Efficient and complete demo predicates for definite clause languages

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

To be referred to as: *Datalogiske skrifter* 51 (tech. report), Roskilde University, 1994

Abstract

By means of constraint logic programming, we obtain an implementation of the binary demo predicate which is logically complete. The specification of the predicate is that a call

demo(*prog-repr*, *query-repr*)

is true whenever its arguments represent (ground names of) a program and query, respectively, such that the query is provable from the program.

Completeness implies that the predicate is equally well suited for executing programs as well as for generating them. A variable in the first argument represents, thus, an unknown program fragment and, if possible, the complete demo predicate produces an answer for it which makes the query provable.

A formal derivation system as well as an implemented system is presented. Examples indicate that demo combined with other meta-level conditions may serve as a general problem solving tool for abduction, induction and other kinds of knowledge discovery or program synthesis tasks.

Keywords: demo predicate, meta-programming, meta-interpreter, program synthesis, completeness, constraints, reflection.

Contents

1 Introduction

We present a constraint-based method for implementing the binary proof predicate demo introduced by Kowalski (1979). A call of demo,

demo(*prog-repr*, *query-repr*),

is true whenever its arguments represent (ground names of) program and formula, respectively, of some object language such that the formula is provable from the program.

By the use of constraint techniques we obtain an implementation which is both logically complete and fairly efficient. One source of efficiency comes from the fact that the constraint system employs a reflection of object language variables, and thus unification, by variables and unification in the meta-language.

Completeness is the ability to handle correctly any query, including those with non-ground arguments. For demo, this implies an ability to handle queries with meta-variables standing for unknown pieces of program text. A computed answer yields, thus, program fragments which make given the query argument provable.

In combination with other conditions to a program sought, demo can be used to specify problems such as abduction, induction and other kinds of knowledge discovery or program synthesis tasks. With an implementation as the one we present here, demo can be viewed as a general problem solver in this area.

This expressive power has been known and the demo predicate has attracted much attention as documented, e.g., by (Bowen, 1985; Abramson, Rogers, 1989; Bruynooghe, 1990; Pettorossi, 1992). However, complete implementations have not been reported until recently in simultaneous works by (Sato, 1992) and (Christiansen, 1992b). These results provide important proofs of existence, but none of them provides an implementation of practical relevance; we may also refer to an earlier proposal of (Numao, Shimura, 1990).

In the present work, we introduce a constraint logic language, in the sense of (Jaffar, Lassez, 1987; Jaffar, Maher, 1994), which includes an explicit naming relation for the object language $\mathcal{O}$ and constraints which make it possible to implement a proof relation for $\mathcal{O}$. We give a derivation system for this constraint language which can be shown to be sound and complete. In addition, we give an implementation in Prolog based on this derivation system; the behaviour of constraint solving is obtained by co-routine mechanisms (Sicstus, 1994; Colmerauer, 1982; Naish, 1986). We can show examples that indicate the feasibility of this system for the class of problems point-

ed out. The system and examples files are available by ftp at the address `ftp://mac-dev.ruc.dk/pub/demo`.

Related work

Our implementation of demo is an adaptation to constraint techniques of an interpreter which has been studied by (Gallagher, 1991, 1993; Hill, Gallagher, 1994). The most important auxiliary predicate — and in our work, the most important constraint — is the form instance(t_1, t_2) expressing that t_1 be the name of an object term which has an instance named by t_2. An investigation of the instance condition reveals an inherent reflection of variables from object to meta-level. It turns out that the interpreter *internally* executes with a non-ground representation of the object language analogously to the vanilla interpreter (see, e.g., Hill, Lloyd, 1989). In this way, no detailed and inefficient simulation of unification is needed as in the interpreters of (Kowalski, 1979; Bowen, Kowalski, 1982; Hill, Lloyd, 1994). The mentioned interpreters, which do not use constraints, are not suited for uninstantiated meta-variables standing for parts of the program being executed. In such cases, they tend to end in floundering states or run into loops.

Sato's (1992) meta-interpreter is complete in three-valued logic for a language with existential and universal quantifiers and negation. It is an impressive theoretical achievement but it is not very practical as it assumes an underlying breath-first execution model. The way it treats an uninstantiated meta-variable is to start an effective enumeration of all names of object language phrases.

From a practical point of view, we find it important that the naming relation be supported by the programming language. Without a notational aid, programming about the ground representation, which tends to consist of large and unhandy terms, is really a burden. Our constraint language allows phrases of the object language be written directly inside special brackets $\lceil - \rceil$ with a 'reversed' operator $\lfloor - \rfloor$ to express parameterization and (de-) composition. A similar approach is taken by (Barklund et al., 1994b; Dell'Acqua, 1994) for a constraint-based meta-programming language which addresses other problems than our approach; due to peculiarities in the language semantics only names of ground object language phrases are allowed.

In the Gödel language (Hill, Lloyd, 1994), the naming relation is embedded as an abstract data type only accessible via a collection of pre-defined predicates. Cervesato and Rossi (1992) suggest the use of a dual representation based on strings together with facilities to convert back and forth. We may also refer to (van Harmelen, 1992) who proposes a notion of user-definable

naming relations. A facility of this kind will be a useful addition to our system, when the constraint language is used as a platform to develop new and richer languages.

However, when talking about reflection of variables between the different levels of language, it must emphasized that reflection in our case is an internal matter concerned with implementation only. This is different from systems in which reflection is a part of the expressive power offered to the programmer although there seems to be some semantics similarities. We may refer to (Costantini, Lanzarone, 1989; Barklund et al., 1994a; Barklund et al., 1994b; Dell'Acqua, 1994).

We can show examples of demo applied to abduction with nontrivial integrity constraints[1] and also a modification of the demo predicate for a linear-logic-like language. Applications are in progress for inductive logic programming (e.g., Muggleton, 1992, 1994), belief revision (Gärdenfors, 1992), and updating of deductive databases (e.g., Lloyd, 1987). We may also refer to (Christiansen, 1993) where we advocate the use of demo to handle pragmatic issues of natural language analysis which otherwise is difficult to approach.

We have studied earlier a simplified language with a demo-like facility (Christiansen, 1990, 1992). A complete resolution method was suggested, based on rewriting of equations which essentially is a simplified version of the present constraint framework, however, with no bias towards practical implementation. An overview of the present approach has been presented as (Christiansen, 1994), but the formalization of a constraint language with a derivation system for it is new.

Overview

In section 2 we define a framework with logical and procedural semantics for a class of constraint languages for meta-programming.

A specific meta-programming language CLP(meta($\mathcal{O}$)), and the demo predicated programmed in it in a brief and concise manner appear in section 3.

The hard technical core of this paper is constituted by section 4. We introduce a new and more detailed language, CLP(meta$'$($\mathcal{O}$)), which is better suited for constraint solving. The constraints of CLP(meta($\mathcal{O}$)) are then implemented by clauses written in CLP(meta$'$($\mathcal{O}$)) in the usual way. At present, it is not known whether constraint solving is decidable in general in a domain with instance constraints. Luckily, demo imposes an invariant property

[1] Integrity constraints have been defined in many ways by different authors. Here we refer to the (original?) meaning being a condition that defines a certain class of 'permitted' programs or databases; no specific way to describe such a condition is implied.

on the constraints actually produced, under which termination is given. We present a derivation system which, with a suitable computation rule, provides a sound and complete implementation for programs that preserve this invariant. We outline also how the derivation system is implemented in Prolog with co-routines and we discuss the imperfection implied by the lack of occurs' check and a few other things. Some of the most important proofs are referred to the appendix while others will appear in a forthcoming publication.

Our implemented system together with examples of applications are described in section 5.

2 A class of constraint languages for meta-programming

Constraint techniques appear to be quite useful for meta-programming as the space of values, i.e., names of all phrases in an object language, is quite large and with no obvious structuring by subsumption or the like. This increases the possibility of meta-programs looping under backtracking (the generate-and-test pattern at its worst) or block in floundering states. And constraints are quite effective in situations like that.

We use the constraint logic framework CLP introduced by (Jaffar, Lassez, 1987); for the terminology and an overview, we refer to (Jaffar, Maher, 1994).

2.1 Naming relations

We consider constraint domains over terms with special function symbols defining the naming relation for an object language $\mathcal{O}$. We consider only object languages of positive Horn clauses, so we can define

$$\mathcal{O} =_{\text{def}} \text{CLP}(\emptyset).$$

The naming relation provides a way of representing within a meta-language the syntax of $\mathcal{O}$ such that any phrase of $\mathcal{O}$ is named by a term in the meta-language. The semantics of $\mathcal{O}$ will be supported by the constraint domain we will develop in section 3.

For defining naming relations, we introduce the following.

Definition 2.1 A *template* is a phrase in a given language with zero or more specific subphrases taken out. Whenever $\mathbf{t}$ refers to a given template, the notation $\mathbf{t}[s_1, \cdots, s_n]$ denotes the phrase which arise when the subphrases $s_1, \cdots, s_n$ are inserted at the indicated positions. □

Example 2.1 If $\mathbf{t}$ refers to a template f(g(−), −) of the object language, the notation $\mathbf{t}$[a, b] denotes the object phrase f(g(a), b). □

Definition 2.2 A *naming relation* $\lceil - \rceil$ is a mapping from the phrases of $\mathcal{O}$ to terms of a meta-language, such that for any template $\mathbf{t}$ of $\mathcal{O}$ there exists a unique template in the meta-language, $\mathbf{t}'$, such that for all relevant sequences of $\mathcal{O}$-phrases, $s_1, \cdots, s_n$, it holds that

$$\lceil \mathbf{t}[s_1, \cdots, s_n] \rceil = \mathbf{t}'[\lceil s_1 \rceil, \cdots, \lceil s_n \rceil].$$

Whenever $\mathbf{t}$ is a template of $\mathcal{O}$ and $\mathbf{t}'$ the unique meta-level template indicated by $\lceil - \rceil$, the notation $\lceil \mathbf{t}[\lfloor s'_1 \rfloor, \ldots, \lfloor s'_n \rfloor] \rceil$ is used to denote the term $\mathbf{t}'[s'_1, \ldots, s'_n]$ for any relevant sequence of $\mathcal{O}$-phrase1 $s'_1, \ldots, s'_n$.

If $\lceil P \rceil$ is ground for any object $\mathcal{O}$-phrase P, we say that $\lceil - \rceil$ is a *ground representation* of $\mathcal{O}$.

If for all ground $\mathcal{O}$-phrases P, $\lceil P \rceil$ is ground, but $\lceil v \rceil$ is nonground for any $\mathcal{O}$-variable v, we say that $\lceil - \rceil$ is a *nonground representation* of $\mathcal{O}$. □

The ground representations favors over the nonground with respect to soundness of meta-programming; we refer to (Hill, Lloyd, 1989) and (Hill, Gallagher, 1994). However, under suitably protected and safe conditions[2] the nonground representations has merits with respect to efficiency.

The unique correspondence between templates at the respective levels of languages means that naming relations are compositional (or inductive, homomorphic). This property makes it easier to describe efficient constraint solvers.

Example 2.2 Assume p(a, b) is an $\mathcal{O}$-atom and $\lceil$p(a, b)$\rceil$ the ground following term,

```
atom(predicate(p, 2),
    termlistcons(constant(a),termlistcons(constant(b),emptytermlist))).
```

Then we have, for meta-variable Z,

```
⌈p(a, ⌊Z⌋)⌉ = atom(predicate(p, 2),
        termlistcons(constant(a),termlistcons(X,emptytermlist))).
```

So $\lceil$p(a, $\lfloor$X$\rfloor$)$\rceil$ serves as a compact way of writing the name of an atom of $\mathcal{O}$ whose second argument is parameterized by X. □

[2] A precise meaning to these words is given in definition 4.8.

The naming relation indicated in the example is quite memory consuming as the name of an object term takes up around four times the space as the object term itself, but it definitely allows a very flexible style of metaprogramming. We can move the bias as to have the name take up the same amount of space as the thing it names by a naming relation for which, e.g., $\lceil$p(a, X)$\rceil$ = p(a, *x) where *x is a constant symbol. The price is an ambiguity (a la Prolog) that one cannot tell the difference between a predicate and a function symbol, and metaprogramming (including constraint solving) becomes more difficult.

For practical reasons we find it important to stress the function symbols $\lceil - \rceil$ and $\lfloor - \rfloor$ be part of the meta-programming languages. Without a notational support, programming about the ground representation is really impractical; in the introduction, we compared with other approaches to supporting the naming relation.

But from the formalist's and the implementor's point of view there is no need to pay attention to the naming relation. The special function symbols are only syntactic sugar, which provide a compact way of writing large terms. So if 'con' is a constraint symbol, the following constellation of symbols,

con($\lceil$p(a, $\lfloor$X$\rfloor$)$\rceil$),

may have a meaning identical to this,

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

The semantic expositions to follow are formulated with respect to constraint domains over terms (i.e., finite trees) with no special meaning assigned to the function symbols.

We end this section with a little example, which illustrates some details of our notation.

Example 2.3 In case of names of programs parameterized by especially anonymous variables, the notation may not always be visually pleasing. We provide a few example for the reader to get used to it.

The name of a program consisting of two given clauses:

$\lceil$ [p(a):– *true* , q(X):– p(X)] $\rceil$

The name of a program whose first clause is given and whose program tail is unknown:

$\lceil$ [p(a):– *true* | $\lfloor$_$\rfloor$] $\rceil$

The name of a program whose first clause and program tail are unknown:

$$\lceil\, [\, \lfloor \_ \rfloor \mid \lfloor \_ \rfloor\,]\, \rceil$$

□

2.2 Logical and procedural semantics

We consider constraint languages $\textsc{Clp}(\mathcal{X})$ where the parameter $\mathcal{X}$ refers to a domain of constraint over terms. The meaning of $\mathcal{X}$ is given by a set of ground constraints referred to as *satisfied* constraints. Without mentioning, any $\mathcal{X}$ is assumed to include the constraint symbol '=' with the usual meaning as syntactic identity.

A constraint set C is *satisfiable* if there exists a substitution[3] μ 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. In general, we prefer to use ground substitutions as answer substitution. There is no deep reason for not using arbitrary substitiutions, but it simplifies the formal statements.

In this paper we are interested in proof predicates and we prefer, thus, to think of logical semantics in terms a proof relation defined as follows.

Definition 2.3 The *proof relation* for a constraint language $\mathcal{L} = \textsc{Clp}(\mathcal{X})$, denoted $\vdash_{\mathcal{L}}$, between programs and ground queries is defined inductively as follows.

- $P \vdash_{\mathcal{L}}$ *true* for any program P.
- Whenever P has a clause with a ground instance H:–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 $P \vdash_{\mathcal{L}} (A, B)$.
- $P \vdash_{\mathcal{L}} C$ 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 $P \vdash_{\mathcal{L}} Q\sigma$. □

[3]For simplicity, we consider only idempotent substitutions, i.e., mappings σ from finite sets of variables such that for no $x, y \in \mathrm{dom}(\sigma)$ that y occurs in $x\sigma$; we refer to (Ko, Nadel, 1991) for a comparison of different kind of substitutions. We will also claim that idempotent substitutions are more natural than general or so-called identity-free substitutions (as used, e.g., by Lloyd, 1987) for the description of logic programming systems; however, we have not worked out a theoretical support for this wiewpoint. For any idempotent substitution σ, we have $\sigma\sigma = \sigma$. We tend to use the letter μ for substitutions at the meta-language level and σ for object language substitutions.

This provides a logical semantics for the object language $\mathcal{O}$ as well as the meta-languages to be introduced about it. With the simple object language we consider, this semantics is equivalent to a conventional, model-based semantics.

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.

As a procedural semantics for CLP($\mathcal{X}$) we use top-down derivations in the sense of (Jaffar, Maher, 1994). We deviate, however, a little from their framework in the following respects.

- We want the notion of derivation to reflect the use of a Prolog-like platform with a built-in, efficiently implemented unification operation. Therefore, the 'passive' parts of our constraint stores are limited to hold a substitution.

- We want derivation rules which reflect implementations in Prolog-like languages with co-routine mechanisms. So transitions rules for constraint solving are restricted as to resemble what one can expect as procedures programmed in such a language.

We do not, however, give a formal framework which mirrors such architectures in all details. Here we review briefly our terminology.

A procedural semantics for CLP($\mathcal{X}$) is defined in terms of *transition rules* over states of the form

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

where I is a multiset of *input* atoms and constraints waiting to be processed, C a multiset of *accumulated constraints* and α an *accumulated substitution*, which represents the explicit variable bindings made so far. Equations may only occur in I and we assume no variable $x \in \mathrm{dom}(\alpha)$ occurs in I or C.[4]

Accumulated constraints correspond to what in some systems are called frozen calls; waking up a frozen call means in our framework to move it over to the input component of a state.

We extend notions of satisfiers (with respect to program P), etc., to states $\langle I, C, \alpha \rangle$ by considering the union of the components, viewing here a substitution as a set of term equations. We assume also a special state called FAILURE with $[\![\text{FAILURE}]\!] = \emptyset$. We notice the following little property which is useful when arguing about the existence of satisfiers for a particular state.

[4] In other words, we assume the α consistently be mapped over I and C. This convention seems a natural generalization of the idempotent property for substitutions.

Proposition 2.1 A state $\langle I, C, \alpha \rangle$ has satisfiers if and only if $I \cup C$ has satisfiers. $\Box$

Transition rules take the form

$$S \rightsquigarrow S'$$

where S and S' are expressions ranging over states. We use also $\rightsquigarrow$ to denote the *derivation relation* induced in the natural way by a set of transition 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 is finite and ends in a final state of the form $\langle \emptyset, C, \alpha \rangle$. Whenever $\langle Q, \emptyset, \emptyset \rangle \rightsquigarrow^* S$ is a successful derivation, any substitution in $[\![S]\!]$ restricted to the variables of Q is called a *computed answer* for Q.

We use the following notation when describing derivation rules. A most general unifier of two atoms (or terms) A and B is denoted $\mathrm{mgu}(A, B)$. In case $\mathrm{mgu}(A, B)$ does not exist for some A and B, the notation $S(\mathrm{mgu}(A, B))$, for some state S, is interpreted as FAILURE. For brevity we assume $I \cup \{true\} = I$.

The following two derivation rules constitute the core of any transition system. They are called *resolution* (with respect to a program P) and *unification*.

$\langle I \cup \{A\}, C, \alpha \rangle \rightsquigarrow_{\mathrm{Res}} \langle I \cup \{B_1, \ldots, B_n, A = H\}, C, \alpha \rangle$
— whenever H:–$B1, \ldots, B_n$ is a variant with new variables of a clause in P.

$\langle I \cup \{A = B\}, C, \alpha \rangle \rightsquigarrow_{\mathrm{Unify}} \langle I \cup C, \emptyset, \alpha \rangle(\mathrm{mgu}(A, B))$.

In addition to actually performing the mgu operation[5], the unification rule also corrects for an aliasing problem which we illustrate by a little example. Assume 'cold' and 'hot' are constraints such that for no x cold(x) and hot(x) can hold at the same time and consider the satisfiable constraint set {cold(X), hot(Y)}. Following the unification of X and Y, it becomes unsatisfiable.

We have chosen here simply to degrade all accumulated constraints to input constraints. — In practice one would prefer a mechanism which only triggered a small number of constraints, being a subset of those containing variables actually affected by aliasing.

If we needed a derivation system for the object language $\mathcal{O} = \mathrm{CLP}(\emptyset)$ we could do with these two rules; here there are no accumulated constraints and thus need to take the aliasing problem into account.

[5] As we have restricted ourselves to idempotent substitutions, we have to assume also that the actual mgu is chosen such that composition with α actually is defined.

We allow only the follow two sorts of rules to be defined for solving specific constraint systems. As in a Prolog execution state, the accumulated substitution is an omnipresent and untouchable object, so we leave it out of the rules assuming it be copied unchanged.

Add rules:

$$\langle I \cup \{c\}, C\rangle \rightsquigarrow \langle I \cup \{c_1, \cdots, c_n\}, C'\rangle$$

Reduction rules:

$$\langle I, C \cup \{c\}\rangle \rightsquigarrow \langle I \cup \{c_1, \cdots, c_n\}, C\rangle$$

In both cases, $c, c_1, \cdots, c_n$ refer to constraints, and c must be different from equations. Intuitively, an add rule takes an input constraint and merges it with the already accumulated constraints ensuring their common satisfiability; the selected input constraint may or may not actually need to be added. In some cases an add rule implies also new 'derived' input constraints in order to preserve invariant properties. A reduction rule splits an accumulated constraint which has become 'unstable' into simpler constraints which must go through the add rules again.

Definition 2.4 A *derivation system* for CLP($\mathcal{X}$) is a set of transition rules $\{\rightsquigarrow\}$ which includes $\rightsquigarrow_{\text{Res}}$ and $\rightsquigarrow_{\text{Unify}}$ such that the conditions (i), (ii) hold.

(i) (Preservation of satisfiers)
Whenever $S \rightsquigarrow S'$, excluding resolution steps, the set of substitutions $[\![S']\!]$, each restricted to the variables of S, coincides with $[\![S]\!]$.

(ii) (Termination in constraint solving)
For any state of the form $S = \langle I_c, C, \alpha\rangle$, with I_c a finite set of constraints, there is a successful derivation $S \rightsquigarrow^* S'$. In this case $S' =$ FAILURE if and only if S is unsatisfiable.

□

The unification rule satisfies a priori condition (i) and the resolution rule a similar condition taking the union of satisfiers over the possible resolution steps, one for each clause in the program. We can show the following soundness and completeness result.

Proposition 2.2 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. □

In this definition and proposition we ignored the possible need for a specific computation rule, i.e., a protocol defining an order in which input literals can be selected. For the constraint language CLP(meta'($\mathcal{O}$)) presented in section 4.2 we will need both an invariant property and a computation rule to ensure termination.

2.3 Stepwise refinement in implementation of constraint languages

In section 3 we introduce a constraint language which is perfect for writing a meta-interpreter, but with the drawback that it is difficult to write an efficient derivation system for it.

In general, as well as in our case special case, it can be advantageous to modularize the implementation of a language by first programming its constraints as clauses in another language. This other language, then, may include new constraints whose intuitive interpretation may not be so obvious but whose purpose is to simplify the construction an efficient derivation system.

Definition 2.5 Let CLP($\mathcal{X}$) and CLP($\mathcal{Y}$) be constraint languages such that the constraint symbols of $\mathcal{X}$ appear as predicate symbols in CLP($\mathcal{Y}$). An *implementation of* CLP($\mathcal{X}$) *in* CLP($\mathcal{Y}$) is a CLP($\mathcal{Y}$)-program $P_{\mathcal{X}}$ such that for any ground constraint c of $\mathcal{X}$, that

c is satisfied if and only if $P_{\mathcal{X}} \vdash_{\mathrm{CLP}(\mathcal{Y})} c$

□

The definition generalizes immediately to the following soundness and completeness result.

Proposition 2.3 Assume an implementation $P_{\mathcal{X}}$ of CLP($\mathcal{X}$) in CLP($\mathcal{Y}$) together with a derivation system for CLP($\mathcal{Y}$). Then, for program P and query Q of CLP($\mathcal{X}$), any correct answer for Q with respect to P (provided by $\vdash_{\mathrm{CLP}(\mathcal{X})}$) is a computed answer for Q with respect to $P \cup P_{\mathcal{X}}$ and vice-versa. □

Although not made explicit in the definition, an implementation of one constraint language in another should only allow a call of an 'implemented constraint' to succeed once as to preserve the overall behaviour of constraints.

3 A sound and complete meta-interpreter

3.1 The meta-language CLP(meta($\mathcal{O}$))

We assume as earlier mentioned an object language $\mathcal{O}$ and a naming relation for it, which we denote $\lceil - \rceil$.

The following constraint domain is especially tailored for writing a specific meta-interpreter for $\mathcal{O}$.

Definition 3.1 Let meta($\mathcal{O}$) be a constraint domain which include following,

- *type constraints* of the form

  $$\tau(t)$$

  for any syntactic category τ of $\mathcal{O}$ which is satisfied if and only if $t = \lceil P \rceil$ where P is an $\mathcal{O}$-phrase of category τ,

- *instance constraints* of the form

  $$\tau\text{-instance}(t_1, t_2)$$

  for $\tau \in \{\text{clause, formula, atom, term}\}$ which is satisfied if and only if $t_1 = \lceil P_1 \rceil$, $t_2 = \lceil P_2 \rceil$ where P_1, P_2 are $\mathcal{O}$-phrases of category τ with P_2 being an instance of P_1.

□

Instance constraints provide a way to express object level unification at the meta-level. To see this, we state in the following propositions which follow directly from the definitions.

Proposition 3.1 Let P_1, P_2, and P be object level phrases of category $\tau \in \{\text{clause, formula, atom, term}\}$ and σ an object level substitution with $P_1\sigma = P_2'\sigma = P$, P_2' a copy of P_2 whose variables have been renamed away from those of P_1. Then the constraints

$$\tau\text{-instance}(\lceil P_1 \rceil, \lceil P \rceil) \text{ and } \tau\text{-instance}(\lceil P_2 \rceil, \lceil P \rceil)$$

are satisfied. □

The renaming of variables in P_2 is quite natural as P_2 in the actual applications in the DEMO program below corresponds object language clauses.

Proposition 3.2 Assume that the following constraint set is satisfiable with satisfier μ,

$\{\tau\text{-instance}(p_1, p_1'),\ \tau\text{-instance}(p_2, p_2')\}$

with $p_1'\mu = p_2'\mu = p$. Then there exists object language phrases P_1, P_2, and P, and object level substitution σ such that $p_1\mu = \lceil P_1 \rceil$, $p_2\mu = \lceil P_2 \rceil$, $p = \lceil P \rceil$ and $P_1\sigma = P_2'\sigma = P$, where P_2' is a copy of P_2 whose variables have been renamed away from those of P_1. □

Instance constraints imply a reflection at the implementation level of object variables to meta-variables, which is very favourable when it comes to the construction of derivation systems for meta($\mathcal{O}$). To characterize this, we introduce some notation.

Definition 3.2 For any object language phrase P, the notation $\lceil P \rceil^\circ$ stands for a meta-level term created by exchanging all subphrases in $\lceil P \rceil$ which are names of variables consistently with new meta-variables. □

The $^\circ$ decoration is chosen to express the intuition that the new variables thus inserted serve as a kind of holes in the name term. Without formalizing it,[6] we can think of $\lceil \cdots \rceil^\circ$ as providing a non-ground representation of the object language, as opposed to the ground representation provided by $\lceil \cdots \rceil$.

Proposition 3.3 For any object language phrase P of category $\tau \in \{$clause, formula, atom, term$\}$, the constraint

$\tau\text{-instance}(\lceil P \rceil, t)$

is satisfied if and only if t is an instance (at the meta-level) of $\lceil P \rceil^\circ$ for which $\tau(t)$ is satisfied.

Any two object language phrases P_1 and P_2 of type $\tau \in \{$clause, formula, atom, term$\}$ has a common instance (at the object level) if and only $\lceil P_1 \rceil^\circ$ and $\lceil P_2 \rceil^\circ$ has a common instance (at the meta-level). □

The unification of two object phrases P_1 and P_2, which is relevant in a meta-interpreter, can thus be implemented by two calls of instance constraints, each of which constructs its respective $\lceil P_i \rceil^\circ$ — and the meta-language unification $\text{mgu}(\lceil P_1 \rceil^\circ, \lceil P_2 \rceil^\circ)$ does the rest of the work.

In other words, the use of instance constraints implies an efficient internal non-ground representation without the negative soundness properties often associated with nonground representations of the object language (Hill, Lloyd, 1989; Hill, Gallagher, 1994).

[6] Technically speaking, $\lceil \cdots \rceil^\circ$ is not a nonground representation unless the correspondence between object and meta-variables is made unique, which in turn implies a need for explicit renaming

Example 3.1 For an object language atom p(a, X), the expression $\lceil p(a, X)\rceil$ denotes a ground meta-level term; $\lceil p(a, X)\rceil^\circ$ is a non-ground meta-level term which alternatively can be characterized as $\lceil p(a, \lfloor Z\rfloor)\rceil$ where Z is a meta-variable.

The object-level substitution

$$\sigma = \{X \mapsto b, Y \mapsto a\}$$

is a unifier for the object language atoms p(a, X) and p(Y, b).

The meta-level substitution

$$\mu = \{Z_1 \mapsto \lceil b\rceil, Z_2 \mapsto \lceil a\rceil\}$$

is a unifier for the meta-level terms $\lceil p(a, \lfloor Z_1\rfloor)\rceil$ and $\lceil p(\lfloor Z_2\rfloor, b)\rceil$. □

3.2 The meta-interpreter

Using CLP(meta($\mathcal{O}$)), we can easily write a meta-interpreter which simulates $\mathcal{O}$'s proof relation. The following meta-program will be referred to as DEMO. The underline character is used as in Prolog to indicate an occurrence of a variable which do not occur elsewhere, for object programs we use a traditional list notation such that $[c|cs]$ stands for a program with a first clause c and a program rest cs; see also example 2.3.

```
demo(P, Q):–
    program(P),
    formula-instance(Q, Q1),
    demo1(P, Q1).

demo1(P, ⌈true⌉):–  true.

demo1(P, A):–
    member(C, P),
    clause-instance(C, ⌈ ⌊A⌋:– ⌊B⌋ ⌉),
    demo1(P, B).

demo1(P, ⌈ (⌊A⌋, ⌊B⌋)⌉ ):–
    demo1(P, A),
    demo1(P, B).

member(C, ⌈ [ ⌊C⌋ | ⌊_⌋ ] ⌉):–  true.

member(C, ⌈ [ ⌊_⌋ | ⌊P⌋ ] ⌉):–
    member(C, P).
```

This meta-interpreter has been studied by (Hill, Gallagher, 1994), however, without the use of constraint techniques for the primitive operations. It is easy to prove the following.

Theorem 3.1 Let P and Q be any object program and formula. Then

$$\textsc{Demo} \vdash_{\textsc{Clp}(\text{meta}(\mathcal{O}))} \text{demo}(\lceil P \rceil, \lceil Q \rceil)$$

if and only if there exists an object level substitution σ such that $P \vdash_{\mathcal{O}} Q\sigma$. □

The theorem follows from the following lemma which is straightforward to prove by induction using propositions 3.1 and 3.2.

Lemma 3.1 Let P, Q, and σ be object program, formula, and substitution. Then

$$\textsc{Demo} \vdash_{\textsc{Clp}(\text{meta}(\mathcal{O}))} \text{demo}_1(\lceil P \rceil, \lceil Q\sigma \rceil)$$

if and only if $P \vdash_{\mathcal{O}} Q\sigma$. □

Referring to proposition 2.2 we get the following theorem which captures the soundness and completeness of querying using a derivation system for $\textsc{Clp}(\text{meta}(\mathcal{O}))$.

Theorem 3.2 Let $\{\rightsquigarrow\}$ be a derivation system for $\textsc{Clp}(\text{meta}(\mathcal{O}))$.

For any meta-level terms p and q and meta-level substitution μ, the following two properties are equivalent.

- μ is an answer computed for query $\text{demo}(p, q)$ with respect to program Demo.
- There exist object program and query P and Q and object level substitution σ such that $p\mu = \lceil P \rceil$, $q\mu = \lceil Q \rceil$ and $P \vdash_{\mathcal{O}} Q\sigma$.

□

Example 3.2 It follows from theorem 3.2 that the set of all computed answers for the query

demo(X, Y)

characterizes the full proof relation for the object language.

For given object program P the set of all computed answers for the query

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

characterizes the set of all object formulas which are logical consequences of P.

Assuming some predicate abducible(−) programmed in CLP(meta($\mathcal{O}$)), the set of all computed answers for the query

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

characterizes the set of all object clauses whose name satisfies the abducible(−) condition and which together with the clauses of P can explain the 'observations' Obs. □

However, these results are only interesting if we can provide a suitable derivation system such as the one developed in section 4

3.3 Alternative demo-predicates

We consider in detail only the demo predicate defined above by the DEMO program. Here we will discuss briefly other ways to design the 'user-interface' to a proof predicate.

Intuitively, demo may be a bit disappointing to use as it does not provide any information about the object level substitution which made the query succeed. In this respect, the demo_1 predicate of the DEMO program can be used as a proof predicate which seems slightly more satisfactory. According to lemma 3.1 we can specify it as follows.

$$\text{demo}_1(\lceil P \rceil, \lceil Q \rceil) \quad \equiv \quad P \vdash_{\mathcal{O}} Q$$

In computed answers we will see values for meta-variables in Q that reflect object level substitutions.

The following demo_2 is perhaps a better reflection at the meta-level of the functionality provided by a normal query system as it explicitly returns (a representation of) the computed object level answer substitution.

$$\text{demo}_2(\lceil P \rceil, \lceil Q \rceil, \lceil \sigma \rceil) \quad \equiv \quad P \vdash_{\mathcal{O}} Q\sigma$$

The demo_2 predicate can easily be implemented in the language CLP(meta$'(\mathcal{O})$) introduced in section 4.2, which contains a representation of object level substitutions. The implementation of demo in CLP(meta$'(\mathcal{O})$) (via CLP(meta($\mathcal{O}$))) do actually compute the mentioned answer substitution.

The alternative predicates discussed here allow the user to produce constraint set which are outside the class of safe constraint sets (section 4.4.2, below) for which we are sure of termination in constraint solving. However, the safeness condition is ensured if no meta-variable is placed more than one argument to demo_1 or demo_2.

4 A sound and complete implementation of the meta-language

In this section we describe the long and troublesome Odyssey from the constraints of meta($\mathcal{O}$) leading to a correct derivation system eventually implemented in Prolog.

We define in section 4.1 the details of a specific naming relation and develop, in section 4.2, a constraint language CLP(meta'($\mathcal{O}$)) based on it. The language CLP(meta'($\mathcal{O}$)) has more kinds of constraints than meta($\mathcal{O}$) and familiar constraints are extended with new arguments, all this to make the construction of an efficient derivation system simpler. In section 4.3 we give the implementation of meta($\mathcal{O}$) in meta'($\mathcal{O}$). Section 4.4 introduces the necessary invariants for proving the correctness of the derivation system introduced in section 4.6. The notion of safeness of a constraint set, which is crucial for termination, is discussed informally in section 4.5.

The main results are stated in section 4.7 and some of the detailed proofs are given in the appendix. Finally, section 4.8, we describe an implementation of the derivation system in Prolog together with the small imperfections this introduces.

4.1 A specific naming relation

For defining CLP(meta'($\mathcal{O}$)), we will decide upon a specific naming relation based on the following notions of types. There are also types concerned with a representation of object level substitutions which formally is not part of the naming relation; this is explained later. The function symbols and constants appearing in table 1 are expected to be part of the meta-language CLP(meta($\mathcal{O}$)) and will be included also in CLP(meta'($\mathcal{O}$)) to be defined; the set of meta-level constants is expected to include integers $0, 1, \ldots$

Definition 4.1 A number of *types* are defined which appear as nonterminals in the grammar of table 1; the types 'predicate', 'function', 'meta-constant', meta-id', and 'meta-int' are called *lexical types*.

In case of a rule

$$\langle\tau\rangle ::= \cdots \mid \langle\tau'\rangle \mid \cdots$$

we say that τ' is a *subtype* of τ, written $\tau > \tau'$; we write $\tau \geq \tau'$ if either $\tau > \tau'$ or $\tau = \tau'$. Additionally, we assume a type $\perp$ (corresponding to type conflict). The *intersection type* for two types τ_1 and τ_2, denoted $\tau_1\tau_2$, is defined as follows,

⟨program⟩ ::= ⟨program-cons⟩ | ⟨empty-program⟩

⟨program-cons⟩ ::= programcons(⟨clause⟩, ⟨program⟩)

⟨empty-program⟩ ::= emptyprogram(end)

⟨clause⟩ ::= clause(⟨atom⟩, ⟨formula⟩)

⟨formula⟩ ::= ⟨true⟩ | ⟨atom⟩ | ⟨conjunction⟩

⟨true⟩ ::= true(end)

⟨atom⟩ ::= atom(predicate(⟨meta-id⟩, ⟨meta-int⟩),
⟨term-list⟩)

⟨conjunction⟩ ::= conjunction(⟨formula⟩, ⟨formula⟩)

⟨term⟩ ::= ⟨variable⟩ | ⟨constant⟩ | ⟨structure⟩

⟨variable⟩ ::= variable(⟨meta-identifier⟩)

⟨constant⟩ ::= constant(⟨meta-constant⟩)

⟨structure⟩ ::= structure(function(⟨meta-id⟩, ⟨meta-int⟩),
⟨term-list⟩)

⟨term-list⟩ ::= ⟨term-list-cons⟩ | ⟨empty-term-list⟩

⟨term-list-cons⟩ ::= termlistcons(⟨term⟩, ⟨term-list⟩)

⟨empty-term-list⟩ ::= emptytermlist(end)

⟨predicate⟩ ::= predicate(⟨meta-id⟩, ⟨meta-int⟩)

⟨function⟩ ::= function(⟨meta-id⟩, ⟨meta-int⟩)

⟨meta-constant⟩ ::= ⟨meta-id⟩ | ⟨meta-int⟩

⟨meta-int⟩ ::= *i*

– *for all numbers i in the meta-language*

⟨meta-id⟩ ::= *id*

– *for all constants id in the meta-language different from integers*

⟨substs⟩ ::= ⟨subst-cons⟩ | ⟨empty-subst⟩

⟨subst-cons⟩ ::= substscons((⟨meta-id⟩, ⟨term⟩), ⟨substs⟩)

⟨empty-subst⟩ ::= emptysubst(end)

Table 1: Grammar for the ground representation

– $\tau_1\tau_2 = \tau_2\tau_1 = \tau_2$ whenever $\tau_1 \geq \tau_2$,

– $\tau_1\tau_2 = \bot$ in all other cases.

In case of a rule

$$\langle\tau\rangle ::= \mathbf{t}[\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle]$$

we say that $\mathbf{t}$ is the *defining template* for τ and that the type τ_i is the *derived type* for the ith component. We may also use the notation $\mathbf{t}[\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle]$ for the defining pattern when we want to indicate the derived types.

The *terms of type* τ, for all types τ, are defined inductively according to the grammar in table 1; there is no term of type $\bot$. A term of type 'term-list' of the following form,

term-list-cons($t_1, \ldots,$ term-list-cons(t_n,emptytermlist(end))$\cdots$)

is said to be of *length* n ($n \geq 0$).

Whenever a term t has type τ with no τ' such that $\tau > \tau'$, τ is a *most specific type* for t.

A given term is said to be *term-list consistent* if, for any subterm of it of the form atom(predicate(p, n), l) or structure(function(f, n), l), that l is of type 'term-list' with length n with $n \geq 0$ for atoms and $n \geq 1$ for structures. □

Proposition 4.1 For any term t, if t has types τ and τ' with $\tau' \neq \tau$, then $\tau > \tau'$ or $\tau' > \tau$; t has exactly one most specific type. □

The strange-looking compound terms such as emptyprogram(end) in table 1 are needed to make the terms of type 'meta-id' disjoint with the other types. The separate type for constants is in principle redundant as a constant can be considered a structure with zero proper subterms. Our choice is motivated by economical reasons as

constant(a)

takes up much less space than

structure(function(a,0), emptytermlist(end)).

Notice that the template for 'predicate' (and 'function') is repeated in the template for 'atom' ('structure'); there is no fundamental reason for this, but it makes it easier to describe the derivation system of section 4.6.

Notation For brevity we will use the same Prolog-like list notation for terms of types 'program', 'term-list', and 'subst'. So depending on context, $[\cdots|\cdots]$

may stand for either programcons(–,–), termlistcons(–,–), or substcons(–,–). Analogously for the emptylist symbol [ ]. □

In the rest of this paper, we let $\lceil - \rceil$ refer to a specific naming relation characterized as follows.

Definition 4.2 The naming relation $\lceil - \rceil$ for $\mathcal{O}$ is defined inductively as follows.

- There is assumed a one-to-one correspondence between variables v of $\mathcal{O}$ and meta-level constants v' of type 'meta-id'. For any such variable v, we let $\lceil v \rceil = \text{variable}(v')$.
- There is assumed a one-to-one correspondence between constants c of $\mathcal{O}$ and meta-level constants c' of type 'meta-constant'. For any such constant c, we let $\lceil c \rceil = \text{constant}(c')$.
- There is assumed a one-to-one correspondence between predicate symbols p of $\mathcal{O}$ with arity $n \geq 0$ and all pairs $\langle p', n \rangle$, p' a meta-level constant of type 'meta-id'. For any such predicate symbol p, we let $\lceil p \rceil = \text{predicate}(p', n)$.
- There is assumed a one-to-one correspondence between function symbols f of $\mathcal{O}$ with arity $n \geq 1$ and all pairs $\langle f', n \rangle$, f' a meta-level constant of type 'meta-id'. For any such predicate symbol f, we let $\lceil f \rceil = \text{function}(f', n)$.
- For any other phrase P of $\mathcal{O}$, $\lceil P \rceil$ is given inductively using the defining patterns in table 1.

□

In our examples, we relax the notation a little leaving out the "′" for predicates, functions, and constants and we let the actual category of an object language phrase appear from the context. So we may write, e.g., $\lceil \text{a} \rceil = \text{constant(a)}$. For an object variable, say X, we use the Prolog like notation 'X' for X′, so we can write $\lceil \text{X} \rceil = \text{constant('X')}$

The actual choice of object language syntax and naming relation influence the degree of complications in the derivation system. We have chosen here to associate explicit arities with predicate and function symbols and thus the derivation system must ensure the consistency between arities and lengths of argument lists. We could have taken a step further, having specific polymorphic types associated with each predicate and function symbol. This again

would give the derivation system more information to keep track of. Another extreme would be to choose a Prolog-like syntax with overlapping sets of function and predicate symbols with no assigned arities. We believe, however, that the example we have chosen is fairly representative as to indicate the general principles.

The following definition gives a representation of object level substitutions as meta-level terms. For convenience we use the same symbol as for the naming relation.

Definition 4.3 A *substitution term* is a term of $\mathcal{M}$ of the form

$$t = [(id_1, t_1), \ldots, (id_n, t_n)|v], \quad n \geq 0$$

where

- $id_1, \ldots, id_n$ are distinct constants of type meta-id,
- $t_1, \ldots, t_n$ are terms,
- v is either $[\,]$ or a variable in which case the substitution term is *open* and v its *tail*.

For an $\mathcal{O}$-substitution σ, a *name of* σ, denoted $\lceil\sigma\rceil$, is a ground substitution term,

$$[(id_1, t_1), \ldots, (id_n, t_n)]$$

such that

$$\sigma = \bigcup_{i=1\ldots n}\{X_i \mapsto T_i \,|\, \lceil X_i\rceil = \text{variable}(id_i), \lceil T_i\rceil = t_i \neq \lceil X_i\rceil\}$$

□

The notation $\lceil\sigma\rceil$ is not unique and should be used with care. This usage of $\lceil -\rceil$ is not part of the naming relation as object level substitutions are not part of the object language.

4.2 A low-level meta-language

Before we define the new meta-language CLP(meta$'(\mathcal{O})$), we give some examples which motivate the need for another and more detailed language.

In order to handle an occur's problem inherent in instance constraints, we have added an extra argument to syntax constraints, which we will illustrate in the following quite lengthy example.

Example 4.1 A CLP(meta($\mathcal{O}$))-constraint of the form τ-instance(t_1, t_2) indicates that the meta-level terms t_1 and t_2 must conform to a common structure down to a certain level of detail. Consider the following.

formula-instance(Z, ⌈p(a)⌉)

Here, any value for Z which satisfies the constraint must be an instance (at the meta-level) of the term ⌈p(⌊_⌋)⌉ which in the following will be defined as a shadow term for ⌈p(a)⌉. We cannot be more specific in giving such a shadow term as the open component of Z could be replaced by, say, ⌈X⌉, ⌈Y⌉, or ⌈a⌉ (but not by ⌈b⌉).

In general, for a constraint τ-instance(t_1, t_2), the value of t_2 constrains the possible t_1's to those that bear the structure of t_2 down to, but not including, proper subterms of type 'term'. There is, however, on exception to this. Consider the constraint

term-instance(Z, ⌈f(a)⌉).

In case Z is constrained somehow to be the name of a variable, the indicated structural resemblance between the two arguments disappears. If Z is known to represent anything else but a variable, the general principle indicated above applies. In the CLP(meta′($\mathcal{O}$)) language, type constraints carry an additional second argument, which represent the shadow term. The constraint solver algorithm will evaluate these shadow terms, and unify different shadows whenever this is required by instance constraints.

We illustrate the occur's problem by the following two simultaneous constraints.

formula-instance(X, Y), formula-instance(conjunction(X, true), Y)

From the second constraint we see that Y must be an instance of a shadow term

$$P_{\mathrm{Y}} = \text{conjunction}(P_{\mathrm{X}}, \text{true})$$

where P_{X} is a shadow term for X. From the first constraint, we see that $P_{\mathrm{X}} = P_{\mathrm{Y}}$ and thus

$$P_{\mathrm{X}} = \text{conjunction}(P_{\mathrm{X}}, \text{true})$$

which not possible.

The derivation system to be presented in section 4.6 will install initially also the constraints

formula(X, P_X), formula(Y, P_Y).

With a suitable computation rule, the derivation system will detect the offending equation and potential loops are avoided. Example 4.6 illustrates how the derivation system uses shadow terms. □

In order to simplify the reduction of instance constraints, they are extended by an additional argument which represents an object level substitution. We explain this in the following example.

Example 4.2 Consider the CLP(meta($\mathcal{O}$))-constraint

term-instance(⌈f(X,X)⌉,⌈f(a,a)⌉).

It is satisfied because there exists an object level substitution $\{X \mapsto a\}$ which changes f(X,X) into f(a,a). Consider now the following.

term-instance(⌈f(X,⌊Z⌋)⌉, ⌈f(a,⌊Z′⌋)⌉)

It is reasonable to reduce this to the simpler constraint,

term-instance(Z, Z′),

but in doing this, we have lost the information that if some value which may be assigned to Z later has ⌈X⌉ in some position, then the value of Z′ should have ⌈a⌉ in the similar position. We add a new third argument to instance constraints which represents an object substitution recording such commitments. When we repeat the example in the new language CLP(meta′($\mathcal{O}$)), the reduced constraint will be of the form

term-instance(Z, Z′, s),

where s represents a substitution which includes the object level binding $X \mapsto a$.

We use a special version of instance constraints of the form

bind(x, t, s),

where x is a constant which identifies some object variable. In case x is unknown, i.e., a value should be bound to an object variable whose name is not known, we prefer to delay a bind constraint instead of updating a substitution term (which would cause trouble in case two such unknown variables got alialized). □

The last innovations in the constraint repertoire are constraints which keep track of the length of term lists which appear as argument lists of atoms or hold the subterms of structures. These length constraints introduce yet another occur's problem to which the shadow terms also are helpful; we discuss this issue in example 4.4 following the definitions.

The following notion is used in the definition of satisfiability for the extended type constraints.

Definition 4.4 Two ground meta-level terms t_1 and t_2 are *instance compatible* if they have common most specific type τ and there exist $\mathcal{O}$-terms T_1, T_2, and T of category τ such that

- $t_1 = \lceil T_1 \rceil$, $t_2 = \lceil T_2 \rceil$, and
- T_1 and T_2 are both instances of T.

We denote this $t_1 \approx t_2$. □

We notice that $t_1 \approx t_2$ if and only if they are identical down to but not (necessarily) including proper subterms of type term.

Example 4.3 Let p be an $\mathcal{O}$-predicate, f be an $\mathcal{O}$-function, and a, b $\mathcal{O}$-constants. Then $\lceil$p(a)$\rceil \approx \lceil$p(b)$\rceil$ and $\lceil$f(a)$\rceil \approx \lceil$f(b)$\rceil$, but $\lceil$a$\rceil \not\approx \lceil$b$\rceil$. □

Definition 4.5 Let CLP(meta$'(\mathcal{O})$) be a constraint language whose signature includes the signature of CLP(meta($\mathcal{O}$)), however, such that the constraint symbols of CLP(meta($\mathcal{O}$)) appear as normal predicate symbols in CLP(meta$'(\mathcal{O})$). The constraint domain meta$'(\mathcal{O})$ includes the following constraints,

- *type constraints* of the form, for any syntactic category τ of the object language,

  $\tau(p_1, p_2)$

  which is satisfied if and only if p_1 and p_2 are of type τ with $p_1 \approx p_2$,

- *instance constraints* of the form, for $\tau \in$ {formula,term,term-list},

  τ-instance(p_1, p_2, s)

  which is satisfied if and only if $p_1 = \lceil P_1 \rceil$, $p_2 = \lceil P_2 \rceil$ where P_1, P_2 are object phrases of category τ and $s = \lceil \sigma \rceil$, where σ is an object level substitution with $P_1\sigma = P_2$.

- *bind constraints* of the form

bind(id, t, s)

satisfied if and only if there exist object variable V, object term T and object substitution σ with $\lceil V \rceil = \text{variable}(id), \lceil T \rceil = t, \lceil \sigma \rceil = s$ and $V\sigma = T$.

– *inequalities* of the form

$t \geq 0$, resp. $t \geq 1$,

satisfied satisfied if and only if t is of type meta-int and greater than 0, resp. 1.

– *length* constraints of the form

length(t, n)

satisfied if and only if t is a term of type term-list with length n.

□

At the declarative level, there appears to be a symmetry between the two arguments of a type constraint, but they serve different purposes in the derivation system to be presented in section 4.6; see also examples 4.1 and 4.6.

We end this section showing another occur's problem which makes the definitions of normalized constraints below slightly more tricky than one might expect.

Example 4.4 Consider the following set of length constraints; v and n are variables.

$$\{\text{length}(v, n), \text{length}([\lceil \text{a} \rceil \mid v], n)\}$$

Clearly this is unsatisfiable as it indicates that the length of a value of v is this length plus one. In the derivation system of section 4.6, we fix this by unifying the shadow terms for all arguments whose length is given by the same variable.

In this example, this leads to the equation

$$P_v = [\_ \mid P_v].$$

□

4.3 Implementation of CLP(meta($\mathcal{O}$)) in CLP(meta′($\mathcal{O}$))

The CLP(meta($\mathcal{O}$)) language in which the demo program is written is implemented the more detailed CLP(meta′($\mathcal{O}$)) by a program, we will refer to as IMPL, specified as follows.

For each syntactic category τ of the object language, IMPL contains a clause

τ(X):– τ(X, _).

To implement instance constraints, we have the following clauses in IMPL. The clauses are not complete symmetric as there are different collections of types with associated instance constraints in the two languages.

formula-instance(F, F′):– formula-instance(F, F′, _).

term-instance(T, T′):– term-instance(T, T′, _).

atom-instance(A, A′):– atom(A, _), formula-instance(A, A′, _).

clause-instance(⌈⌊H⌋:– ⌊B⌋⌉, ⌈⌊H′⌋:– ⌊B′⌋⌉):–
atom(H, _), formula-instance(H, H′, S),
formula-instance(B, B′, S).

It is easy to verify the following from the definitions of satisfiability for the two constraint systems.

Proposition 4.2 The program IMPL is an implementation of CLP(meta($\mathcal{O}$)) in CLP(meta′($\mathcal{O}$)). □

In summary, to fulfill our original goal of developing a sound and complete implementation of the demo predicate, we need a to develop a suitable derivation system for for CLP(meta′($\mathcal{O}$)).

4.4 Properties of constraint sets

Here we define some properties of constraint sets which we will need in order to formulate the relevant correctness statements. But firstly, we will discuss briefly some factors that influence the complexity of these conditions.

One might get the impression that a typed meta-language would make the formal work simpler, as this would simplify the constraint system. However, a dynamic treatment of subtypes is needed anyhow since it is not known in advance whether a variable, say, of type 'formula' will develop into an atom, a conjunction or *true*. There is also the shadow mechanism which we merged with type constraints and which also needs to be taken care of.

The conditions below reflect also a decision concerned with the exact point in time when type constraints actually lead to the instantiation of variables.

Consider, for example, a type constraint constant(X, P), X, P both variables. Since any possible value for X conforms to the same defining pattern we could safely instantiate X = P = constant(Z) and reduce the constraint to meta-identifier(Z). By experience from our implemented system, we found that a principle of delaying all instantiations as long as possible, provides a better synchronization with user-defined constraints which are supposed to cooperate with demo; we discuss this pragmatic issue in sections 4.8.5 and 5. Our derivation system postpones such instantiations until a variable such as X above gets involved in an instance constraint. The alternative 'faster' instantiation principle indicated above would lead to slightly (but not radically) simpler formulations.

4.4.1 Type constraints and shadow terms

The second argument of a type constraint serves as a shadow giving the overall appearance of the first argument and a shadow can be shared among a classes of similar, but not collection of identical, terms.

Definition 4.6 The set of *shadow terms* for a term t of some type τ is defined inductively as follows.

- a variable is a shadow term for any term.
- if t is not of type 'term' and t is an instance of a defining template $t = \mathbf{t}[t_1, \ldots, t_n]$ and s_i is a shadow term for t_i, $i = 1, \cdots, n$, then $\mathbf{t}[s_1, \ldots, s_n]$ is a shadow term for t.
- no variable appears more than once in a shadow term.

A *most specific* shadow term for t is one which is a copy of t in which each proper subterm of type 'term' is replaced by a unique variable. □

For example, the term $\lceil p(\lfloor X \rfloor) \rceil$ is a common, most specific shadow term for $\lceil p(a) \rceil$, $\lceil p(b) \rceil$, and $\lceil p(f(g(Z))) \rceil$.

We will refer to the following properties of shadow terms.

Proposition 4.3 Let τ and τ' be types, t a term, and p and p' arbitrary shadow terms and consider the following constraint set.

$$C_1 = \{\tau(t, p), \tau'(t, p')\}$$

If $\tau\tau' = \bot$ then $[\![C_1]\!] = \emptyset$; otherwise $[\![C_1]\!] = [\![C_2]\!]$ where

$$C_2 = \{\tau\tau'(t, p), p = p'\}.$$

When p is a most specific shadow term for some term of type τ, then for any terms t, t', the following constraint set,

$$\{\tau(t,p), \tau(t',p)\}$$

is satisfiable if and only if $t \approx t'$. □

Proposition 4.4 Let t_1 and t_2 be ground terms so that t_1 is not the name of an object variable, and p a most specific shadow term for t_2. Then the constraint

$$\tau\text{-instance}(t_1, t_2, s)$$

is satisfied for some s if and only if

- t_1 is an instance of a most specific shadow term of t_2
- t_2 is an instance of $\lceil t_i \rceil^\circ$, and
- $\tau(t_1), \tau(t_2)$ are satisfied.

□

4.4.2 Invariant properties for constraint sets

The following properties will be used in different combinations as invariants in the statements of correctness and termination of the derivation system to be introduced in section 4.6.

Definition 4.7 A constraint set C is *correctly typed* if there exists a substitution μ such that $C\mu$ is ground and

- for any equation $t_1 = t_2 \in C\mu$, t_1 and t_2 have the same type,
- for any $\tau(t_1, t_2) \in C\mu$, t_1 and t_2 have type τ,
- for any τ-instance$(t_1, t_2, s) \in C\mu$, t_1 and t_2 have type τ, s type 'subst',
- for any bind$(t_1, t_2, s) \in C\mu$, t_1 has type 'meta-id', t_2 type 'term', s type 'subst',
- for any length$(t_1, t_2) \in C\mu$, t_1 has type 'term-list', t_2 type 'meta-int', and
- for any $t \geq 0 \in C\mu$ or $t \geq 1 \in C\mu$, t has type 'meta-int'.

When C is correctly typed and μ is as above, we say that a term (or variable) t has *inherent type* τ whenever $t\mu$ is of type τ.

A correctly typed constraint set C is *substitution well-formed* if and only if

- any substitution term in C is of the form

  $$t = [(id_1, t_1), \ldots, (id_n, t_n)|v], \quad n \geq 0$$

  where $id_1, \ldots, id_n$ are distinct constants of type 'meta-id' and the tail v a variable, and

- if a variable v occurs in C as the tail of two substitution terms s and s', none of which is a proper subterm of another term, then s and s' are identical.

A constraint set C is *well-formed* whenever it is correctly typed, substitution well-formed and each second argument of a type constraint is a shadow term. □

The uniqueness condition for tail variables of substitution terms means that each such variable can be understood as a fill-pointer for the construction of a representation of an object level substitution. Notice that the well-formedness condition does not require anything about the consistency of term list lengths.

The following definition includes the notion of safeness which is very important for termination; roughly speaking, this condition says that no variable (apart from those with inherent lexical types) can occur simultaneously in both a first and a second argument of an instance constraint; we remind that the notion of lexical types is given by definition 4.1.

Definition 4.8 For a given well-formed constraint set C it is defined inductively as follows what it means for a variable or an occurrence of a term to be *lexical*, *external*, *internal*, or *shadow*.

- Any occurrence in a term with inherent lexical type is lexical.
- For any constraint $\tau(\ldots, p) \in C$, any non-lexical occurrence in p is shadow.
- For any constraint τ-instance$(e, i, [\cdots(\cdots, i')\cdots]) \in C$
  or bind$(\cdots, i, [\cdots(\cdots, i')\cdots]) \in C$,
  - any non-lexical occurrence in e is external, and
  - any non-lexical occurrence in i, i' is internal.

- Any variable which has a lexical (external, internal, shadow) occurrence is lexical (external, internal, shadow).

- For any $\tau(t,\ldots)$, $t \geq 0$, $t \geq 1$, or $\mathrm{length}(t,\ldots) \in C$ any non-lexical occurrence in t is external (internal, shadow) whenever t includes an external (internal, shadow) variable.

- For any $t_1 = t_2 \in C$ any non-lexical occurrence in t_1 and t_2 is external (internal, shadow) whenever t_1 or t_2 includes an external (internal, shadow) variable.

For any well-formed constraint set C, we define $\mathrm{Lexical}(C)$ (and $\mathrm{External}(C)$, $\mathrm{Internal}(C)$, $\mathrm{Shadow}(C)$) as the sets of lexical (external, internal, shadow) variables in C.

A constraint set C is said to have *protected shadow variables* whenever

- variables of $\mathrm{Shadow}(C)$ occurs only in second arguments of type constraints or in equations, and

- $\mathrm{Shadow}(C) \cap (\mathrm{External}(C) \cup \mathrm{Internal}(C)) = \emptyset$.

If, furthermore,

$$\mathrm{External}(C) \cap \mathrm{Internal}(C) = \emptyset,$$

C is said to be *safe*.

A state $\langle I, C, \alpha\rangle$ has *protected shadow variables* (is *safe*) whenever $I \cup C \cup \alpha$ has protected shadow variables (is safe); the substitution α here viewed as a set of equations $\{x = x\alpha\}$. □

Definition 4.9 A constraint set C is *normalized* if it satisfies the following conditions.

- C is well-formed and safe.

- There are no equations in C.

- For any constraint $\tau(t,p) \in C$,

  - t and p are variables,
  - if τ is lexical then $t = p$,
  - there is no other $\tau'(t,p') \in C$, and
  - if there is another $\tau'(t',p) \in C$, then $\tau' = \tau$.

- For any constraint τ-instance$(t_1, t_2, s) \in C$,
  - t_1 is a variable and $\tau(t_1, p) \in C$ for some variable p,
  - if there is another τ'-instance$(t_1, t_2', s') \in C$, then $\tau' = \tau$ and $s' \neq s$,
  - if $\tau \neq$ 'term', then t_2 is a variable and $\tau(t_1, p), \tau(t_2, p) \in C$ for some variable p,
  - if $\tau =$ 'term-list', then length(t_1, n), length$(t_2, n) \in C$ for some variable n.
- For any constraint bind(t_1, t_2, s),
  - t_1 is a variable and meta-id$(t_1, t_1) \in C$, and
  - if there is another bind$(t_1, t_2', s') \in C$, then $s \neq s'$.
- For any term occurring in C of the form[7]

  structure(function$(f, n), l)$

  the following holds,
  - l is of the form $[t_1, \ldots, t_m \mid t]$ with $m \geq 0$, t a variable or $[\,]$,
  - $t = [\,]$ if and only if n is ground if and only if n is identical to m and $m \geq 1$,
  - if t is a variable so is n and length$(l, n) \in C$.
- For any constraint length$(l, n) \in C$,
  - n is a variable with meta-int$(n, n) \in C$.
  - l is of the form $[t_1, \ldots, t_m \mid v]$ with $m \geq 0$, v a variable with term-list$(v, p) \in C$ for some variable p, and
  - for any other length$(l', n) \in C$, l' is of the form $[t_1', \ldots, t_m' \mid v']$, v' a variable with term-list$(v', p) \in C$.
- For any constraint $t \geq 0$ or $t \geq 1 \in C$, t is a variable with meta-int$(v) \in C$.

[7] The other conditions imply that there are no terms with inherent type 'atom', so we do not have to worry about those. Terms of type 'structure' as indicated may occur in a second argument of a term-instance constraints, in the third argument of any instance or bind constraint, or inside the first argument of a length constraint. We could have required that each variable in these arguments be covered by a type constraint, however the well-formedness condition is sufficient.

A state is *normalized* if it is of the form $\langle \emptyset, C, \alpha \rangle$ with C normalized. A state is *constraint normalized* if it is of the form $\langle I, C, \alpha \rangle$ with I free of constraints and C normalized. □

Lemma 4.1 Any normalized constraint set or normalized state is satisfiable. □

The proof is given in appendix A.

4.5 An informal discussion of instance constraints

We can illustrate the intuition in the definition of safeness by means of colours. By definition, the external occurrences in a constraint set are concerned with the ground representation of the object language; a meta-variable here stands for a piece of object program text. We choose a cool blue colour for it to indicate the fool-proof and secure properties of the ground representation for meta-programming. For the internal parts we choose a warm red colour to indicate the more efficient nonground representation with its real, live variables as reflections of $\mathcal{O}$-variables, cf. proposition 3.3. The shadow parts are associated with white (to indicate that they are not so interesting) and lexical occurrence are black (which remain black if accidentally touched by another colour).

Confronted with a given constraint set we begin painting all lexical occurrences black, so we do not have to consider those. For each instance constraint, we paint the external parts blue, the internal parts red,

$$\tau\text{-instance}(\mathit{blue}, \mathit{red}, [\cdots(\cdots, \mathit{red})\cdots]).$$

For each type constraint, we paint its second component, the shadow part, white,

$$\tau(\ldots, \mathit{white}).$$

We continue now painting the colours blue, red, and white recursively as indicated in definition 4.8. The constraint set is thus safe if no purple or pale colours can be observed.

We are especially interested in the colour patterns which can arise in constraint sets created during the execution of predicates in the DEMO program. First of all, we see that the clauses of the IMPL program excludes the possibility of pale colours, i.e., the shadow variables have their own separate space. Purple colour can be created by unrestricted calls of demo_1 of the form $\text{demo}_1(\cdots \text{X} \cdots, \cdots \text{X} \cdots)$. In other words, if a user can call demo_1 directly, this may lead to unsafe constraint sets.

However, the call of formula-instance in the clause for demo gives a perfect separation between internal and external variables. We can generalize the colouring method to a static analysis of the demo program, and we achieve the colour pattern

demo(*blue*, *blue*)

which means that a demo-user (in a very strict sense) only have access to the ground representation. The colour pattern for the demo_1 predicate (which "does the work") is

demo_1(*blue*, *red*).

The safeness condition separates the spaces of external and internal variables and thus avoids a confusion of meta-variables in the ground and nonground representations. As we will see, safeness is essential for the desired termination properties, section 4.7.

4.6 A derivation system for $\text{meta}'(\mathcal{O})$

We define a derivation system $\mathcal{DS}$ for $\text{CLP}(\text{meta}'(\mathcal{O}))$ consisting of the resolution and unification rules (section 2.2) together with the rules (AT1–2), (RT1–2), (AI1–2), (RI1–2), (AB), (RB), (AL1–2), (RL), (AGe0–1) and (RGe0–1) introduced below. Labels (A$\cdots$) are used for add rules, (R$\cdots$) for reduction rules; the notation used is explained in section 2.2. For an easier overview, we group the rules according to the sort of constraints they select.

Type constraints

(AT1) $\langle I \cup \{\tau(v,p)\}, C \cup \{\tau'(v,p')\}\rangle \rightsquigarrow \langle I \cup \{p = p'\}, C \cup \{\tau\tau'(v,p')\}\rangle$
— when v is variable.
However, if $\tau\tau' = \bot$, the result is FAILURE.

(AT2) $\langle I \cup \{\tau(t,p)\}, C\rangle \rightsquigarrow \langle I \cup I', C \cup \{\tau(t,p)\}\rangle$
— when (AT1) do not apply.
If τ is lexical, $I' = \{t = p\}$, otherwise $I' = \emptyset$.

(RT1) $\langle I, C \cup \{\tau(t,t')\}\rangle \rightsquigarrow \langle I \cup \{t = t'\}, C\rangle$
— when t or t' not a variable, τ is 'meta-constant' or a subtype thereof.
However, if the one of t and t' different from a variable is not of type τ, the result is FAILURE.

(RT2) $\langle I, C \cup \{\tau(t,p)\}\rangle \rightsquigarrow \langle I \cup I', C\rangle$
— when t or p not a variable and (RT1) do not apply.
Let τ^* be a type with defining pattern $\mathbf{t}[\langle\tau_1\rangle,\ldots,\langle\tau_n\rangle]$ such that $\tau \geq \tau^*$ and $\mathbf{t}[\langle\tau_1\rangle,\ldots,\langle\tau_n\rangle]$ has the same top function symbol as t or p, then

- $t = \mathbf{t}[v_1,\ldots,v_n] \in I'$,
- $p = \mathbf{t}[u_1,\ldots,u_n] \in I'$, and
- if τ_i = term, we have $\tau_i(v_i, w_i) \in I'$, otherwise $\tau_i(v_i, u_i) \in I'$ for $i = 1,\ldots,n$,

where $v_1,\ldots,v_n,u_1,\ldots,u_n,w_1,\ldots,w_n$ are new variables.
If τ^* = atom we have, furthermore, $\{\text{length}(v_3, v_2), v_2 \geq 0\} \subseteq I'$.
If τ^* = structure we have, furthermore, $\{\text{length}(v_3, v_2), v_2 \geq 1\} \subseteq I'$.
If τ^* = predicate we have, furthermore, $v_2 \geq 0 \in I'$.
If τ^* = function we have, furthermore, $v_2 \geq 1 \in I'$.
However, if no such τ^* exists, the result is FAILURE.

Instance constraints

(AI1) $\langle I \cup \{\tau\text{-instance}(v,t,s)\}, C \cup \{\tau'\text{-instance}(v,t',s)\}\rangle$
$\rightsquigarrow \langle I \cup \{t = t'\}, C \cup \{\tau'\text{-instance}(v,t',s)\}\rangle$
However, if $\tau \neq \tau'$, the result is FAILURE.

(AI2) $\langle I \cup \{\tau\text{-instance}(t_1,t_2,s)\}, C\rangle \rightsquigarrow \langle I \cup I', C \cup \{\tau\text{-instance}(t_1,t_2,s)\}\rangle$
— when (AI1) do not apply.
We have

$$\{\tau(t_1,p_1), \tau(t_2,p_2), \text{term}(r_1,p_{r1}),\ldots,\text{term}(r_n,p_{rn})\} \subseteq I'$$
$$\text{where } s = [(id_1,r_1),\cdots,(id_n,r_n)|v], \quad n > 0,$$

with $p_1, p_2, p_{r1},\ldots,p_{rn}$ new variables.
If $\tau \neq$ term we have, furthermore, $p_1 = p_2 \in I'$.

Comment: Most of the new type constraints created in (AI2) are unnecessary, but we add them here in order to simplify the proofs.

(RI1) $\langle I, C \cup \{\text{term-instance}(t_1,t_2,s)\}\rangle \rightsquigarrow$
$\langle I \cup \{\text{bind}(id,t_2,s), t_1 = \text{variable}(id)\}, C\rangle$
— when t_1 of the form variable$(\cdots)$
or t_1 a variable with variable$(t_1,p) \in C$.
id is a new variable.

(RI2) $\langle I, C \cup \{\tau\text{-instance}(t_1, t_2, s)\}\rangle \rightsquigarrow \langle I \cup I', C\rangle$
— when (RI1) do not apply, t_1 not a variable or $\tau'(t_1, p_1) \in C$, $\tau \neq \tau'$.
Let τ^* be a type with defining pattern $\mathbf{t}[\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle]$ such that $\tau > \tau^*$ and either $\mathbf{t}[\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle]$ has the same top function symbol as t or $\tau^*(t_1, p_1) \in C$. Then

- $t_1 = \mathbf{t}[v_1, \ldots, v_n] \in I'$,
- $t_2 = \mathbf{t}[u_1, \ldots, u_n] \in I'$,
- τ_i-instance$(v_i, u_i, s) \in I'$ for those $i = 1, \ldots, n$ with $\tau_i \in \{$formula, term, term-list$\}$, and
- $\{v_i = u_i, \tau_i(v_i, p_i)\} \in I'$ for those $i = 1, \ldots, n$ with τ_i lexical, p_i is a new variable,

where $v_1, \ldots, v_n, u_1, \ldots, u_n$ are new variables.
In case τ^* = structure or τ^* = constant, we have additionally

$\tau^*(t_1, p), \tau^*(t_2, p) \in I'$, p a new variable.

p a new variable.
However, if no such τ^* exists, the result is FAILURE.

Comment: The purpose of the two new τ^*-type constraints is to unify the shadow terms for the two arguments t_1 and t_2. When $\tau \neq$ term this has already been done by rule (AI2), but in case of terms we must unify the patterns when and only when t_1 is known not to stand for an object variable.

Bind constraints

(AB) $\langle I \cup \{\text{bind}(id, t, s)\}, C\rangle$
$\rightsquigarrow \langle I \cup \{\text{meta-identifier}(id), \text{term}(t)\}, C \cup \{\text{bind}(id, t, s)\}\rangle$.

(RB) $\langle I, C \cup \{\text{bind}(id, t, s)\}\rangle \rightsquigarrow \langle I \cup I', C\rangle$
— when id is not a variable and either

- $s = [\cdots(id, t')\cdots]$ in which case $t = t' \in I'$, or otherwise
- $s = [\cdots|v]$ in which case $v = [(id, t)|v'] \in I'$, v' a new variable.

Length constraints

(AL1) $\langle I \cup \{\text{lenght}(t, n)\}, C \cup \{\text{lenght}(t', n)\}\rangle$
$\rightsquigarrow \langle I \cup \{\text{term-list}(t, p), \text{term-list}(t', p)\}, C \cup \{\text{lenght}(t', n)\}\rangle$
where p is a new variable.

Comment: Notice the identification of patterns for t and t'; we do not need to accumulate lenght(t, n).

(AL2) $\langle I \cup \{\text{lenght}(t,n)\}, C\rangle$
$\rightsquigarrow \langle I \cup \{\text{meta-integer}(n, p_n), \text{term-list}(t,p)\}, C \cup \{\text{lenght}(t,n)\}\rangle$
— when (AL1) do not apply; p_n, p are new variables.

(RL) $\langle I, C \cup \{\text{lenght}(t,n)\}\rangle \rightsquigarrow \langle I \cup I', C\rangle$,
— when either

- t is sufficiently instantiated to tell it different from a term list or n is sufficiently instantiated to tell it different from an integer ≥ 0, in this case the result is FAILURE, or otherwise
- n is an integer $m \geq 0$, or t of one of the forms

  $[t_1, \ldots, t_m]$, or
  $[t_1, \ldots, t_m \mid v]$, v a variable with empty-term-list$(v) \in C$.

  In these cases

  $$\{t = p, n = m\} \subseteq I'$$

  where

  $$p = [v_1, \ldots, v_m],$$

  $v_1, \ldots, v_{n'}$ new variables.

Inequalities

(AGe0) $\langle I \cup \{t \geq 0\}, C\rangle \rightsquigarrow \langle I \cup \{\text{meta-integer}(t,p)\}, C \cup \{t \geq 0\}\rangle$
where p is a new variables.

(RGe0) $\langle I, C \cup \{t \geq 0\}\rangle \rightsquigarrow \langle I, C\rangle$
— when t is nonground.
However, if t is not of type meta-integer greater than or equal to 0, the result is FAILURE.

(AGe1) and (RGe1) — analogous.

In the following example, we illustrate how the algorithm incorporates the inherent non-ground representation implied by the very nature of instance constraints as indicated by proposition 3.3.

Example 4.5 An input constraint,

formula-instance(⌈p(a, X)⌉, Y, S),

reduces via repeated applications of the rule (RI2) to a state in which the variable Y has been unified with

⌈p(a, X)⌉° = ⌈p(a, ⌊Z⌋)⌉ for some meta-variable Z,

and S has been unified with the following substitution term.

[('X', Z) | _]

The object level variable X has replaced or reflected by a meta-variable Z and the substitution term records which object variables has been reflected by which meta-variables.

Consider, now, another input constraint where X is a meta-variable.

formula-instance(⌈p(a, ⌊X⌋)⌉, Y, S)

It reduces via repeated applications of the rule (RI2) to a state in which the variable Y has been unified with

⌈p(a, _)⌉° = ⌈p(a, ⌊Z⌋)⌉ for some meta-variable Z,

whereas S here is unbound. This state include among others the following constraint,

term-instance(X, Z, S).

So the same meta-level term ⌈p(a, ⌊Z⌋)⌉ is constructed for ⌈p(a, X)⌉ as well as for ⌈p(a, ⌊X⌋)⌉, but the commitments on the meta-variable Z are different. □

We end this section by an example which illustrates the information flow in the shadow arguments.

Example 4.6 In earlier examples, 4.1 and 4.4, we showed the syntax constraints' shadow argument used for handling some subtle occurs' problems. In general, the shadows are used by the derivation system $\mathcal{DS}$ for communicating syntactic information between constraints that depend on each other. Consider a state which includes the following constraints, where X, Y, Z, and V are variables.

formula-instance(X, Y, s_1), formula-instance(X, Z, s_2),
formula-instance(V, Y, s_3).

When these constraints are processed successively by the add rule (AI2), the following additional syntax constraints will be created,

formula(X, P), formula(Y, P), formula(Z, P), formula(V, P).

The variable P stands for the common pattern which all possible values for X, Y, Z, and V has to conform to. So if, say, V is instantiated to ⌈p(A),⌊W⌋⌉, (RT1) will instantiate P to a pattern ⌈p(⌊P_1⌋),⌊P_2⌋⌉ so that (RI2) and (RT2) will be applicable to all three instance constraints and the remaining type constraints. □

4.7 Correctness results

We notice the following invariant properties for the derivation system $\mathcal{DS}$.

Proposition 4.5 Let $S \rightsquigarrow S'$ be an arbitrary derivation step using a rule of $\mathcal{DS}$.

- If S is safe, so is S'.
- If $S \rightsquigarrow S'$ is not a resolution step, the set of substitutions $[\![S']\!]$, each restricted to the variables of S, coincides with $[\![S]\!]$.

□

The proof will appear in a forthcoming publication.

The following restrictions on the computation rule are necessary in order to provide interesting termination behaviour.

Definition 4.10 A computation rule is *fast-unifying* if the unification rule is applied whenever possible before any other rule.

A computation rule is *fast-solving* if it allows the resolution rule only when no other rule can be applied.

When a fast-solving computation rule is used, any maximal subderivation without resolution steps is called a *constraint solver derivation*. □

As we observed in examples 4.1 and 4.4, unification of certain shadow terms are needed in order to detect some occur's problems which otherwise may cause the derivation to loop; a fast-unifying computation rule takes care of this. The fast-solving property ensures that the constraints in a clause is solved before the next resolution step takes place. In this way, we prevent bad (object as well as meta-) programs destroy the possible termination of constraint solving.

Proposition 4.6 Consider derivation with $\mathcal{DS}$ using a fast-unifying and fast-solving computation rule and consider any constraint solver derivation D

starting from a state S. Then, if S is safe, D ends with a normalized state or FAILURE. □

The proof will appear in a forthcoming publication.
An analysis of the dataflow in the DEMO and IMPL programs yields the following. A formal proof can be established on the basis of the ideas expressed in section 4.5 together with an observation that no derivation rule of $\mathcal{DS}$ mixes up the different classes of internal, external and shadow variables.

Proposition 4.7 Consider derivation with $\mathcal{DS}$ using a fast-unifying and fast-solving computation rule where resolution is with respect to the program DEMO ∪ IMPL. Then we have the following.

- Any state in a derivation starting with a state $\langle \text{demo}(p,q), \emptyset, \emptyset \rangle$ is safe.
- Any state in a derivation starting with a state $\langle \text{demo}_1(p,q), \emptyset, \emptyset \rangle$ is has protected shadow variables.
- There exists states $\langle \text{demo}_1(p,q), \emptyset, \emptyset \rangle$ from which derivation leads to unsafe states.

□

The important difference between calls $\text{demo}(p,q)$ and $\text{demo}_1(p,q)$ is the call of an instance constraint on q in the clause for demo. It 'neutralizes' any meta-variable in q by forming a new copy q' with new variables, cf. proposition 3.3.

Combining the propositions 4.5, 4.6, and 4.7 we get finally the desired result.

Theorem 4.1 (Soundness and completeness of demo)
Consider derivation with $\mathcal{DS}$ using a fast-unifying and fast-solving computation rule where resolution is with respect to the program DEMO ∪ IMPL. Then any computed answer for a query $\text{demo}(p,q)$ is a correct answer and vice versa.
□

Analogously to a normal proof system for definite clauses, any desired satisfier can then be found by an algorithm which combine the indicated computation rule with a breadth-first management for the alternative resolution steps. However, as we will see in our examples, the demo predicate can still produce interesting results when traditional depth-first combined with co-routining is used.

The correctness and termination results stated above are concerned with safe constraints only. The termination behaviour of $\mathcal{DS}$ in case of nonsafe con-

straint sets is not known and in general, we have no satisfactory results concerned with the decidability of the constraint satisfaction problem. In the following, we discuss the problem.

We can easily show that unsafe constraint sets can express semi-unification problems.

Definition 4.11 Let S_1, T_1, S_2, and T_2 be arbitrary terms and consider the following problem.

Do there exist XXX substitutions σ, θ_1, and θ_2 such that

$$S_1\sigma\theta_1 = T_1\sigma \text{ and } S_2\sigma\theta_2 = T_2\sigma?$$

In case XXX stands for 'idempotent' ('general') this is called an *idempotent (a general) semi-unification problem.* □

(By general substitution, we mean any mapping from a set of variables to terms; composition defined as usual).

Proposition 4.8 The problem, for given meta-level terms s_1, t_1, s_2, and t_2, whether the following constraint set of meta($\mathcal{O}$),

$$\{\text{term-instance}(s_1, t_1), \text{term-instance}(s_2, t_2)\}$$

is satisfiable, is equivialent with an idempotent semiunification problem. □

The proof is given in the appendix.

The *general* semiunification problem has been shown to be undecidable by (Kfoury, Tiuryn, Urcyczyn, 1990). However, the semantics of our languages is formulated in terms of idempotent substitutions so this result does not help to our case. The decidability properties of idempotent semiunification is not known at present.[8] Clearly, the change from general to idempotent substitutions changes the problem in the sense that the set of possible solutions shrinks drastically, but whether this essentially changes the complexity is an open question. It has been shown by (Henglein, 1994), that a version of semiunification is undecidable, in which the substitutions sought are idempotent but their general composition not necessarily idempotent.

Finally, we mention that general semiunification has been shown to be semidecidable in the sense that there exist algorithms that terminate when a solution is known to exist (Henglein, 1989, Ließ, 1990, Stärk, 1992).

[8] We have earlier published papers about a related constraint solving problem (Christiansen, 1992a, b) where we claimed that solvability in general is undecidable. The arguments here are not correct as we erroneously applied the result for general semiunification in a context of idempotent substitutions. The paper by (Ko, Nadel, 1991) points out a number of similar mistakes.

4.8 Embedding the derivation system in Prolog

The derivation system $\mathcal{DS}$, or rather an approximation of it, has been implemented in Sicstus Prolog (Sicstus, 1994). Each predicate and constraint of CLP(meta($\mathcal{O}$)) and CLP(meta$'$($\mathcal{O}$)) is reflected as a Prolog predicate and the behaviour of constraint solving is obtained by the co-routine mechanisms in this Prolog dialect. Accumulation and (perhaps later) reduction of a constraint is handled by facilities in Sicstus Prolog which make it possible to delay a call of a predicate until the point in time when its arguments via other instantiations fulfill certain conditions. We use especially the notion of a call being *frozen* on a given variable `X` in the sense that it delays until `X` becomes instantiated.

In order to ensure satisfiability of the whole collection of constraint, we use the primitive `frozen(X, ···)` which gives a list of those constraint that are already frozen on `X`. In this way, we can determine whether a new input constraint should delay (i.e., be added to the accumulated constraints) or fail.

In the following we point out the most important aspects for which the correspondence between the rules of $\mathcal{DS}$ and the actual Prolog code is not obvious. We give some informal statements about efficiency and a discussion of the interaction with user-defined constraints. The actual code can be found at the ftp address mentioned in the introduction of this report.

4.8.1 Resolution and unification

There is no explicit interpreter which executes the resolution step. Programs of CLP(meta($\mathcal{O}$)) (such as DEMO) and of CLP(meta$'$($\mathcal{O}$)) (such as IMPL) are written directly as Prolog clauses so Prolog's resolution procedure is in force.

The unification rule has no explicit representation either, it is done implicitly by Prolog within each resolution step. Any equation needed by the add and reduce rules are executed directly by the Prolog primitive "`=`".

This seems optimal with respect to efficiency as no intermediate interpretation level is involved, but it affects the overall correctness properties of the implemented demo predicate.

- We loose completeness (as is the case in Prolog), some answers will not be found when the program loops.
- We loose soundness as Prolog's unification does not support the mechanism in our $\rightsquigarrow_{\text{Unify}}$ rule, which handles the alias problem. Occasionally, the system may answer with constraint sets that are not satisfiable.

– There is no occurs' check, neither for the general unification step nor for handling the special phenomena which we pointed out in examples 4.1 and 4.4.

However, as we can see in our programming examples, section 5, user-defined constraints provide an effective way to achieve an interesting behaviour despite these deficiencies.

Should any application turn up, for which these factors really are important, it is anyhow a routine programming task to implement a breath-first procedure and a correct unification. Such a system would of course be significantly slower than the one we have implemented — and we believe that significantly slower would mean prohibitively slow.

Type constraints

The different type constraints have been merged into one predicate,

`type(` *type*`,` *sub-type*`,` t`,` p`)`

such that, for example,

– term(t, p) appears as `type( term, _, `t`, `p`)`, and

– constant(t, p) appears as `type( term, constant, `t`, `p`)`.

This makes possible an efficient implementation of rule (AT1) by simply unifying type, resp., subtype arguments for a new input and an 'old' accumulated type constraint concerning the same variable.

4.8.2 Instance constraints

The different instance constraints have been merged into one predicate in the same manner as type constraints above. We have put an effort into reducing the number of additional constraints produced by (AI2). Most of these constraints can be avoided as they can be statically determined to be present already.

Let us consider a constraint τ-instance(t_1, t_2, s). The constraints concerned with the details of the substitution term s can all be removed; this follows from the protection of substitution terms inherent in the IMPL program. For t_1, we call only a type constraint when t_1 is a variable; when t_1 is not a variable, the repeated applications of (RI2) imply the same syntactic decomposition as (RT2). For t_2 we follow the same pattern. However, these optimizations are based on a flow-analysis, which we have not formalized.

4.8.3 Length constraints

The conditions in the rule (RL) for when a length constraint should reduce includes a condition saying that the list argument should be sufficiently instantiated, either to tell it different from list or recognizable as a list of a determinate length. Here we have an internal predicate which 'guards' the tail of the growing list and executes the actions of (RL) when the list comes to an end.[9]

4.8.4 Efficiency

We have not made a formal analysis of the time consumption, but we can give some statements about the order of magnitude. A given constraint implies a of number recursive calls of itself which is proportional to the size of its arguments; the safeness condition ensures this. Furthermore, the add rules include mechanisms to keep the number of identical calls occurring in the execution history at a minimum.

However, there are some internal operations in the constraints whose net time consumption may be more than a linear relative to the size of the arguments. We can mention the following.

- Before a type, instance, or length constraint can be frozen on a given variable, we have to go through a list of all those calls frozen on that variable. And this list can become very long; in the case of a meta-variable which stand for part of a clause, there may be one delayed instance constraint for each nested call of demo_1.

- Each time a bind constraint is reduced, a search in a substitution term will take place. The number of entries in substitutions terms will correspond to the number of variables in the object clauses — and in a context in which object programs are not necessarily produced by humans, clauses can be of arbitrary size.

Programmed in Prolog, these operations cannot be of constant time, but using an external bookkeeping machinery programmed with hashing and similar techniques (using, e.g., the C language interface of Sicstus Prolog), constant time can be achieved.

The demo program can be optimized by a pre-unification step, such that information about the selected atom is used to guide the member predicate.

[9] Reducing a length constraint into a new length constraint for the tail of a list would require a machinery for handling integer constraints such as $n_2 = n_1 - 1$, a good reason for using the the method we propose.

In this way, any clause with an obviously wrong predicate symbol will never be selected, it is filtered out before the potentially expensive instance-constraint is called.

```
demo1(P, atom(Pred, ArglistA)):–
    member( clause(atom(Pred, ArglistH), B), P),
    clause-instance( clause(atom(Pred, ArglistH), B),
                     clause(atom(Pred, ArglistA), B1)),
    demo1(P, B1).
```

In this way we achieve an efficiency which is only a constant factor slower than the underlying Prolog — disregarding the indexing on one or more arguments made by some Prolog systems when searching for a clause. However, the whole is a little superficial as the demo predicate is intended for a much larger class of problems than a Prolog interpreter.

4.8.5 Synchronization with user-defined constraints

Typically a user-defined condition is needed in order to have demo to produce interesting answers; it is difficult to imagine an application for which the user will be happy with any program which makes something provable. In this discussion, we assume the user can formalize by a meta-level predicate interesting(−) what it means for a program to be interesting with respect to the given application. In section 5 we will see examples of such conditions.

The predicate interesting(−) may define what sort of atoms may occur, overall conditions on the recurrence for variables inside each clause, the number of atoms in it etc., and overall requirements to the collection of clauses constituting the program. Except from very trivial cases, we can expect that interesting(−) points out a large subset of all programs. We can expect that the query

```
interesting(P), demo(P, ···)
```

as well as

```
demo(P, ···), interesting(P)
```

would both be bad examples of the generate-and-test method, if interesting(−) is programmed in plain Prolog without any use of delay mechanisms.

Of course, interesting(−) should use co-routines construct in order to execute as 'lazy' as possible. This can be achieved, e.g., by setting up block declarations (see Sicstus, 1994) on all (or most) arguments to interesting(−) and any auxiliary predicate it may call. So whenever demo's internal actions

instantiate a meta-variable which is part of the program sought, this event will immediately trigger some action implied by the interesting(−) predicate.

In order to have the interesting(−) predicate make its commitments as late as possible, we should thus have the constraints called by demo to instantiate their arguments as late as possible. This is the reason why we decided that the reduction rules in $\mathcal{DS}$ should not instantiate the arguments of a type constraint corresponding to a subtype with the defining pattern. So, e.g., conjunction(X, P) delays immediately and do not expand its argument before X (or, indirectly, P) gets involved in an instance constraint.

We see, thus, that such user-defined predicates may serve as a sort of constraints which interacts with the constraints provided by the system. However, there is no facility that prevents the user from writing meta-programs which lead to the accumulation of unsatisfiable constraints.

We have found that this methodology, despite the deficiencies of an implementation in Prolog we pointed out in section 4.8, makes demo a powerful tool for alternative reasoning methods or program synthesis task, cf. the examples below, section 5, and the growing collection of examples in the ftp package which contains the system; the ftp address appears in the introduction of this report.

5 A running system and examples of applications

Here we give a brief sketch of an implemented system together with examples of applications which illustrate how user-defined predicates can co-operate with demo. The examples are here restricted to abduction with integrity constraints and under linear-logic-like conditions. These examples are available in the ftp package mentioned, which furthermore contains examples of inductive reasoning and diagnosis. We may also refer to (Christiansen, 1993) where we have suggested the demo predicate be used for automatic construction of grammar and encyclopedia rules in natural language processing.

5.1 Overview of facilities in the system

The system is described in details in the documentation material in the ftp package mentioned in the introduction of this report; here we give only a very brief sketch. It includes an implementation of the demo predicate according to the principles described in this report. One and only one little change in the overall behaviour of demo has been necessary in order limit unwanted backtracking.

Whenever demo needs to expand a meta-variable in the position of a program tail into a structure with one new clause and yet a new tail, it will not try to expand this new tail again on backtracking.

Without this restriction, demo tends to loop too often. The change in declarative semantics is minimal, it will only inhibit answers in case user-defined conditions depend on the actual order of the clauses. Actually the restriction can be removed, it just implies that the user conditions become responsible for preventing these potential loops.

The meta-language CLP(meta($\mathcal{O}$)) is available to the user, who might wish to use the constraints in additional predicates to support demo — or perhaps to write another demo predicate with, e.g., a more intelligent search strategy.

The system uses an extended notation for naming relations such that object language phrases are written in a Prolog-like syntax. The inherent ambiguity is resolved using three different operators corresponding to $\lceil - \rceil$, the prefix operator '\' for names of programs and clauses, '\\' for names of formulas, and '\\\' for names of terms. The reverse operator $\lfloor - \rfloor$ is written uniformly as prefix '?'. The following are examples of names of a program with three clauses, a clause, an atom, and a term.

```
\ [ (p(X):- q(X)), p(a), q(b) ].
\ ( p(X):- q(X) )
\\ p(a, X)
\\\ f(a,X)
```

This notation can be applied in user programs and queries and is also used for printing out answers. In fact, we have also used it for the implementation in Prolog of the derivation system.

Of the system's utilities we can mention a predicate

`close_constraints(`t`)`

which uses a strategy inspired by the constructive proof of lemma 4.1 to instantiate the meta-variables in t in a way which will satisfy the constraints accumulated for these variables; this is very useful in order to obtain readable answers from final states otherwise dominated by a large number of accumulated constraints. We will see it used below in section 5.3.

Finally we will mention a module concept for object programs. Here is a little example

```
:- object_module( pq_program,

     \[ (p(X):- q(X)),
```

```
    p(a),
    p(b)
]).
```

We can now write `\pq_program` as an argument to demo and also combine it with other clauses or meta-variables to form other, extended programs, e.g., as follows

```
\ ( [(q(Y):- r(Y))] & pq_program & ?X ).
```

When the `object_module` directive is executed, each object language clause is processed once and for all by an instance constraint and represented internally in the resulting non-ground form.[10] The demo_1 predicate is thus refined as to bypass instance constraints when clauses are fetched from such modules.

5.2 Combining demo with other meta-level predicates

In section 4.8.5 we discussed principles for synchronizing user-defined predicates with the constraints called by the demo predicate. Here we discuss the practical question about how such user-defined predicates can used for generating interesting programs together with demo; we will use simple abduction (Kakas et al., 1993) to illustrate.

The task is to find an extension of a given theory T with one or more facts so that an observation G can be explained. However, only certain 'primitive' or 'basic' hypotheses are allowed. Assume this be implemented (using delays as explained) as a predicate `abducible(X)` which is satisfied whenever `X` is a program of such facts. The abduction problem can now be expressed as follows in a query to the system.

`abducible(X), demo(`$\lceil T \rceil$ `& X,` $\lceil G \rceil$`)`

The execution behaviour will be optimal in the sense that demo proceeds as usual only assisted by `abducible(`$\cdots$`)` at the exact points where it needs to consult `X`.

As discussed earlier, this methodology contains no protection against unsound answer constraint sets, as the user can delay any strange conditions of his own, but in many cases, the `close_constraints(`$\cdots$`)` mentioned above can be used to detect and report this.

5.3 Example: Abduction with integrity constraints

Here we demonstrate the use of non-trivial integrity constraints as side-conditions.

[10] We relate this to partial evaluation in section 5.5.

Let T be a program describing a number of objects together with some of their properties, e.g., `thing(the_flower)`, `thing(the_vase)`, `thing(the_table)`, `container(the_vase)`. We assume T be defined as an object program module with the name `things`. An actual scene is described by 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 the following object program module which describes sentences that are either simple or folded.

```
:- 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'. We consider the problem of abducing descriptions of the scene from sentences about it. Any such description must satisfy some integrity constraints with respect to T; an 'in' fact, for example, must satisfy the following.[11]

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

The `dif` condition serves, together with other conditions, to preserve a sensible, physical interpretation of the programs generated. We can write a

[11] There is a convention in the system that any constraint in meta($\mathcal{O}$) appears with a terminating underscore in its name, e.g., `constant_` in order to keep the names separate from a few Prolog built-ins.

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.

When a certain T theory is given (as an object program module `things`), the abduction problem of identifying the possible scenes behind a specific sentence can be stated as follows.

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

The `close_constraints` predicate is explained in section 5.1 above. We get 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)]
```

An exchange of the argument of `sentence` with

```
[the_flower, is, in, the_table]
```

yields failure as `the_table` is not a `container`. We can extend the example by also abducing the 'things' theory T in parallel with the scene description; here a fact `container(`c`)` should only be abducible when

`demo(`$\lceil T \rceil$`, \\ thing(`c`))`

succeeds. In this example, the integrity constraint activated for a potential abducible (e.g., the `container(`c`)` above), calls demo which in order to succeed may cause the abduction of yet other facts (e.g., `thing(`c`)`).

5.4 Example: Abduction under linear-logic-like conditions

Another way to assist demo is provided by adding a third argument giving the proof which we will define as a list of (names of) the clauses which are used. We will consider the use of a predicate `no_duplicates(-)` which accepts exactly lists of distinct elements. It can be programmed elegantly and declaratively using delays and a lazy `dif(-,-)` predicate as provided by Sicstus Prolog (Sicstus, 1994). With this we can have demo behave in the style of linear logic (Girard, 1987) as follows.

```
demoL(P,Q):- no_duplicates(Proof), demo(P, Q, Proof).
```

We can use this principle for abductive problems in which each rule is viewed as a resource. Let, e.g., `abducible(-)` accept programs of facts of the sort `drink(tuborg)`, `drink(another_tuborg)`, etc. The following query,

```
abducible_program(How),
demoL( \([(drunk:- drink(_), drink(_), drink(_))] & ?How),
       \\drunk)
```

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

Hodas and Miller (1994) defines a programming language, Lolli, based on linear logic and (Cervesato, 1994) gives a vanilla-like meta-interpreter for it. Combining the structure of this meta-interpreter with our constraints, it seems possible achieve a complete, program-generating demo predicate also for Lolli.

5.5 Possible extensions of the demo system

The optimization of object program modules, section 5.1, by pre-processing their clauses by instance constraints to obtain a non-ground representation, is a special case of partial evaluation. Partial evaluation of a (non-constraint-based) meta-interpreter similar to the demo program has been studied in detail by (Gallagher, 1991, 1993) and (Hill, Gallagher, 1994). Object clauses can be compiled further into new, specialized demo_1 clauses. In case of an object program specified as a number of known clauses and an open tail we can extend this technique by having these new clauses run together with the general demo_1 clauses. Another interesting application of partial evaluation would be to optimize the DEMO program automatically for particular sorts of object clauses, e.g., those that can be recognized as grammar rules.

The inherent use of a nonground representation makes possible a communication from the simulated object level to Prolog by extending the object language with a suitable "escape" construct. — It appears to be quite useful not having to redefine all of Prolog utilities just because you are using a meta-interpreter.

We can also refer to the work by (Brogi et al, 1990) who use a version of the demo predicate which synthesizes program expression containing names of predefined object language modules by means of operators such as union and intersection. It may be useful to combine this with our ultra-fine grained manner of synthesis.

Finally a comment about negation. As in traditional logic programming, there can be a need for negation in clauses and queries, however, in logic programming implies a high complexity which is difficult to control. And this

increases drastically when multiplied with the additional complexity in a complete demo predicate which essentially approximates higher-order phenomena. A delaying `dif(_,_)` predicate as provided by Sicstus Prolog (Sicstus, 1994) has turned out to interact well with our constraints and can be used for implementing constructive negation (Chan, 1988; Stuckey, 1991) for some special cases. However, we cannot announce a general mechanism.

We may notice that for many applications of abduction and induction with positive and negative conditions one can easily simulate negation by adding an extra argument which takes the value 'yes' or 'no', such that p(X) is encoded as p(X, yes), and ¬p(X) as p(X, no).

A Selected proofs

Proof 1 (Lemma 4.1) Let C be a normalized constraint set. We will construct a satisfier μ for C in the following steps.

1. Consider any constraint in C of the form

   formula-instance(v_1, v_2, s).

   Here v_1 and v_2 are variables with formula(v_1, p) and formula$(v_2, p) \in C$, p a variable, and v_1, v_2 do not occur elsewhere in C.
   Let then $v_1\mu = v_2\mu = p\mu = \lceil true \rceil$.
   This satisfies any such mentioned instance and type constraints, which we thus can ignore in the following.

2. Consider any constraint in C of the form

   term-instance(v, t, s).

   The variable v may occur elsewhere in instance constraints

   term-instance(v, t', s')

   and in one syntax constraint term(v, p), p a variable, and perhaps also in a few other positions we consider below.
   For any such variable v, let $v\mu =$ variable(id_v), where id_v is a unique constant of type meta-identifier which do not occur elsewhere; for any such p do the same thing. This will satisfy the mentioned 'term' constraints. The mentioned instance constraints will be satisfied under any further specified μ which do not violate type constraints implied for variables in t, s (and any such t', s') and for which $s\mu$ contains the pair $(id_v, t\mu)$ and correspondingly $s'\mu$ contains $(id_v, t'\mu)$ for any such t', s'. We can thus ignore the mentioned 'term' and instance constraints in the following.

   The mentioned variable v may, furthermore, occur in length constraints, which we consider below.

3. Any constraint bind(v, t, s) is treated in a similar way and thus ignored in the following.

4. Consider the set of all length constraints in C. Each such constraint is of the form,

   length$([t^i_1, \ldots, t^i_{m^i} \mid v^i], n^i)$,

where term-list$(v^i, p^i) \in C$ and perhaps $n^i \geq 0 \in C$ or $n^i \geq 1 \in C$.
For all i, let $v^i\mu =$ [constant(a)], $n^i\mu = m^i + 1$ and $p^i\mu =$ variable(id) for some constant id of type 'meta-id'.
The mentioned length, type and inequality constraints will be satisfiable under any further specified μ which do not violate type constraints implied for variables in t^i_k for all i and k. Notice that the argument also is valid for those i, j with $n^i = n^j$; in such a case the definition of normalized gives $m^i = m^j$ and that v^i and v^j has the same associated shadow variable. (The decisions made about μ for variables in any t^i_k in step 2 do not give troubles).
We can thus ignore the mentioned length, type and inequality constraints in the following.

5. Consider any constraint in C of the form

   term-list-instance(v_1, v_2, s).

   Here v_1 and v_2 are variables with term-list(v_1, p) and term-list$(v_2, p) \in C$, p a variable, and v_1, v_2 do not occur elsewhere in C (apart from positions which we have allowed ourselves to ignore by the previous arguments).
   Let here $v_1\mu = v_2\mu =$ [constant(a)], and $p\mu =$ variable(id) for some constant id of type 'meta-id' (perhaps part of this decision is already in effect after step 4).
   This satisfies the mentioned instance and 'term-list' constraints, which we thus can ignore in the following.

   The variables v_1 and v_2 may occur in other instance constraints as well, but the bindings made do not affect the satisfiability of these instance constraints.

6. For any remaining inequality

   $v \geq 0$ or $v \geq 1$,

   let $v\mu = 666$. This will satisfy the mentioned inequalities together with the implied meta-int(v, v) constraints, which we thus can ignore in in the following.

7. The only remaining constraints are syntax constraints

   $\tau(v, p)$.

   The variable v does not appear in C (apart from positions which we have allowed ourselves to ignore by the previous arguments) and p only

elsewhere in constraints $\tau(v', p)$. To satisfy all such constraints in C, choose for each τ an arbitrary term of type τ, t_τ and let for any such mentioned v, v' and p, $v\mu = v'\mu = p\mu = t_\tau$.

8. For any substitution term with tail variable v in the original constraint set C, collect the decisions implied for $v\mu$ in all previous steps and construct a suitable closed substitution term for $v\mu$.

The constructed substitution μ grounds any variable in C and there are no constraints left in C which is not satisfied by μ. I.e., C is satisfiable. □

In the proof, we observed that the requirement of tails of substitution terms be open and unique were crucial for the construction of a satisfier.

Proof 2 (Proposition 4.8) Consider the semiunification problem

(*) $S_1\theta\sigma_1 = T_1\theta$ and $S_2\theta\sigma_2 = T_2\theta$

and let s_1, t_1, s_2, and t_2 be object level terms which arise when subterms that are name terms for object variables in $\lceil S_1 \rceil$, $\lceil T_1 \rceil$, $\lceil S_2 \rceil$, and $\lceil T_2 \rceil$ are replaced consistently by new meta-level variables. We can assume, thus, a meta-level substitution ρ that maps meta-variables to name terms for object variables such that

$$s_1\rho = \lceil S_1 \rceil, t_1\rho = \lceil T_1 \rceil, s_2\rho = \lceil S_2 \rceil, t_2\rho = \lceil T_2 \rceil.$$

Consider also the following constraint set,

$$C = \{\text{term-instance}(s_1, t_1), \text{term-instance}(s_2, t_2)\}.$$

We will argue that satisfiability of C is equivalent with (*).

Firstly, assume a meta-level substitution θ' such that $C\theta'$ is satisfied. By definition of satisfiability for instance constraints, there exist object level terms S_1^+, T_1^+, S_2^+, T_2^+ and object level substitutions σ_1 and σ_2 such that

$$s_1\theta' = \lceil S_1^+ \rceil,\ t_1\theta' = \lceil T_1^+ \rceil,\ s_2\theta' = \lceil S_2^+ \rceil,\ t_2\theta' = \lceil T_2^+ \rceil, \text{ and}$$

(**) $S_1^+\sigma_1 = T_1^+$ and $S_2^+\sigma_2 = T_2^+$.

Define, now an object level substitution θ such that

$$V\theta = T_V \text{ whenever } v\rho = \lceil V \rceil \text{ and } v\theta' = \lceil T_V \rceil.$$

With this we have, by compositionality of $\lceil - \rceil$,

$$S_1\theta = S_1^+,\ T_1\theta = T_1^+,\ S_2\theta = S_2^+,\ T_2\theta = T_2^+,$$

and it follows from (**) that θ, σ_1, σ_2 solves (*).

The other way round, assume (*) holds for some object level substitutions θ, σ_1, and σ_2. Define, now, a meta-level substitution θ' such that

$$v\theta' = \lceil V\theta \rceil \text{ whenever } v\rho = \lceil V \rceil.$$

With this we have, by compositionality of $\lceil - \rceil$,

$$s_1\theta' = \lceil S_1\theta \rceil,\ t_1\theta' = \lceil T_1\theta \rceil,\ s_2\theta' = \lceil S_2\theta \rceil,\ t_2\theta' = \lceil T_2\theta \rceil.$$

We get now

$$C\theta' = \{\text{term-instance}(\lceil S_1\theta \rceil, \lceil T_1\theta \rceil), \text{term-instance}(\lceil S_2\theta \rceil, \lceil T_2\theta \rceil)\},$$

which is satisfied according to (*). □

References

Abramson, H., and Rogers, M.H., eds., *Meta-programming in Logic Programming.* MIT Press, 1989.

Barklund, J., Boberg, K., and Dell'Acqua, P., A basis for a multilevel meta-logic programming language. Proc. 4th Intl. Workshop on Metaprogramming in Logic, Lecture Notes in Computer Science, to appear 1994.

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.

Bowen, K.A., Meta-level programming and knowledge representation. *New Generation Computing* 3, pp. 359–383, 1985.

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.

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.

Bruynooghe, M., ed., *Proc. of the Second Workshop on Meta-programming in Logic.* April 4–6, 1990, Leuven, Belgium.

Chan, D., Constructive negation based on the database completion, *Proc. of Fifth International Conference and Symposium on Logic Programming*, (eds. Kowalski, Bowen), pp. 111–125, MIT Press, 1988.

Christiansen, H., Declarative semantics of a meta-programming language. *In:* Bruynooghe, 1990, pp. 159–168.

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. (1992a).

Christiansen, H., A complete resolution method for logical meta-programming languages. *In:* Pettorossi, A, 1992, pp. 205–219. (1992b).

Christiansen, H., Why should grammars not adapt themselves to context and discourse? Abstract collection, *4th International Pragmatics Conference, Kobe, Japan, July 23–30 1993*, International Pragmatics Association, p. 23, 1993. (Full paper available from the author).

Christiansen, H., On proof predicates in logic programming. in: 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, 1994.

Cervesato, I., Lollipops taste of Vanilla too. in: 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, 1994.

Cervesato, I. Rossi, G., Logic meta-programming facilities in 'log. *In:* Pettorossi, A, 1992, pp. 148–161.

Colmerauer, A., *Prolog-II Manual de Reference et Modele Theorique. Groupe Intelligence Artificielle*, Universite d'Aix-Marseilles II, 1982.

Costantini, S. and Lanzarone, G.A., A metalogic programming language. *Proc. of the Sixth International Conference on Logic Programming*, (Levi, G., and Martelli, M., eds.), pp. 218–233, MIT Press, 1989.

Dell'Acqua, P., SLD-resolution with reflection. Ph.L. thesis, Uppsala University, 1994 (submitted).

Gallagher, J.P., *A system for specialising logic programs*. Technical Report TR-91-32, University of Bristol, Department of Computer Science, 1991.

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.

Gärdenfors, P., Belief revision: A vade-mecum. *In:* Pettorossi, A, 1992, pp. 1–10.

Girard, J.Y., Linear logic. *Theoretical Computer Science* 50, pp. 1-101, 1987.

van Harmelen, F., Definable naming relations in metalevel systems. *In:* Pettorossi, A, 1992, pp. 89–104.

Henglein, F., *Polymorphic type inference and semi-unification*. PhD thesis, New York University, Department of Computer Science, 1989.

Henglein, F., *Personal communication*, December 1994.

Hill, P.M. and 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., Analysis of meta-programs. *In:* Abramson, Rogers, 1989, pp. 23–51.

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

Hodas, J.S. and Miller, D. Logic programming in a fragment of linear logic. *Journal of Information and Control*, To appear 1994

Jaffar, J. and Lassez, J.-L., Constraint logic programming. *Proc. 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, 1987.

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

Kakas, A.A., Kowalski, R.A., Toni, F., Abductive logic programming. *Journal of Logic and Computation* 2, pp. 719–770, 1993.

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.

Ko, H.-P. and Nadel, M.E., Substitution and refutation revisited, *Logic Programming, Proc. of the Eighth International Conference*, ed. Furukawa, K., pp. 680–692, MIT Press, 1991.

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

Leiß, H., Polymorphic recursion and semi-unification. Computer Science Logic, Kaiserslautern, FRG, October 1989. *Lecture Notes in Computer Science* 440, Springer-Verlag, pp. 211–224, 1990.

Lloyd, J.W., *Foundations of logic programming*, Second, extended edition. Springer-Verlag, 1987.

Muggleton, S., ed., *Inductive logic programming.* Academic Press, 1992.

Muggleton, S., Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin* 2, no. 1, pp. 5–11, Jan. 1994.

Naish, L., Negation and control in Prolog, *Lecture Notes in Computer Science* 238, Springer-Verlag, 1986.

Numao, M. and Shimura, M., Inductive program systhesis by using a reversible meta-interpreter. *In:* Bruynooghe, 1990, pp. 123–136.

Pettorossi, A., ed. Proc. of META-92, Third International Workshop on Metaprogramming in Logic. *Lecture Notes in Computer Science* 649, Springer-Verlag, 1992.

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.

[Sicstus, 1994]: *SICStus Prolog user's manual.* Version 2.1 #9, SICS, Swedish Institute of Computer Science, 1994.

Stärk, R.F., *The completeness of semiunification.* Technical Report, CIS, Universität München, 1992.

Stuckey, P.J., Constructive negation for constraint logic programming. *Proc. of the 6th Annual Symposium on Logic in Computer Science (LICS)*, Amsterdam, pp. 328–339, 1991.