Implicit program synthesis by a reversible metainterpreter

Henning Christiansen

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

Abstract. Synthesis of logic programs is considered as a special instance of logic programming. We describe experience made within a logical metaprogramming environment whose central component is a reversible metainterpreter, in the sense that it is equally well suited for generating object programs as well as for executing them. Requirements telling that certain goals should be provable in a program sought can be integrated with additional sideconditions expressed by the developer at the metalevel, and the resulting specifications tend to be quite concise and declarative. For problems up to a certain degree of complexity, this provides a mode of working characterized by experimentation and an ability to combine different methods which is uncommon in most other systems for program synthesis. Reversibility in the metainterpreter is obtained using constraint logic techniques.

1 Introduction

The synthesis of a logic program is usually understood as a constructive process starting from a more or less complete logical specification and/or examples of what belongs (or not belongs) to the desired relation. This may be guided by the developer's growing understanding of the task or by heuristics built in to automatic or semiautomatic tools. This process can be performed on logic specifications, gradually being transformed into the subset consisting of efficient logic programs, or on other mathematical structures such as proofs from which programs can be extracted.

In this paper, we propose a slight shift which is more along the line of logic programming in general. The logic programmer tends to state requirements to the result he wants rather than concentrating on the procedure or process that leads to this result. In program synthesis, the desired result is a program — as opposed to, say, a sorted or reversed list.

We describe a metaprogramming environment whose central component is a reversible 'demo' predicate. This is a binary proof predicate parameterized by representations of object program and query, reversibility means that it can be used, not only for executing object programs, but also for generating them. By means of 'demo', the program developer (i.e., the metaprogrammer) can state semantic requirements to the program sought of the form "this and this must be provable" or "... not provable". This can be combined with syntactic requirements concerning which predicates, control patterns, etc. that are allowed, and perhaps other calls to demo that express integrity constraints or similar things. As in logic programming in general, we expect the underlying

interpreter to come up with solutions (here: programs) that satisfy the different requirements.

The possible success of such an approach depends on to what extent the available linguistic means make it possible to express "in a natural way" requirements that effectively delineate a class of interesting programs. We will show examples developed in our system and especially for simple abduction and induction tasks the approach looks promising, the definitions of such tasks being surprisingly concise and readable. The most interesting observation is, perhaps, to see how declarative specifications can be re-used in order to express different ways of reasoning. It appears also that the experimental mode of working, for which logic programming is appreciated (by some developers, at least) is preserved, by continually enhancing and testing a specification. For the synthesis of recursive programs, we cannot present similar experience and we will discuss possible extensions of our approach in the final section.

It is obvious to us that the sort of technology we present only can develop into a complement to "explicit" techniques based on program transformations. There *are* knowledge and heuristics needed for complex and realistic programming tasks which best can be described — if it can be described at all — by means of transformation techniques. This is quite analogous to the fact that declarative programming never completely can rule out procedural programming.

In the rest of this introduction we give an overview of the approach and a comparison with related work. Section 2 describes the principles behind our constraint-based implementation of 'demo' and section 3 sketches the features of the implemented system that we will refer to as the Demo system. Section 4 shows examples of program synthesis developed in this system, one giving a combination of default reasoning with abduction and induction, and a natural language analysis based on abduction in a way that would be difficult to fit into other systems for abduction. Section 5 discusses possible improvements to make it possible to approach more complex problems. The Demo system is available on the World Wide Web at the address http://www.dat.ruc.dk/software/demo.html.

1.1 A brief overview

An approach to program synthesis has an object language in which the resulting programs appear, in our case positive Horn clause programs. We represent object programs and fragments thereof as ground terms in a metalanguage which basically is Prolog extended with various metaprogramming facilities, most notably the 'demo' predicate. For any phrase P of the object language, the notation $\lceil P \rceil$ will stand for the term in the metalanguage designated as the name of P. The 'demo' predicate is a formalization of the usual notion of provability embedded in a logic program interpreter; we can specify it as follows.

$$\text{demo}(\lceil P \rceil, \lceil Q \rceil) \quad \text{iff} \quad P \text{ and } Q \text{ are object program and query such that there exists substitution } \sigma \text{ with } P \vdash Q\sigma$$

A metavariable, say X, placed in the first argument will thus stand for a piece of program text, and a logically satisfactory implementation will produce program fragments

which make Q provable. By means of additional side-conditions, 'demo' can be instructed to produce programs within a certain class as illustrated by the following pattern.

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

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

1.2 The need for constraints and delays

It can be argued that a direct implementation of in "classical" Prolog, with its top-down computation rule, is insufficient in order to provide the reversibility in the 'demo' predicate that we have assumed. The problem is that the space of solutions to a call to 'demo' — or the subgoals we can expect inside 'demo' — typically is infinite in such a way that no finite set of terms will cover it by subsumption. This implies that a failure in the computation easily can provoke a generate-and-test-forever loop. In addition, all known implementations in Prolog of 'demo' (i.e., those parameterized by a detailed representation of the object program) include tests which, depending on the underlying interpreter, will lead to either floundering or unsound behaviour in case of an uninstantiated metavariable standing for a part of the program being interpreted. An obvious way to get around this problem is to use constraint logic. We have used a well known program structure for implementing 'demo', but with the primitive operations implemented as constraints — which means that these operations typically delay when called with partly instantiated arguments, and the constraint solver makes sure that the set of delayed constraints always is satisfiable. In section 2 we describe our constraint-based implementation; the constraint solver can be found in the appendix.

A similar argument can be made for the additional sideconditions depicted as the 'useful' predicate above. However, it is a quite difficult task to write a constraint solver and luckily, as it will appear in our examples, it turns out to be sufficient to use the sort of delay mechanisms that are available in current implementations of Prolog [36].

The 'demo' predicate was initially suggested by Kowalski in his book from 1979 [28], however, with no attention to the possibility of using it in reverse for program synthesis. In 1992, T. Sato [35] and Christiansen [7] published independently different methods for obtaining this kind of reversibility in 'demo' although these results were mostly of a theoretical interest. Apart from our own constraint-based implementation, we are not aware of any practically relevant, fully reversible versions; for more background in this topic, see [8].

1.3 Related work

For a general overview of the area of logic program synthesis, we refer to the present and previous proceedings of the LOPSTR workshops, and to comprehensive reviews contained in two recent books [9, 10].

In order to compare our approach with other work, we emphasize the notion of program schemas. A program schema is a template which represents a certain class of programs, e.g., the class of divide-and-conquer programs, having placeholders which can be instantiated as to produce concrete programs. Such schemas are used in "manual" approaches to program construction, e.g., [1, 17], and in (semi-) automatic transformation-based approaches, e.g., [14, 12, 5, 13, 33]. A transformation can be specified in such a way that a program matching one schema is transformed into another, intendedly improved, program described by another schema. Program schemas can be understood as second-order objects [6, 18, 22]. Other authors [5, 13] represent program schemas by means of so-called open first-order programs, open in the sense that some of the predicates used are unspecified.

In our framework, the ground representation combined with metalevel predicates supplied by the user serves the purpose of program schemas. Programs are given by first order terms, and ordinary logical variables take the role of placeholders, referred to here as metavariables as they are part of a metalanguage and in general stand for representations of object program fragments. In [6] a suggestion is given for a general framework for program schemas integrated with constraints, quite similar to what we obtain in a logic metaprogramming setting.

Most approaches to program synthesis, including those mentioned above, incorporate some kind of strategy for controlling the application of transformation steps in order to provide a convergence towards an improved program. In our approach, which is an application of logic programming, 'demo' becomes a blind robot, which will produce in principle any program satisfying the different requirements set up by the developer. As in logic programming in general, it is difficult to point out a *best* solution to a problem, which is relevant when we are searching not only for a correct program but also for an efficient one. On the other hand, this is partly compensated by the ease with which new criteria can be set up.

We also mention the work by Numao and Shimura from 1990 [34] which seems to be the first suggestion for using a reversible 'demo' predicate for program synthesis. They propose a quite interesting approach, using explanation based learning [30] for analyzing existing programs in order to re-use their inherent strategy for the synthesis of new programs. This can be seen as an automatic way to extract "program schemas" that otherwise may be difficult to formalize by hand. It could be interesting to take up this idea with a capable implementation of 'demo' such as our constraint-based version.

Inductive logic programming (ILP) is another area concerned with problems similar to those that we approach, namely the synthesis of programs from samples of their intended behaviour and additional conditions about the sort of programs that are preferred (in ILP called "bias"); we refer to [2, 31, 32] for an overview and introduction to the most common methods. The ILP systems described in the literature are typically directed towards specific kinds of applications and the techniques used are often related to those used in transformation-based program synthesis. Within specific areas, ILP has reported impressive results that we cannot match in our declarative and highly generic approach. On the other hand, it appears to be fairly easy in our system to adapt and combine different methods, e.g. integrating abduction in induction which also have been introduced in ILP methods recently [11, 29].

Hamfelt and Nilsson [19] have proposed to use 'demo' together with a higher-order 'fold' operator and least-general-generalization techniques for producing recursive programs. The authors apply a multilayered structure ('demo' interpreting 'demo') as a means to get around the mentioned generate-and-test problems which we avoid using constraints. The advantage of using recursion operators such as 'fold' is that 'demo' only needs to invent non-recursive plug-in's in order to define, say, the 'append' predicate. It is difficult to judge the generality of this proposal but it seems worthwhile to combine our 'demo' with more recent and thorough work by the authors [20, 21] on recursion operators.

Finally, we contrast our approach with the large-scale project presented by Bibel *et al* [3] in these proceedings, clearly heading at a methodology for programming in the large. We consider our Demo system as a light-weight and easy-to-use environment, primarily intended for experimentation and illustrative purposes in research and teaching, perhaps useful as a prototyping tool in larger contexts.

2 A constraint-based metainterpreter

In this section we describe the theoretical background for our constraint-based implementation of 'demo'. In a first reading of this paper, it may be recommended only to take a brief look at the first subsection and then go directly to the examples in section 3. For the full account on the technical matters, including proofs, we refer to [8].

2.1 Syntax and semantics of object and metalanguage

The presence of a naming relation induces a natural classification of metalevel terms into different types, namely those that stand for programs, those that stand for clauses, etc., We consider a class of constraint logic languages CLP($\mathcal{X}$) similarly to [25], but here adapted for typed languages. In our case, the parameter $\mathcal{X}$ refers to some domain of constraints over terms with no interpreted function symbols. Each such constraint logic language is characterized by a finite set of types,[1] collections of predicate, constraint and function symbols, and variables, each with fixed ranks/types.

Capital letters such as X and Y are used for variables; the underline character '_' is used as an anonymous variable in the sense that each occurrence of it stands for a variable that does not occur elsewhere.

A *program* is a finite set of *clauses* of the form $h \leftarrow b_1 \wedge \ldots \wedge b_n$ with h being an atom, each b_i an atom or a constraint, composed in the usual way respecting the ranks of each symbol; a *query* is similar to the body of a clause. Queries and bodies of clauses are collectively called *formulas* and we assume two inclusion operators, both denoted $\uparrow$, from atoms and from constraints into formulas; to simplify the notation, we leave out these operators except in a few, essential cases. Finally, the truth constant *true* is used to indicate the empty body of a fact.

The meaning of the constraints in a given language is assumed given by a set of ground constraints referred to as *satisfied* constraints. We assume, for each type τ, constraint symbols '$=: \tau * \tau$' and '$\neq: \tau * \tau$' with the usual meanings of syntactic identity

[1] We consider only simple types with no notion of parameterization or subtypes.

and non-identity. To cope with the semantics of $\neq$ constraints, we require, for each type τ, that there exist infinitely many constant symbols (not necessarily of rank $\rightarrow \tau$) which can occur in a term of type τ.

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

Definition 1. *The* proof relation *for a constraint language $\mathcal{L} = \text{CLP}(\mathcal{X})$, denoted $\vdash_{\mathcal{L}}$, between programs and ground queries is defined inductively as follows.*

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

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

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

2.2 The metalanguage CLP($\mathcal{HCL}$)

The 'demo' predicate is programmed in a language CLP($\mathcal{HCL}$) having function symbols that reflect the syntax of $\mathcal{HCL}$ and constraints that make it possible to express its proof relation. CLP($\mathcal{HCL}$) has the following types:

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

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

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

and for each $\mathcal{HCL}$ variable, say X, a constant $'\text{X}: \rightarrow$ *term*. For any phrase P of $\mathcal{HCL}$, the notation $\lceil P \rceil$ refers to the ground term that arises when each symbol f occurring in P is replaced by $'f$ and we call $\lceil P \rceil$ a *name* for P. Formally, these brackets are not part of the metalanguage but will be used as syntactic sugar. The reverse brackets $\lfloor \cdots \rfloor$ are used inside $\lceil \cdots \rceil$ to indicate the presence of a metavariable. If, for example, Z is a metavariable of type *atom*, we have

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

In addition, there are functions to build representations of object programs and substitutions, we will use Prolog's list notation for those, and substitution pairs will be written (x, t) where x names an object variable and t an object term.

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

instance$_\tau$: $\tau * \tau *$ *substitution*.

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

no-duplicates: *program*,
member: *clause* $*$ *program*
not-member: *clause* $*$ *program*

Constraint satisfaction is defined by exactly the following ground constraints recognized as satisfied:

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

The 'member' constraints, used for selecting clauses out of an object program, could in principle have been specified as an ordinary predicate, but in order to suppress the generation of different representations of the same object program, we need to have a detailed procedural control, which is only possible by explicit derivation rules. The 'no-duplicates' constraints are used to ensure that terms of type *program* really are names of programs; 'not-member' and $\neq$ constraints are used here as auxiliaries. A constraint solver for CLP($\mathcal{HCL}$) is described in the appendix.

2.3 The metainterpreter

Using CLP($\mathcal{HCL}$), we can give our constraint-based version of the so-called 'instance-demo' predicate, which has been studied by several other authors recently [15, 16, 23, 4].

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

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

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

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

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

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

The $\uparrow$ symbol is the inclusion operator of atoms into formulas that is left implicit in most other cases; the purpose of the 'no-duplicates' constraint is to impose our convention of programs being sets and not lists of clauses, a property which cannot be expressed directly with the sort of types normally used in logic programming. This, together with the rules of the constraint solver (see appendix) for 'member' constraints, prevents the generation of alternative presentations of the same program due to permutation and duplication of clauses.

Precise statements and proofs of soundness and completeness can be found in [8]. We will mention here that satisfiability of instance constraints in general may be expected to be undecidable. More precisely, if we had used general substitutions in our framework — and not idempotent ones, as we prefer — the satisfiability problem becomes equivalent to the multiple semi-unification problem known to be undecidable [27]. Whether this disappears in the idempotent case is not known at present. However, the way the constraints are used in the 'demo' program implies a number of invariant properties that ensure termination in constraint solving. The most important invariant is that of *safeness* which means that the sets of metavariables occurring in first arguments to instance constraints and those occurring in second arguments are disjoint.

3 Outline of the implemented Demo system

The metalanguage CLP($\mathcal{HCL}$) has been implemented in Sicstus Prolog [36] using its attributed variables' library. Soundness and completeness results are preserved for the constraint solver but the 'demo' program is interpreted directly by Prolog and inherits, thus, Prolog's overall termination behaviour.

In the implemented system, we use actually a more detailed naming relation which makes it possible to implement a subtype relation, so that, e.g., *constant* is a subtype of *term* and *atom* a subtype of *formula* (which means that the inclusion operator $\uparrow$ becomes unnecessary). Types are implemented in Sicstus Prolog also as constraints, so for each type τ of CLP($\mathcal{HCL}$), there is a constraint $\tau(t)$ satisfied for exactly all terms t of type τ. These constraints are represented as predicates `program_(-)`, `clause_(-)`, `atom_(-)`, etc., the underline character consistently used to distinguish constraints from a few Prolog built-in's.

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

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

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

`m1` and `m2` are expected be defined as object program modules.

We can illustrate the use of the extended syntax and program modules by a very simple example of abduction inspired by [26]. The following source file defines an object module called `garden` together with an auxiliary predicate called `abducible`.

```
:- object_module( garden,
     \[ (grass_is_wet:- rained_last_night),
        (grass_is_wet:- sprinkler_was_on)
      ]).

:- block abducible(-).

abducible( \ (rained_last_night:- true) ).
abducible( \ (sprinkler_was_on:- true) ).
abducible( \ (full_moon:- true) ).
```

The `block` directive is a Sicstus Prolog primitive which causes the `abducible` predicate to delay until its argument gets instantiated. In less trivial examples, the use of delays may become essential in order to prevent such auxiliary conditions from enumerating under backtracking the perhaps infinite space of their solutions. It should be emphasized that although the `block` directive is a procedural device, it does not destroy the declarative semantics as do other control facilities in Prolog.

We can now express the abductive problem of finding a cause why `grass_is_wet` holds in the following dialogue with the system.[2]

```
?- abducible(WHY), demo(\ (garden & [?WHY]), \\grass_is_wet).

WHY = \ (rained_last_night:-true) ? ;
WHY = \ (sprinkler_was_on:-true) ? ;

no
```

The program argument to `demo` describes a conjunction of the `garden` module and a program consisting of one unknown clause, indicated by the metavariable `WHY` and the the system returns the possible, abducible explanations.

Other facilities of the implemented system will be explained as they appear in the examples.

4 Examples, program synthesis by means of 'demo'

Here we will show two examples developed in the Demo system, the first showing an integration of different modes of reasoning, the second a variant of abduction applied for a simplified natural language analysis problem. These and other application of Demo concerning view update, abduction in a fragment of linear logic, and diagnosis are described in [8] and available together with the implemented system (see WWW address in the introduction).

[2] All examples are authentically produced in our implemented system except that we have retouched away explicit calls to an alternative reader that recognizes the extended syntax; by a more careful coding of the user interface, this could have been avoided anyhow.

We do not consider, here, synthesis of recursive programs as we cannot present any convincing examples; we discuss this issue in the concluding section.

4.1 Default reasoning combined with abduction and induction

This is intended as a "working example" and we do not care about the formal relation to other work on default reasoning. We consider only monadic predicates and constants. The current system cannot handle negation so we represent a negative literal $\neg p(a)$ as `p(a,no)` and correspondingly $p(a)$ as `p(a,yes)`. We consider a default theory to be a triplet $\langle F, D, E \rangle$ where, informally,

- F is a set of facts,
- D is a set of default rules of the form p`(X,yes) :-` q`(X,yes)`; only ground instances of default rules that do not violate the overall consistency can be used in a proof,
- E is a set of exception of the form p`(X,no) :-` q`(X,yes)`.

The notion of consistency of a program with encoded `yes`/`no` values can be formalized at the metalevel in the following way; the notation `\\ ··· ?Q-[?C,?YN1]` displays an alternative syntax for atoms which makes it possible to parameterize over the predicate symbol, so here `Q` is a metavariable ranging over predicate names.

```
consistent(P):-
    for_all( (constant_(C), constant_(YN1), constant_(YN2),
              demo(P, \\ (?Q-[?C,?YN1], ?Q-[?C,?YN2]))),
             YN1=YN2 ).
```

The `for_all` construct is control device using a negation-as-failure principle[3] to generate all solutions given by the first argument and the whole construct succeeds if the second argument succeeds in all cases. So the definition reads:

> A program is consistent whenever, for any predicate q and constant a, we do not have q`(`a`,yes)` and q`(`a`,no)` at the same time.

We should be aware, however, that this implementation of the consistency check only works correctly when the program argument does not contain uninstantiated metavariables.

A first sketch of a formalization in our system of provability in a default theory may look as follows, where `D1` stands for some set of ground instances of the default rules `D`, and `F`, `E` for the other components of the default theory, `Q` represents the query or "observations" to be proved.

```
demo(\ (?F & ?D1 & ?E), Q),
consistent(\ (?F & ?D1 & ?E ).
```

The following proof predicate will be sufficient for deductive reasoning within a default theory. In this case, the default theory is completely given in advance, i.e., given by a ground term at the metalevel.

[3] Implemented as follows, `for_all(P,T):- \+ (P, (T -> fail ; true)).`

```
demo_default(F,D,E, Q):-
    default_instances(D, D1),
    demo(\ (?F & ?D1 & ?E), Q),
    close_constraints(D1),
    consistent(\ (?F & ?D1 & ?E) ).
```

We will not show definitions for all auxiliary metapredicates, but `default_instances` can give a good illustration of the general pattern.

```
% default_instances(Defaults,Instances).
:- block default_instances(?,-).

default_instances(_,\[]).

default_instances(D, \[(?P-[?C,yes]:- ?Q-[?C,yes] ) | ?E]):-
    member_(\ (?P-[?_,yes]:- ?Q-[?_,yes]), D),
    constant_(C),
    default_instances(D, E).
```

This implements, quite directly, that the second argument is a program of clauses created from members of the first by replacing the object variable by some constant. The `block` directive makes the predicate delay until its second argument gets instantiated (here by an event inside 'demo' when it attempts to grab a clause in the yet uninstantiated program component `D1`). The tail recursion means that the predicate will iterate in a lazy fashion through `D1` as more and more elements may be needed. The `member_` predicate is the system's standard constraint for selecting clauses in object programs.

Finally, we should explain the `close_constraints` predicate used in the definition of `demo_default`. It is a device built into the system that investigates the possibly uninstantiated metavariables in its argument and, according to certain heuristics, instantiates to prototypical values that will satisfy the pending constraints. In the particular case above, it will close the open program tail of `D1`, which is necessary in order to have the naively implemented consistency check to execute correctly.

To test our proof predicate for default theories, we define the following object modules.

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

:- object_module( defs,
     \[ (bird(X,yes):- penguin(X,yes)),
        (fly(X,yes):- bird(X,yes))     ]).

:- object_module( excs,
     \[ (fly(X,no):- penguin(X,yes))   ]).
```

The following queries and answers show the overall behaviour of the `demo_default` predicate defined above.

```
?- demo_default(\facts,\defs,\excs,
                \\ (fly(john,yes), fly(tweety,no))).
```

```
yes
?- constant_(X),
   demo_default(\facts,\def,\excs, \\fly(?X,yes)).

X = \\\john
```

When moving from deduction to abduction and induction we have to formalize the assumptions we have made implicitly about what constitutes a default theory, otherwise the `demo` predicate will be free to invent any strange program that makes the given query provable. As a first approach, we can write predicates `facts(-)`, `defaults(-)`, and `exceptions(-)` which in a straightforward way formalize the syntactic assumptions mentioned in the introduction to this example, e.g.,

```
:- block exceptions(-).

exceptions(\[]).

exceptions(\[(?P-[X,no]:- ?Q-[X,yes]) | ?More]):-
    exceptions(More).
```

We can now improve the definition of the `demo_default` predicate in such a way that it may be able to answer in a reasonable way queries with partly specified theory.

```
demo_default(Facts,Defaults,Excs,Obs):-
    facts(Facts),
    defaults(Defaults),
    exceptions(Excs),
    default_instances(Defaults, DefIs),
    demo(\ (?Facts & ?Excs & ?DefIs), Obs),
    close_constraints(\ (?Facts & ?Excs & ?DefIs) ),
    consistent(\ (?Facts & ?Excs & ?DefIs) ).
```

The following queries and answers illustrate abduction and induction of defaults-with-exceptions from examples.

```
?- demo_default(F,\defs,\excs, \\fly(tweety,no)).

F = \[(penguin(tweety,yes):-true)]

?- demo_default(\facts,\[?D],\[?E],
                \\ (fly(tweety,no), fly(john,yes))).

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

In the first query we abduce, from the background knowledge given by modules `defs` and `excs`, the fact that `tweety` is a `penguin` from the observation that `tweety` does not `fly`. In the second, the background knowledge `facts` together with the observations about the two individuals' (lacking) ability to `fly` makes the system conclude the expected pair of default and exception rules.

While `demo_default` works quite well in these examples, where some background knowledge is given in each case, it will not work as a general concept learner that produces from scratch a class of "intuitively correct" default theories that can explain or systematize a larger sample of observations. This will require a further refinement of the metalevel conditions that takes into account a suitable stratification among the different predicates.

This example showed the Demo system "at work" for small problems of automated reasoning and it is worth noticing the very little amount of code that the developer had to supply in order to implement different reasoning methods. Another important point is displayed, namely the learning process went through by the developer in this test-refine-and-extend cycle which is characteristic for logic programming. If someone elaborated the example above further into a general concept learner as indicated, we will claim that he or she would have learned some essential points about logic and machine learning — although not so much about the algorithmic concerns that need to be considered for solving complex problems.

4.2 A natural language example

Here we show a variation of abduction which seems difficult to implement in a similar direct way in those systems for abduction, we have seen described in the literature, see [26] for an overview. Our example is concerned with the relation between simple still-life scenes and sentences about them.

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

```
:- object_module( grammar,

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

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

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

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

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

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

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

The system produces the following three answers.

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

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

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

We can also extend the example by abducing a `things` program T in parallel with the `scene`. In case a fact, say, `in(the_dog, the_house)` is abduced, the integrity constraint will abduce in turn as part of T the facts `thing(the_dog)`, `container(the_house)`. Furthermore, the integrity constraint concerned with T (not shown) will trigger the abduction of `thing(the_house)`. This recursive triggering of new abductions via integrity constraints, that appears quite natural in our metaprogramming setting, does not appear to be possible in other implementations of abduction that we are aware of.

As in the previous example, we could continue to refine and extend the metalevel conditions, for example for inducing grammars from sample sentences. This would require a formalization at the metalevel of, firstly, what sort of object programs that can be conceived as grammars and, secondly, what it means for a grammar to be a good grammar.

5 Conclusions, possible extensions of the Demo system

We have presented an approach to program synthesis based on metaprogramming in logic, using a ground representation of object programs and a reversible 'demo' predicate. The ground representation (when supported by a proper notation!) together with metalevel predicates provided by the user, appears to be a quite flexible tool for describing different classes of not-too-complex programs, thus serving the purpose program schemas used in other approaches. For the sort of problems that we have illustrated in our examples, the Demo system appears as a highly generic environment in which different modes of reasoning can be developed and combined with each other in a straightforward way which seems quite uncommon in those other systems for program synthesis and automated reasoning that we are aware of.

When it comes to the specification of more complicated patterns, say, defining a class of divide-and-conquer programs, the direct and isomorphic representation that we have used tends to be insufficient. In order to complete our approach for handling more complex program synthesis tasks, it seems relevant to suggest alternative representation forms together with specialized constraints that give a certain bias towards particular program structures. If, for example, we are interested in synthesizing recursive programs, a representation based on the recursion operators of [20, 21] may seem appropriate (see also discussion in section 1.3). In a forthcoming re-implementation of the Demo system, we will support the development of such representations, by making the naming relation extensible by the user and by providing a layer for constraint programming, which makes it possible for users to write constraint solvers in terms of derivation rules similar to those appearing in the appendix. We will also refer to [6] which can be seen as a starting point for the systematic development of such representations.

Another striking shortage in the present Demo system is its lack of ability to handle negative examples in a proper way. We suggest here, to add an additional proof predicate, `demo_fails(`$\lceil P \rceil$`,` $\lceil Q \rceil$`)` with the meaning that the query Q fails in the program P. This predicate should be implemented in such a way that, whenever in a standard negation-as-failure search, it runs into an open program part (i.e., an uninstantiated metavariable), it delays. In general, this will result in several delayed or-branches and if one of them reports that Q can succeed, the call to `demo_fails` should fail immediately. In this way, negative examples can be set up as conditions before the usual `demo` predicate is set to work, and when some action triggered inside `demo` adds something to the object program that makes one negative example succeed, `demo` (or the 'responsible' side-conditions) will be forced to take another choice. With such an extension, it should be possible to approximate some of the results gained in the ILP area. With similar techniques it may also be possible to have integrity constraints in, say, view update problems to execute in a truly incremental way.

Appendix: A constraint solver for CLP($\mathcal{HCL}$)

Constraint solving for CLP($\mathcal{HCL}$) is described by means of top-down derivations in the sense of [25], however here adapted for typed languages. As we have in mind implementations using Prolog-like technology, we assume an indivisible (and efficiently implemented) unification operation in the underlying interpreter.

In general, we define a *derivation system* to consist of *transition rules* $S \rightsquigarrow S'$ over *states* of the form

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

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

States are assumed to be *idempotent* in the sense that no variable $x \in \mathrm{dom}(\alpha)$ occurs in C, i.e., α is mapped consistently over the constraint set C. We assume a special state called FAILURE with empty set of satisfiers.

We use also $\rightsquigarrow$ to denote the *derivation relation* induced in the natural way by a set of derivation rules $\{\rightsquigarrow\}$; $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$. Whenever, for some query Q, that $\langle Q, \emptyset \rangle \rightsquigarrow^* S$, where S does not contain atoms and no derivation step is possible from S, any satisfier for S restricted to the variables of Q is called a *computed answer* for Q.

The following transition rules, labeled (Unif), (Res), (Dif1), (Dif2), and (True), constitute the core of any transition system. The unification rule (Unif) is the only one that changes the accumulated substitution.

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

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

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

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

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

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

Definition 2. *Two terms t_1 and t_2 are said to be* distinguishable *if they have no unifier, i.e., for any substitution σ, $t_1\sigma$ and $t_2\sigma$ are different.*
□

The following two rules define a behaviour of inequalities quite similarly to the `dif` predicate of Sicstus Prolog [36].

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

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

The constraint solver for CLP($\mathcal{HCL}$) assumes a computation rule which is

fast-solving: the resolution rule (Res) cannot be applied if another rule is applicable, and

fast-unifying: the rules (Unif), (Dif1), and (Dif2) take precedence over all other rules.

The following derivations rules are specific for CLP($\mathcal{HCL}$). We start with the rules concerned with instance constraints. The rule (I) expresses that a given variable can have one and only one instance under a given substitution.

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

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

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

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

Notice that a fast-unifying computation rule is relevant for (It2) in order to avoid different and incompatible expansions of the substitution tail w. — An invariant property can be shown, that these substitution argument always have open tails so that rule (It2) always can apply when relevant.

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

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

The reduction of a term instance constraint is, thus, triggered by its first argument being non-variable. Rules for all other syntactic constructs in $\mathcal{HCL}$ of categories *clause*, *formula*, *atom*, and *constraint* are defined similarly to (It3) except that they are triggered also by the second argument being non-variable. As an example, we show the rule (Ic) for reduction of instance constraints on clauses.

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

Member constraints and its companions are reduced by the following rules.

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

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

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

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

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

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

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

Termination and soundness conditions are formulated and proved in [8].

References

1. Barker-Plummer, D. Cliche programming in Prolog. Bruynooghe, M. (ed.), *Proc. of the Second Workshop on Meta-programming in Logic*. April 4–6, 1990, Leuven, Belgium. pp. 247–256, 1990.
2. Bergadano, F., Gunetti, D., *Inductive logic programming, From machine learning to software engineering*. MIT Press, 1996.
3. Bibel, W., Korn, D., Kreitz, C., Kuruc, F., Otten, J., Schmitt, S., Stopmann, G., A multilevel approach to program synthesis. In: Fuchs, N.E. (ed), *Proc. of LOPSTR'97* (this volume).
4. Bowers, A.F., Gurr, C.A., Towards fast and declarative meta-programming, In: Apt, K.A., Turini, F. (eds), *Meta-Logics and Logic Programming*, MIT Press, pp. 137–166, 1995.
5. Büyükyıldız, H., Flener, P., Generalized logic program transformation schemas. In: Fuchs, N.E. (ed), *Proc. of LOPSTR'97* (this volume).
6. Chasseur, E., Deville, Y. Logic program schemas, constraints and semi-unification. In: Fuchs, N.E. (ed), *Proc. of LOPSTR'97* (this volume).
7. Christiansen, H., A complete resolution method for logical meta-programming languages. In: Pettorossi, A. (ed.), Meta-Programming in Logic, *Lecture Notes in Computer Science* 649, pp. 205–219, 1992.
8. Christiansen, H., Automated reasoning with a constraint-based metainterpreter. To appear in: *Journal of Logic Programming*, 1998.
9. Flach, P. *Simply logical, Intelligent reasoning by example.* Wiley, 1994.
10. Flener, P. *Logic program synthesis from incomplete information*. Kluwer, 1995.
11. Flener, P. Inductive logic program synthesis with DIALOGS. In: Muggleton, S. (ed), Proc. of ILP'96. *Lecture Notes in Artificial Intelligence* 1314, Springer-Verlag, pp. 175–198, 1997.
12. Flener, P., Deville, Y. Logic program transformation through generalization schemata. In: Proietti, M. (ed), Proc. of LOPSTR'95. *Lecture Notes in Computer Science* 1048, Springer-Verlag, pp. 171–173, 1996.
13. Flener, P., Lau, K.-K., Ornaghi, M. On correct program schemas. In: Fuchs, N.E. (ed), *Proc. of LOPSTR'97* (this volume).
14. Fuchs, N.E., Fromherz, M.P.J, Schema-based transformation of logic programs. In: Clement, P., Lau, K.-K. (eds.), *Proc. of LOPSTR'91*, Springer-Verlag, pp. 111–125, 1992.

15. Gallagher, J.P., *A system for specialising logic programs*. Technical Report TR-91-32, University of Bristol, Department of Computer Science, 1991.
16. 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.
17. Gegg-Harrison, T.S. Learning Prolog in a schema-based environment. *Instructional Science* 20, pp. 173–192, 1991.
18. Gegg-Harrison, T.S. Representing logic program schemata in λProlog. Sterling, L. (ed.). *Proc. of ICLP'95*, MIT Press, pp. 467–481, 1995.
19. Hamfelt, A., Nilsson, J.F. Inductive metalogic programming. In: S. Wrobel (ed), *Proc. of ILP'94*, pp. 85–96. *GMD-Studien* Nr. 237, Sankt Augustin (Germany), 1994.
20. Hamfelt, A., Nilsson, J.F. Declarative logic programming with primitive recursive relations on lists. Maher, M. (ed.). *Proc. of JISCLP'96*, MIT Press, pp. 230–243, 1996.
21. Hamfelt, A., Nilsson, J.F. Towards a logic programming methodology based on higher-order predicates. *New generation Computing* 15, pp. 421–228, 1997.
22. Hannan, J., Miller, D. Uses of higher-order unification for implementing program transformers. Kowalski, R.A., Bowen, K.A. (eds). *Proc. of ICLP'88*, MIT Press, pp. 942–959, 1988.
23. Hill, P.M., Gallagher, J.P., Meta-programming in Logic Programming. To be published in Volume V of *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press.
    Currently available as *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.
24. Hill, P.M. and Lloyd, J.W., Analysis of meta-programs. *Meta-programming in Logic Programming*. Abramson, H., and Rogers, M.H. (eds.), MIT Press, pp. 23–51, 1989.
25. Jaffar, J., Maher, M.J., Constraint logic programming: A survey. *Journal of logic programming*, vol. 19,20, pp. 503–581, 1994.
26. Kakas, A.A., Kowalski, R.A., Toni, F., Abductive logic programming. *Journal of Logic and Computation* 2, pp. 719–770, 1993.
27. 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.
28. Kowalski, R., *Logic for problem solving*. North-Holland, 1979.
29. Lamma, E., Mello, P., Milano, M., Riguzzi, F. A hybrid extensional/intensional system for learning multiple predicates and normal logic programs. In: Fuchs, N.E. (ed.), *Proc. of LOPSTR'97*, (this volume).
30. Mitchell, T.M., Keller, R.M., Kedar-Cabelli, S.T. Explanation-based generalization: A unifying view. *Machine Learning* 1, pp. 47–80, 1986.
31. Muggleton, S., de Raedt, L. Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19/20, pp. 669–679, 1994.
32. Nienhuys-Cheng, S.-H., de Wolf, R., Foundations of Inductive Logic Programming. *Lecture Notes in Artificial Intelligence* 1228, Springer-Verlag, 1997.
33. Richardson, J., Fuchs, N.E., Development of correct transformation schemata for Prolog programs. In: Fuchs, N.E. (ed.), *Proc. of LOPSTR'97*, (this volume).
34. Numao, M., Shimura, M. Inductive program synthesis using a reversible meta-interpreter. In: Bruynooghe, M. (ed.), *Proc. of the Second Workshop on Meta-programming in Logic*. April 4–6, 1990, Leuven, Belgium. pp. 123–136, 1991.
35. 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.
36. *SICStus Prolog user's manual*. Version 3#5, SICS, Swedish Institute of Computer Science, 1996.
    See also `http://www.sics.se/isl/sicstus/sicstus_toc.html`.