following metalevel predicate.

```
scene_fact(T, \ (in(?A,?B) :- true)):-
    constant_(A),
    constant_(B),
    demo(T, \\ (thing(?A), container(?B))),
    dif(A,B).
```

The `dif(A,B)` condition serves, together with other conditions, to preserve a sensible, physical interpretation of the programs generated. We can write a similar rule for '`on`' and then pack the whole thing together as a predicate `scene_description(`$\lceil T \rceil$`,` $\lceil S \rceil$`)` satisfied whenever S is a sensible scene built from the objects defined by T. An example of the abductive problem can now be stated by the following query.

```
?- scene_description( \things, X),
   demo( \ (grammar & things & ?X),
         \\(sentence([the_flower, is, on, the_table]))).
```

The system produces the following three answers.

```
X = \[(on(the_flower,the_table):-true)]
X = \[(on(the_flower,the_vase):-true),
      (on(the_vase,the_table):-true)]
X = \[(in(the_flower,the_vase):-true),
      (on(the_vase,the_table):-true)]
```

We can also extend the example by abducing a `things` program T in parallel with the `scene`. In case a fact `in(the_dog, the_house)` is abduced, the integrity constraint will abduce in turn as part of T the facts `thing(the_dog)`, `container(the_house)`. Furthermore, the integrity constraint concerned with T (not shown) will trigger the abduction of `thing(the_house)`.

In principle, the example can be extended further with induction, leaving part of the grammar unspecified and having demo to generate it from sample sentences. However, to be of any use, this will require that we are able to formalize at the metalevel, what it means for a grammar to be a good grammar.

4.8. A classical case in diagnosis

Diagnosis as described by Reiter [47] can be viewed as a special case of abduction, but with the extra requirement that minimal explanations are preferred.

We illustrate how diagnosis can be modelled in our system by considering an example from [47], of identifying the faulty components in a logical circuit defining a full-adder. The interesting issue here, compared with previous examples, is the characterization of minimality.

The topology of the full-adder circuit can be described by the following object program.

```
:- object_module( fulladder,
   \ [ (fulladder(A, B, CarryIn, Sum, CarryOut):-
          xorgate(x1, A,       B,       X),
          andgate(a1, A,       B,       Y),
          andgate(a2, X,       CarryIn, Z),
          xorgate(x2, CarryIn, X,       Sum),
          orgate( o1, Y,       Z,       CarryOut) )] ).
```

Each gate in the circuit is marked by an identifier, say `a1`, and its function is determined by a fact of the form `status(a1, S)`. If $S =$ `ok`, the gate behaves correctly as defined by a truth table, in case $S =$ `not_ok`, the behaviour is unpredictable. This is defined by the following object program, where we only show the parts concerned with 'and' gates.

```
:- object_module( gates,
   \ [ (andgate(Ident, In1, In2, Out):-
          status(Ident, ok),
          and(In1, In2, Out) ),
       (andgate(Ident, In1, In2, Out):- status(Ident, not_ok) ),
```

```
    . . .
    and(0, 0, 0), and(0, 1, 0), and(1, 0, 0), and(1, 1, 1),
    . . .
  ]).
```

In case the `status` facts for all gates indicate `ok`, the circuit implements a correct full adder function, in which case the result `fulladder(1,1,0,1,0)` would hold. If an illegal result such as `fulladder(1,1,0,1,1)` is observed it means that one or more of the `status` facts must indicate `not_ok`. By an *explanation* is understod a set of `status` facts that makes it possible to prove the observed results. We assume a metalevel predicate `explanation` which define the shape of programs of `status` facts, one for each gate in the full adder. Now explanations can be found as follows.

```
?- explanation(E),
   demo( \ (fulladder & gates & ?E), fulladder(1,1,0,1,1)).
```

This query yields a total of 37 answers, one of which states that all components are unpredicable. Clearly this can explain any behaviour, but it is not a very useful explanation. A *diagnosis* is defined as a minimal explanation D, meaning that if any `status` of `not_ok` in D is changed into `ok`, the observed behaviour cannot be explained anymore. The definition suggests an obvious way of implementing a test whether an explanation found also is a diagnosis, but this can be optimized in several ways.

First of all, we can use the information provided by the minimal solutions already found to exclude any explanation that is an extension of an existing one. Assume, for example, we have found a diagnosis D stating that the `x1` and `a2` gates are those that are `not_ok`. We can exclude extensions of D by setting up the following condition when searching for a next solutions; the metavariables `Sx1` and `Sa2` are expected to refer to the parts of the new explanation sought that stand for the status value for `x1` and `a2`.

```
dif((Sx1,Sa2),(\\\not_ok,\\\not_ok))
```

We can write a predicate `not_contained_in(Previous, E)` which sets up such `dif` constraints, one for each diagnosis in a list `Previous`.

In practice, it turns out to be easy — referring to the procedural semantics of the underlying Prolog system — to arrange the definition of the `explanation` predicate so that the first explanation found will be a minimal one. With this in mind, we can implement the diagnosis problem as follows; a call `all_diagnoses(`*Observations*`[], Ds)` will generate the list of all correct diagnoses for the given observations.

```
all_diagnoses(Obs, Previous, All):-
   explanation(D),
   not_contained_in(Previous, D),
   demo( \ (fulladder & gates & D), Obs),
   !, all_diagnoses(Observations, [D | Previous], All).

all_diagnoses(_,Ds,Ds).
```

With the observation `fulladder(1,1,0,1,1)`, the answer provides two diagnoses, one stating that `x1` is out of order, and another that `x2` is out of order.

For more complex systems than logical circuits, it may be recommended not to trust the procedural argument about minimal explanations being generated first.

In this case, add a call to a predicate `minimize(E,D,Obs)` to the definition of `all_diagnoses` which extracts a diagnosis `D` from an explanation `E` generated by the call to `demo`.

A. THE RELATION BETWEEN INSTANCE CONSTRAINTS AND SEMIUNIFICATION

Here we investigate the close relation between the constraint satisfaction problem for instance constraints and the undecidable semiunification problem, which has attracted much attention due to its relevance for type checking in polymorphic languages [41, 22, 37].

Definition A.1. An *instance of the semiunification problem* is a finite set of constraints $S = \{s_1 \leq_1 t_1, \ldots, s_n \leq_n t_n\}$ where $\leq_1, \ldots, \leq_n$ are binary constraint symbols and $s_1, t_1, \ldots, s_n, t_n$ terms over some untyped signature; a *general solution* to S is a set of general substitutions[7] $\{\sigma, \theta_1, \ldots, \theta_n\}$ such that $s_i\sigma\theta_i = t_i\sigma$, $i = 1 \ldots, n$.

An *idempotent solution* is a solution in which each substitution is idempotent and in addition, each composition $\sigma\theta_i$ is idempotent.

The *general (idempotent) semiunification problem* is the problem of deciding, for any instance whether it has a general (idempotent) solution.

It has been shown by [34] that the general semiunification problem is undecidable for $n \geq 2$, but whether or not this goes for the idempotent case as well is not known at present. The same authors have also shown that a special case called acyclic semiunification is decidable [35]. Here we define this notion in a less general and simpler form, which is sufficient for our comparisons.

Definition A.2. An instance $S = \{s_1 \leq_1 t_1, \ldots, s_n \leq_n t_n\}$ of the semiunification problem is *acyclic* if the sets of variables occurring in $s_1, \ldots, s_n$ and those occurring in $t_1, \ldots, t_n$ are disjoint. The *general (idempotent) acyclic semiunification problem* is defined analogously as above.

The decidability result of [35] goes for acyclic semiunification with general substitutions and we can argue as follows, that it extends also to the idempotent case: There exist algorithms that produce most general solutions to instances of the general semiunification problem and termination is guaranteed for those instances for which a solution exists [22]. We can apply such an algorithms to a solvable, acyclic instance, and then check whether the solution produced is or subsumes an idempotent one.

In a comparison of instance constraints and semiunification it will appear that instance constraint separate the two levels of instantiation (i.e., the σ, resp. the θs) to object and metalevels, but essentially they express the same as the $\leq$ symbols. The instance constraints' substitution arguments are, qua the substitution well-formedness notion, equivalent with the subscripts on the $\leq$ symbols; the explicit representation of substitutions has been chosen to simplify constraint solving.

[7] A general substitution is an arbitrary mapping from a finite set of variables to terms.

We make the relation between the two problems precise in the following proposition and proof.

Proposition A.1. The satisfiability problem for substitution well-formed sets of instance_{term} *constraints is equivalent with the idempotent semiunification problem.*

The satisfiability problem for safe sets of instance_{term} *constraints is equivalent with the idempotent, acyclic semiunification problem.*

PROOF. We will show that a solution for an instance S of the semiunification problem can be constructed from a satisfier for a set C of substitution well-formed instance constraints and vice-versa.

For this purpose, we introduce two operators defined as follows, o mapping metalevel terms to the object level, and m mapping object level terms to the metalevel.

- For any metavariable x, $o(x)$ is a new, unique object variable.
- For any name of an object variable $'X$, $o('X) = X$.
- Whenever $'f$ names an object function symbol, $o('f(t_1, \ldots, t_n)) = f(o(t_1), \ldots, o(t_n))$.
- For any object variable X, $m(X)$ is a new, unique metavariable.
- Whenever $'F$ names an object function symbol, $m('F(T_1, \ldots, T_n)) = 'F(m(T_1), \ldots, m(T_n))$.

So m is similar to the naming relation except that it maps object variables into metavariables instead of constants; o is like an inverse to the naming relation, which is defined also for nonground metalevel terms.

Firstly, let an instance of the semiunification problem $S = \{S_1 \leq_1 T_1, \ldots, S_n \leq_n T_n\}$ be given, and consider the following constraint set,

$$C = \{\text{instance}_{term}(m(S_1), m(T_1), z_1), \ldots, \text{instance}_{term}(m(S_n), m(T_n), z_n)\},$$

where $z_1, \ldots, z_n$ are new variables. Clearly C is substitution well-formed and, by definition, if S is acyclic, C will be safe. If there exists a satisfier ρ for C, we construct a solution $\{\sigma, \theta_1, \ldots, \theta_n\}$ to S as follows.

- For any object variable X in some $S_1, T_1, \ldots, S_n, T_n$ with $m(X)\rho = \lceil T_X \rceil$, let $X\sigma = T_X$.
- For all i, let $\theta_i = \{X_{i1} \mapsto T_{i1}, \ldots, X_{in_i} \mapsto T_{in_i}\}$, where $z_i\rho = [('X_{i1}, \lceil T_{i1} \rceil), \ldots, ('X_{in_i}, \lceil T_{in_i} \rceil)]$.

The other way round, let a substitution well-formed set of constraints

$$C = \{\text{instance}_{term}(s_1, t_1, w_1), \ldots, \text{instance}_{term}(s_n, t_n, w_n)\},$$

be given. We can assume that all w_i are variables, otherwise, if

$$w_i = [('X_{i1}, t_{i1}), \ldots, ('X_{in_i}, t_{in_i})],$$

replace the ith constraint with

$$\text{instance}_{term}('f(s_i, 'X_{i1}, \ldots, 'X_{in_i}), 'f(t_i, t_{i1}, \ldots, t_{in_i}), w_i').$$

We will also assume that the w_i's are distinct, otherwise if $w_j = w_k$ for some j, k, replace the jth and kth constraint by

$$\text{instance}_{term}('f(s_j, s_k), 'f(t_j, t_k), w_j).$$

Clearly, these transformations do not change the set of satisfiers. Consider now, the following instance of the semiunification problem,

$$S = \{o(s_1) \leq_1 o(t_1), \ldots, o(s_n) \leq_1 o(t_n)\},$$

and assume it has a solution $\{\sigma, \theta_1, \ldots, \theta_n\}$. If C is safe, it follows by definition that S is acyclic. We construct a satisfier ρ for C as follows.

- For any metavariable x in some $s_1, t_1, \ldots, s_n, t_n$, let $x\rho = \lceil o(x)\sigma \rceil$,
- For all i, let $w_i\rho = [(\lceil X_{i1} \rceil, \lceil T_{i1} \rceil), \ldots, (\lceil X_{in_i} \rceil, \lceil T_{in_i} \rceil)]$ where $\theta_i = \{X_{i1} \mapsto T_{i1}, \ldots, X_{in_i} \mapsto T_{in_i}\}$.

□

From proposition A.1, we may conclude that satisfiability of instance constraints (defined in terms of idempotent object substitutions) is likely to be undecidable, but that the special case with safe instance constraints is decidable. The last conclusion follows also from the termination result for our own constraint solver, proposition 3.8.

Proposition A.1 implies also that the transitions rules (Unif), (I), and (It1–3) can be used as a derivation system for solving particular cases of idempotent semiunification. Due to the property of substitution well-formedness, it has not been necessary to have these rules explicitly prevent the formation of non-idempotent substitutions. It may, thus, be possible to use our derivation system for general semiunification problems as well, however, with the risk of nontermination. There seems to be close links between our derivation rules (biased towards the application of Prolog technology) and the semiunification algorithms of [22, 37] although this connection has not been studied in detail.

REFERENCES

1. Bacha, H., Meta-level programming: A compiled approach. *Logic Programming, Proceedings of the fourth international conference*, ed. Lassez, J.-L., MIT Press, pp. 394–410, 1987.
2. Bacha, H., MetaProlog design and implementation. *Logic Programming, Proceedings of the fifth international conference*, ed. Kowalski, R.A., Bowen, K.A., MIT Press, pp. 1371–1387, 1988.
3. Bowen, K.A., Meta-level programming and knowledge representation. *New Generation Computing* 3, pp. 359–383, 1985.
4. Bowen, K.A. and Kowalski, R.A., Amalgamating language and meta-language in logic programming. *Logic Programming*, Clark, K.L. and Tärnlund, S.Å., eds., pp. 153–172, Academic Press, 1982.
5. Bowers, A.F., Gurr, C.A., Towards fast and declarative meta-programming, *Meta-Logics and Logic Programming*, eds. Apt, K.A., Turini, F., pp. 137–166. MIT Press, 1995.
6. Brogi, A., Mancarella, P., Pedreschi, D., and Turini, F., Composition operators for logic theories. *Computational Logic*, ed. Lloyd, J., pp. 117–134, Springer-Verlag, 1990.
7. Christiansen, H., Declarative semantics of a meta-programming language. Bruynooghe, M., ed., *Proc. of the Second Workshop on Meta-programming in Logic.* April 4–6, 1990, Leuven, Belgium. pp. 159–168, 1990.
8. Christiansen, H., Even non-recursive calls of binary demo may loop. *Logic programming, The newsletter of the association for Logic Programming*, 5/4, pp. 16–17, 1992.

9. Christiansen, H., A complete resolution method for logical meta-programming languages. *Lecture Notes in Computer Science* 649, pp. 205–219, 1992.
10. Christiansen, H., A complete demo predicate with co-routine control for automatic program synthesis, Extended abstract, *Proc. Second Compulog Area Meeting on Programming Languages joint with Workshop on Logic Languages, Progretto Finalizzato Sistemi Informatici e Calcolo Parallelo, Pisa, Italy, May 6–7*, 1993.
11. Christiansen, H., Why should grammars not adapt themselves to context and discourse? *4th International Pragmatics Conference, Kobe, Japan, July 23–30 1993*, (Abstract collection), International Pragmatics Association p. 23, 1993.
(Full paper available from the author).
12. Christiansen, H., Efficient and complete demo predicates for definite clause languages. *Datalogiske skrifter* 51, Roskilde University, 1994.
13. Christiansen, H., On proof predicates in logic programming. *Proof-Theoretical Extensions of Logic Programming, Proceedings of an ICLP-94 Post-Conference Workshop*, Momigliani, A., Ornaghi, M. (eds.), CMU, Pittsburgh, PA 15231-3890, USA, 1994.
14. Christiansen, H., Alternative reasoning in a metaprogramming system. Proc. of the ICLP'95 Joint Workshop "Deductive Databases and Logic Programming", "Abduction in Deductive Databases and knowledge-based Systems", Shonan Village Center, Japan, June 17, 1995. *GMD-Studien* 266, Gesellshaft fr Mathematik und Datenverarbeitung MBH, pp. 175–186, 1995.
15. Dovier, A., Omodeo, E.G., Pontelli, E., and Rossi, G., {log}: A Logic Programming Language with Finite Sets. *Logic Programming, Proc. of the Eighth International Conference*, ed. K. Furukawa, MIT Press, pp. 111–124, 1991. .
16. Flach, P., Abduction and induction: syllogistic and inferential perspectives. *Proc ECAI'96 Workshop on Abductive and Inductive Reasoning, Budapest, August 12, 1996*, pp. 31–35, 1996.
17. Gallagher, J.P., *A system for specialising logic programs*. Technical Report TR-91-32, University of Bristol, Department of Computer Science, 1991.
18. Gallagher, J.P., Tutorial on specialisation of logic programs. *Proc. of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93), Copenhagen*, pp. 88–98, 1993.
19. Girard, J.Y., Linear logic. *Theoretical Computer Science* 50, pp. 1-101, 1987.
20. Gödel, K., Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38, pp. 173–198, 1931.
(English translation in [21], pp. 596–616).
21. van Heijenoort, J., ed., *From Frege to Gödel, A Source on Mathematical Logic, 1879–1931*, Harvard University Press, 1967.
22. Henglein, F., *Polymorphic type inference and semi-unification*. Ph.D. thesis, New York University, Department of Computer Science, 1989.
23. Hilbert, D., Matematiche Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongress zu Paris 1900. *Archiv der Mathematik und Physik*, 3rd series, 1, pp. 44–62&213–237, 1901.
(English translation of excerpts in [46], pp. 74–83).
24. Hilbert, D., Über die Grundlagen der Logik und der Arithmetik. *Verhandelungen des Dritten Internationalen Mathematiker-kongress in Heidelberg vom 8. bis 13. August 1904*, Teubner, pp. 174–185, 1905.
(English translation in [21], pp. 129–138).
25. Hill, P.M., Gallagher, J.P., Meta-programming in Logic Programming. To be published in Volume V of *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press.
Currently available as *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.

26. Hill, P.M. and Lloyd, J.W., Analysis of meta-programs. *Meta-programming in Logic Programming*. Abramson, H., and Rogers, M.H. (eds.), MIT Press, pp. 23–51, 1989.
27. Hill, P.M. and Lloyd, J.W., *The Gödel programming language*, MIT press, 1994.
28. Hill, P.M., Topor, R.W., A semantics for typed logic languages. In *Types in Logic Programming*, ed. F. Pfenning, MIT Press, pp. 1–62, 1992.
29. Hobbs, J.R., Stickel, M.E., Appelt D.E., and Martin, P., Interpretation as abduction. *Artificial Intelligence* 63, pp. 69142, 1993.
30. Holzbaur, C., *Specification of constraint based inference mechanisms through extended unification.* Dissertation, Dept. of Medical Cybernetics and AI, University of Vienna, 1990.
31. Holzbaur, C., Metastructures vs. attributed variables in the context of extensible unification. *Lecture Notes in Computer Science* 631, pp. 260–268, 1992.
32. Jaffar, J., Maher, M.J., Constraint logic programming: A survey. *Journal of logic programming*, vol. 19,20, pp. 503–581, 1994.
33. Kakas, A.A., Kowalski, R.A., Toni, F., Abductive logic programming. *Journal of Logic and Computation* 2, pp. 719–770, 1993.
34. Kfoury, A.J., Tiuryn, J., and Urcyczyn, P., The undecidability of the semi-unification problem. *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pp. 468–476, 1990.
35. Kfoury, A.J., Tiuryn, J., and Urcyczyn, P., ML typability is Dexptime-complete. *Lecture Notes in Computer Science* 431, pp. 206–220, 1990.
36. Kowalski, R., *Logic for problem solving*. North-Holland, 1979.
37. Leiß, H., Polymorphic recursion and semi-unification. *Lecture Notes in Computer Science* 440, Springer-Verlag, pp. 211–224, 1989.
38. Lloyd, J.W., *Foundations of logic programming*, Second, extended edition. Springer-Verlag, 1987.
39. Maes, P. and Nardi, D., *Meta-level architectures and reflection*, North-Holland, 1988.
40. McCarthy, J., Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3, pp. 184–195, 1960.
41. Mycroft, A, Polymorphic type schemes and recursive definitions. *Lecture Notes in Computer Science* 167, Springer-Verlag, pp. 217–228, 1984.
42. Muggleton, S. and De Raedt, L., Inductive logic programming: Theory and Methods. *Journal of Logic Programming* 19,20, pp. 629–679, 1994.
43. von Neumann, J., et al., Preliminary discussion of the logical design of an electronic computing instrument. *US Army Ordnance Department Report*, 1946.
    Reprinted in Bell, C.G., Newell, A. (eds.), *Computer Structures: Readings and Examples.* McGraw-Hill, pp. 92–119, 1971.
44. Numao, M. and Shimura, M., Inductive program synthesis by using a reversible meta-interpreter. *Proc. of the Second Workshop on Meta-programming in Logic.* Bruynooghe, M. (ed.), April 4–6, 1990, Leuven, Belgium. pp. 123–136.
45. Peirce, C.S., *Collected Papers of Charles Sanders Peirce.* Hartshorne, C., Weiss, P., and Burks, A.W. (eds.), 8 vols., Harvard University Press, 1931–1958.
46. Reid, C., *Hilbert*, Springer-Verlag, 1970.
47. Reiter, R., A theory of diagnosis from first principles. *Artificial Intelligence* 32, pp. 57–95, 1987.
48. Robinson, A., A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, pp., 23–41, 1965.
49. Sato, T., Meta-programming through a truth predicate. *Logic Programming, Proc. of the Joint International Conference and Symposium on Logic Programming*, ed. Apt, K., pp. 526–540, MIT Press, 1992.
50. *SICStus Prolog user's manual*. Version 3.0, SICS, Swedish Institute of Computer Science, 1995.

51. Warren, D.H.D., Implementing Prolog — Compiling Predicate Logic Programs. *D.A.I. Research Report*, no's 39, 40. Edinbourgh University, 1977.