On solving instance and member constraints

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

1 Introduction

In this paper, we consider the particular problems that arise for the primitive operations in the demo predicate when used "backwards" as a tool for generating programs. The binary proof predicate demo is specified as follows.

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.

useful(X), 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. As demonstrated elsewhere (Christiansen, 1994a, 1994b, 1995, 1996), this principle can be used for defining quite concisely a wide range of reasoning tasks such as abduction, induction, diagnosis, default reasoning, etc. The present paper concerns the methods that are necessary in order to achieve an implementation of demo providing this functionality.

We use the structure of the so-called "instance-demo" predicate which has been studied by several other authors recently (Hill, Gallagher, 1994; Bowers, Gurr, 1995). It replicates SLD-resolution using the following primitive operations,

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

member(C', P')
— C' and P' are names of clause and program C and P with C a member of P.

The instance conditions can express object level unification in an efficient way which operationally is very similar to the use of a nonground representation in the Vanilla interpreter. In the referenced studies, programs are represented as list structures with 'member' being the usual list operation; 'instance' is implemented in a similar straightforward way in Prolog with substitutions represented as lists of variable-value pairs.

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

1. Uninstantiated variables in the first argument to an instance constraint can easily lead the programs of (Hill, Gallagher, 1994; Bowers, Gurr, 1995) into floundering states, so no answer is provided. This arise in the maintenance of substitution arguments.

2. A subgoal instance(X, Y, Z), for variables X, Y, Z, will initiate by backtracking a generation process of all possible names of terms in the object language until one that meets the subsequent subgoals is reached.

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 call `member(a,L), member(b,L)`). Backtracking on failure is thus condemned to loop.

4. Satisfiability for sets of calls to 'instance' (allowing arbitrary use of uninstantiated variables) is closely related to the undecidable semiunification problem, so establishing termination result is problematic.

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

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

With respect to the last problem, we have identified an invariant property called safeness for constraint sets produced by the demo predicate 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.

The constraint solver has been mapped into an executable program in Sicstus Prolog (SICS, 1995) based on a notion of attributed variables, originally suggested by (Holzbaur, 1990, 1992). In this way, the constraints appear as predicates in Prolog with very little addition overhead. When demo is used for specific tasks as outlined above, the defining side-conditions depicted as a 'useful' predicate above, should be instructed to behave in a "lazy" way by means of delay mechanisms in order to have preserve the optimal execution behaviour.

Background

The demo predicate was suggested originally by (Kowalski, 1979) but with only a sketch of an implementation given. Theoretically outlined implementation methods were suggested much later by (Sato, 1992; Christiansen, 1992). It appears that the topic essentially has been awaiting the advent of constraint logic techniques, which we adopted in what seems to be the first implementation with the desired functionality (Christiansen, 1994a, b). The constraint solver described in the present paper is much simpler and illustrative compared to what we have presented earlier, partly because we apply here a *typed* framework, and we have been able to give a termination proof. A comprehensive account on this work, including all proofs, is given as (Christiansen, 1996), here we only state the most important properties.

In section 5, we compare briefly our implementation of sets with the language Set-log (Dovier & al., 1994) which introduces sets in logic programming. Another language with constraints concerned with naming relations is described by (Barklund & al., 1994). They use a form of dynamic reflection between the object and metalanguage which has no equivalent in our approach.

Although they do not consider 'instance' constraints the behaviour of metavariables serving as substitutes for object variables (see section 3) can also be observed in their language. There have been reported other approaches to synthesize programs with demo, but according to our knowledge, only from a fixed collection of clauses given in advance. In this case, a straightforward implementation of demo will do, see e.g., (Brogi & al, 1990), that combine existing programs by operators such as union and intersection.

Overview

In section 2, we give a framework for constraint logic languages based on (Jaffar, Maher, 1994), but extended for typed languages and modified slightly in order to allow an efficient use of metalevel unification (i.e., in the end Prolog unification) for simulating object level unification.

In section 3, we describe a naming relation for the object language $\mathcal{HCL}$ to be embedded in the metalanguage called CLP($\mathcal{HCL}$) and discuss the notational support which is needed for practical reasons. Section 4 concerns instance constraints and the associated derivation rules for constraint solving. In section 5, we take a closer look at the representation of programs and the constraints needed to handle it properly. Thus having introduced the necessary catalogue of constraints, we show in section 6 our version of the 'instance-demo' metainterpreter. The translation of the derivation rules into efficient Prolog programs is sketched in section 7.

2 Typed constraint logic languages

It is quite obvious to suggest a typed metalanguage as the representation of the object language naturally induces a classification of metalevel terms, namely those that stand for programs, those that stand for clauses, etc.

A language is characterized by a *structure* giving a set of type identifiers and ranked function symbols defined in the usual way. For our purposes, we need only consider simple types (called monotypes by Hill, Topor, 1992) and uninterpreted functions. Each particular language includes its disjoint collections of *predicate symbols* and of *constraints symbols*. 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;[1] 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 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 symbol '$=: \tau * \tau$' and '$\neq: \tau * \tau$' 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 be idempotent corresponding to the sort of answers generated by Prolog and for technical simplicity, we define satisfiers and answer substitutions to be ground substitutions. The logical semantics is defined in terms of the following proof relation for ground queries.

[1] These inclusion operators make it easier to define a naming relation as part of typed metalanguage without having to introduce a notion of subtypes.

Definition 2.1 The *proof relation* for a constraint language $\mathcal{L}$, 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.

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

A constraint set C is *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. It is convenient to extend the notions of satisfiers and satisfiability 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.

The object language for our demo predicate is called $\mathcal{HCL}$ defined as a constraint logic language with just one type (corresponding to untyped terms) and no additional constraints, i.e., the language of positive Horn Clauses extended with equality an inequality constraints. The metalanguage CLP($\mathcal{HCL}$), in which demo is to be programmed, is characterized by the types and constraints to be introduced step by step in sections 3–5.

The reason to stick with an untyped object language is a matter of simplicity and it is very easy to adapt our approach to typed object languages as well. Adding new constraints at the object level would require a bit more work.

As a procedural semantics, we use top-down derivation systems in the sense of (Jaffar, Maher, 1994), 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 multiset of literals and α an *accumulated substitution*. States are assumed to be *idempotent* in the sense that no variable $x \in \mathrm{dom}(\alpha)$ occurs in C, in other words, α is mapped consistently over the constraint set C; FAILURE denotes a special state with empty satisfier set.

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

The following transition rules are fixed for any language.

(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 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.

The resolution rule 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.

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

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

where two terms $f(t_1, \ldots, t_n)$ and $f'(t'_1, \ldots, t'_{n'})$ are said to be distinguishable if either f and f' are different, or for some i, that t_i and t'_i are distinguishable.

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

We can characterize the soundness and completeness of a derivation system as follows.

Proposition 2.1 Assume a derivation system for which the following 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.

Then, for any program P and query Q of the given, a correct answer for Q with respect to P is a computed answer and vice-versa. □

It can be shown that the derivation system consisting of the rules (Res), (Unif), (True), (Dif1), and (Dif2) satisfies this property for the $\mathcal{HCL}$ language.

Condition (ii) above 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 ← a=b ∧ p' and 'p'. A fast-solving computation rule yields one failed and one successful derivation whereas a "slow-solving" rule in addition would produce one infinite derivation and infinitely many failed ones.

For analogous reasons, we require also the computation rule be *fast-unifying:*

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

Finally, we have to remark that the characterization in proposition 2.1 turns out to be too naive in many cases. For the derivation system we present below 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).

3 The naming relation

The metalanguage CLP($\mathcal{HCL}$) includes the following types and function symbol, which define a naming relation or ground representation for the object language $\mathcal{HCL}$.

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

For each symbol f of $\mathcal{HCL}$, assume a 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 $X_1, \ldots, X_n$.

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. In our own implemented system, the naming brackets are compiled away before any computation is performed and the answers correspondingly decompiled before printed out.

With this, we do not need any awkward predicates for structuring and destructuring name terms, we can rely on the convenient pattern matching style — which is one of the significant features of logic programming.

4 Instance constraints

The instance condition is represented as constraints in our metalanguage CLP($\mathcal{HCL}$). For each type $\tau \in \{\mathit{clause}, \mathit{atom}, \mathit{constraint}, \mathit{formula}, \mathit{term}\}$, we assume a constraint symbol

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

The type subscript will be left out when unnecessary or when implied by the context. We define as satisfied 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 $P_1\sigma = P_2$.

The following rule (I) expresses that an item can have one and only one instance under a given substitution. Notice this rule also applies immediately following a unification aliasing 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 reduce instance constraints that express bindings to object variables.

(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) do not apply, and $s = [\cdots|w]$; w' is a new variable.

Notice that a fast-unifying computation rule is relevant 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 one and only one representation of a substitution determined by a set of constraints, the one chosen depends on the arbitrary order of applications of rules.

A structured object language term in the first argument of a term instance constraint is decomposed 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.

As noticed in our own work and by other authors, the advantage of using instance constraint to express the semantics of $\mathcal{HCL}$ is that a detailed simulation of unification — and the implied change propagation — are avoided. Instead, it turns out that these things are mapped into the semantics of the metalanguage; we illustrate this by means of an example. Consider the object level equation p(X,a) = p(b,Y), which, by definition, is equivalent to the following metalevel constraints:

$$\text{instance}(\lceil \text{p(X,a)} \rceil, \text{Z}, \text{S}_1), \text{instance}(\lceil \text{p(b,Y)} \rceil, \text{Z}, \text{S}_2)$$

In the reduction performed by the derivation rules, the following identities appear,

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

together with constraints

$$\text{instance}(\lceil \mathrm{X} \rceil, \mathrm{Z_X}, \mathrm{S}_1), \text{instance}(\lceil \mathrm{Y} \rceil, \mathrm{Z_Y}, \mathrm{S}_2),$$

$\mathrm{Z_X}$ and $\mathrm{Z_Y}$ are new metavariables. Now, doing (Unif) and twice (It2) will record the object level bindings of the object variables in the respective substitution terms and all constraints have vanished. Notice that the substitution arguments were used only "locally" for each of the original terms — and, operationally speaking, their purpose is to preserve a consistency in the translation of object variables to the meta-level. Having the two initial instance constraints execute with their own substitution corresponds to renaming the variables apart in the object terms, a feature which makes this pattern perfectly suited in a simulation of a proof predicate.

If one of the parties in the object-level unification had been partly specified, which is possible using name terms with uninstantiated metavariables, the final state would contain delayed but satisfiable instance constraints. It should be emphasized, however, that simple delay mechanisms as found in, e.g., the Gödel language (Hill, Lloyd, 1994), are insufficient for implementing the rules above: The rule (I) implies a comparison across several delayed constraints which cannot be expressed by conditions that only involve the state the arguments of a single goal.

Reduction for instance constraints over types *clause*, *formula*, *atom*, and *constraint* are defined similarly to (It3) except that they are triggered also by the second argument being non-variable, e.g.

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

Soundness

To formulate and prove soundness and completeness for the rules described so far, we need to indicate a notion of equivalence between satisfier sets, because some steps make commitments about the particular representation of an object substitution, but without actually changing which substitution is represented.

We define two satisfiers to be equivalent if, for any variable, they provide values that are either identical or (perhaps different) names of identical object level substitutions. Two sets of substitutions are equivalent if each member of one set has an equivalent counterpart in the other and vice versa.

We can prove (see Christiansen, 1996) that any derivation step performed by one of the rules shown in this section from a state S_1 to S_2 implies that the set of substitutions $[\![S_1]\!]$, each restricted to the variables of S_1, is equivalent with $[\![S_2]\!]$. From this follows soundness; completeness needs the termination result described in the following.

Termination properties

Satisfiability of sets of instance constraints can be shown to be equivalent with the semi-unification problem which has attracted much attention due to its relevance for type checking in polymorphic languages. In has been shown by (Kfoury, Tiuryn, Urcyczyn, 1990a) that semiunification formulated with respect to general substitutions is undecidable, and thus the same thing holds for satisfiability of instance constraints — had this been defined by general substitutions. In our framework, we use idempotent substitutions, and the impact of this restriction is not know at present.

However, a dataflow analysis of the demo program, section 6, together with the derivation rules, reveals that the constraint sets we need consider satisfies a property of safeness under which we can show termination. Actually, such safe constraint sets can be considered a simple case of so-called acyclic semiunification which is decidable (Kfoury, Tiuryn, Urcyczyn, 1990b).

Definition 4.1 A set of 'instance' constraints C 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.

- 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 $\mathcal{HCL}$ variable.

□

Intuitively, external variables stand for unspecified parts of program (or query) text, whereas internal variables are those generated by instance constrains as 'reflections' of object variables. Safeness means that these two usages of metavariables are not confused.

Considering only instance constraints over type *term*, safeness is sufficient to ensure termination. When extending to all types, a strange recurrence problem occurs which may force the derivation system to loop. Also here, we can prove an invariant property that rules out the pathological cases, an example of which is the following.

$$\text{instance}_{formula}(\lceil(true, \lfloor \text{Z} \rfloor)\rceil, \text{Y}, \ldots), \text{instance}_{formula}(\text{Z}, \text{Y}, \ldots)$$

A termination proof covering all constraints in CLP($\mathcal{HCL}$) including those presented in the next section is given in (Christiansen, 1996).

5 On the representation of object programs

In (Hill, Gallagher, 1994; Bowers, Gurr, 1995) studies of an interpreter similar to ours, they view programs as lists of clauses, a choice which is sufficient when object programs are given by the use. The same characterizes the modules given by the Gödel system for handling object programs (Hill, Lloyd, 1994). However, as we identified in the introduction, this approach is too naive when the goal is to use the interpreter "backwards" for synthesizing programs. We argued that it leads to the generation of infinitely many equivalent representations of the same program which in turns prohibit any interesting use of demo.

We will describe our solution below, viewing programs as sets and treating the 'member' condition as a constraint.

First of all, notice that the kind of types normally used in logic programming is too weak to represent sets directly, and we have complemented the type definition with a constraint 'no-duplicates(t)' with the meaning that t of type *program* really is the name of a program (cf. the definition in section 3). It uses an auxiliary 'not-member'.

We define as satisfied any constraint of the forms

- member(c, [$\ldots, c, \ldots$]),

– not-member$(c, [c_1, \ldots, c_n])$, $n \geq 0$ where c is different from any of $c_1, \ldots, c_n$, and

– no-duplicates$([c_1, \ldots, c_n])$, $n \geq 0$ where all $c_1, \ldots, c_n$ are different.

These relations could obviously have been specified as ordinary predicates, but we need a detailed procedural control which can only be expressed in explicit derivation rules.

First of all, consider the member constraint, that are treated in the following way.

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

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*. Rule (M1) applies in the case when a new clause is added to an uninstantiated program or program tail. There is no possible nondeterminism in this case so we avoid to generate different representations of the same object program. As is the case for representations of substitutions discussed earlier, one arbitrary representation is created, being determined by the order in which the member constraints are reduced.

This last property indicates a behaviour which makes it impossible to program 'member' in the declarative part of any usual logic programming language (a Prolog '!' is needed). To formulate and prove soundness, we will need to refine the notion of equivalent satisfier sets also to cope with programs analogous to what we have described for substitutions.

The remaining rules are straightforward.

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

The Set-log language (Dovier & al., 1994), provides an implementation of sets in logic programming which includes also a general mechanism for unification of sets which we do not need for our purposes. Our use of a 'no-duplicates' constraint provides a very small improvement in a very special case. Our derivation rules will produce only one solution to the following,

no-duplicates(P), member(X,P), member(Y,P), X=Y

whereas the analogous expression in Set-log seems to produce two solution for P, one with the element X=Y repeated.

6 The metainterpreter

Here we show our version of the "instance-demo" metainterpreter written in the constraint logic language CLP($\mathcal{HCL}$). We remind that the $\uparrow$ in the second to fourth clause of demo_1 is the the inclusion of atoms and constraints to formulas that we normally prefer to leave out.

$\text{demo}(P, Q) \leftarrow$
$\quad \text{no-duplicates}(P) \wedge$
$\quad \text{instance}(Q, Q_1, _) \wedge$
$\quad \text{demo}_1(P, Q_1).$

$\text{demo}_1(P, \lceil \mathit{true} \rceil) \leftarrow \quad \mathit{true}.$

$\text{demo}_1(P, \lceil \uparrow \lfloor A \rfloor \rceil) \leftarrow$
$\quad \text{member}(C, P) \wedge$
$\quad \text{instance}(C, \lceil \lfloor A \rfloor \leftarrow \lfloor B \rfloor \rceil, _) \wedge$
$\quad \text{demo}_1(P, B).$

$\text{demo}_1(P, \lceil \uparrow \lfloor T_1 \rfloor = \lfloor T_2 \rfloor \rceil) \leftarrow \quad T_1 = T_2.$

$\text{demo}_1(P, \lceil \uparrow \lfloor T_1 \rfloor \neq \lfloor T_2 \rfloor \rceil) \leftarrow \quad T_1 \neq T_2.$

$\text{demo}_1(P, \lceil \lfloor A \rfloor \wedge \lfloor B \rfloor \rceil) \leftarrow$
$\quad \text{demo}_1(P, A) \wedge$
$\quad \text{demo}_1(P, B).$

In addition to the ability to run "backwards", our version differs from those of (Hill, Gallagher, 1994; Bowers, Gurr, 1995) by the treatment of inequality constraints in the object language. Although this looks as a minor thing, it is important because it gives a straightforward way to express exceptions to a predicate without introducing the problem sphere associated with mechanisms such as negation by failure.

7 Mapping the constraint solver into efficient Prolog programs

An efficient implementation CLP($\mathcal{HCL}$) has been achieved using mechanisms in Sicstus Prolog (SICS, 1995) especially suited for constraint solving. In this way, we have been able to represent constraints as Prolog predicates that delay and reactive themselves in a suitable way. This has eliminated the need for an additional layer of overhead that would arise in a straightforward simulation of the derivation system, operating explicitly on sets of constraints by continually searching for patterns to which a derivation rule can apply.

Sicstus Prolog supports a notion of *attributed variables* (Holzbaur, 1990, 1992) which essentially allows the programmer to set, inspect and alter named attributes attached to each Prolog variable. In addition, the programmer may supply a hook predicate `veryfy_attributes` which is called automatically whenever Prolog unifies a variable with another 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` do not carry a different type constraints and 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`)` vanish 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 to identify possible applications of derivation rule (I), and the constraints that remain form the new attribute for the variable `X=Y`.
- Whenever `X` is unified with a structural term, the constraint is reduced (if possible) as specified by the derivation 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)` as attached as an attribute of `X` as well as of `Y`, so 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 derivation rules (ND1–2, NM1–2), with a unconditional delay when arguments 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).
```

References

Barklund, J., Costantini, S., Dell'Acqua, P., and Lanzarone G.A., SLD-Resolution with reflection. *Proc. International Logic Programming Symposium*, Ithaca, New York, November 14–17, 1994.

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.

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.

Christiansen, H., A complete resolution method for logical meta-programming languages. *Lecture Notes in Computer Science* 649, pp. 205–219, 1992.

Christiansen, H., Efficient and complete demo predicate. for definite clause languages. *Datalogiske skrifter* 51, Roskilde University, 1994a.

Christiansen, H., On proof predicates in logic programming. A.Momigliani and M.Ornaghi, eds. *'Proof-Theoretical Extensions of Logic Programming'*, CMU, Pittsburgh, PA 15231-3890, USA. Proceedings of an ICLP-94 Post-Conference Workshop, 1994b.

Christiansen, H., Alternative reasoning in a metaprogramming system. *Deductive databases and abduction — precise reference to be included in final paper!* Proceedings of an ICLP-95 Post-Conference Workshop, Kanagawa, Japan, 1995.

Christiansen, H., Automated reasoning with a constraint-based meta-interpreter. *To appear* 1996. Paper and implemented system available at `http://www.dat.ruc.dk/software/demo.html`.

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. .

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.

Hill, P.M. and Lloyd, J.W., *The Gödel programming language*, MIT press, 1994.

Hill, P.M., Topor, R.W., A semantics for typed logic languages. In *Types in Logic Programming*, ed. F. Pfenning, pp. 1–62, 1992.

Holzbaur, C., *Specification of constraint based inference mechanisms through extended unification.* Dissertation, Dept. of Medical Cybernetics and AI, University of Vienna, 1990.

Holzbaur, C., Metastructures vs. attributed variables in the context of extensible unification. *Lecture Notes in Computer Science* 631, pp. 260–268, 1992.

Jaffar, J., Maher, M.J., Constraint logic programming: A survey. *Journal of logic programming*, vol. 19,20, pp. 503–581, 1994.

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, 1990a.

Kfoury, A.J., Tiuryn, J., and Urcyczyn, P., ML typability is Dexptime-complete. Lecture Notes in Computer Science 431, pp. 206–220, 1990b.

Kowalski, R., *Logic for problem solving.* North-Holland, 1979.

SICStus Prolog user's manual. Version 3.0, SICS, Swedish Institute of Computer Science, 1995.

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.