Inductive Logic Program Synthesis with DIALOGS

Pierre Flener

Department of Computer Engineering and Information Science
Faculty of Engineering, Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr Voice: +90/312/266-4000 ext. 1450

Abstract. DIALOGS (Dialogue-based Inductive and Abductive LOGic program Synthesizer) is a schema-guided synthesizer of recursive logic programs; it takes the initiative and queries a (possibly computationally naive) specifier for evidence in her/his conceptual language. The specifier *must* know the answers to such simple queries, because otherwise s/he wouldn't even feel the need for the synthesized program. DIALOGS can be used by any learner (including itself) that detects, or merely conjectures, the necessity of invention of a new predicate. Due to its foundation on a powerful codification of a "recursion-theory" (by means of the template and constraints of a divide-and-conquer schema), DIALOGS needs very little evidence and is very fast.

1 Introduction

This paper results from a study investigating (*i*) what is the minimal knowledge a specifier must have in order to want a (logic) program for a certain concept, and (*ii*) how to convey exactly the corresponding information, and nothing else, to a (logic) program synthesizer (be it automated or not). I argue that "knowing a concept" means that one can act as a decision procedure for answering certain kinds of simple queries [1] about that concept, but that it doesn't necessarily imply the ability to actually write such a decision procedure. More provocatively, I could argue [13] that writing a complete formal specification is often tantamount to writing such a decision procedure (because it actually features a naive or inefficient algorithm), and is thus often beyond the competence of a "computationally naive" specifier. But the reader need not agree on the latter claim, so let's assume, for whatever reasons, that some specifier wants to, or can only, give incomplete information about a concept for which s/he wants a (logic) program. This is an innovative program development technique, especially aimed at two categories of users:

- *experienced programmers* would often rather just provide a few carefully chosen examples and have a synthesizer "work out the details" for them;
- *end users* are often computationally naive and cannot provide more than examples, but this should allow them to do some basic programming tasks, such as the recording of macro definitions, etc.

As this project is not about natural language processing, let's also assume that the specification language is nevertheless formal.

The synthesizer must thus be of the inductive and/or abductive category. However, many (but not all) such synthesizers have the drawback of requiring large amounts of ground positive (and negative) examples of the intended concept, especially if the resulting program is recursive. The reasons are that ground examples are a poor means of communicating a concept to a computer, and/or that the underlying "recursion theory" is poor. To address the first reason, some researchers have successfully experimented with non-ground examples [18], if not Horn clauses [9, 12] or even full clauses [6], as evidence language. To address the second reason, schema-guided synthesis has been proposed [9, 12].

Especially since the advent of ILP (Inductive Logic Programming), the learning/synthesis of non-recursive programs (or concept descriptions) has made spectacular progress, but not so the synthesis of recursive programs. I have therefore decided to focus on the latter class of programs, to the point where my synthesizers even *assume* that there exists a recursive logic program for the intended concept. Even though this seems counterproductive, because a synthesizer can't decide in advance whether a concept has a recursive program or not, there are two good reasons for this focus and assumption. First, as advocated by Biermann [3], I believe it is more efficient to try a suite of fast and reliable class-specific synthesizers (and, if necessary, to fall back onto a general-purpose synthesizer) than to simply run such a slow, if not unreliable, general-purpose synthesizer. It is thus worthwhile to study the properties of any sub-class of programs and hardwire its synthesis. Second, as the recent interest in constructive induction [10, 20] shows, necessarily-invented predicates have recursive programs. It is thus worthwhile to study the class of recursive programs, because any learner (even a general-purpose one) can use such a specialized recursion-synthesizer once it has detected, or merely conjectured, the necessity of a new predicate.

Finally, let's assume that our specifier is "lazy," that is s/he doesn't want to take the initiative and type in evidence of the intended concept without knowing whether it will be "useful" to the synthesizer or not. So we need an interactive synthesizer, and even one that takes the initiative and queries the specifier only for strictly necessary evidence. This is actually another solution to the mentioned example voraciousness of many learners. The query and answer languages need to be carefully designed, though, so that even a computationally naive specifier can use the system. For instance, during the synthesis of a sorting program, the specifier cannot be queried about an insertion predicate (assuming the synthesis "goes towards" an insertion-sort program), because this is an auxiliary concept that is not necessarily known to the specifier, her/his "mental" sorting algorithm being not necessarily the insertion-sort one. Also note that such an interaction scenario does not necessarily assume a human specifier.

I plan to combine all of the mentioned ideas into one system. So, in summary, I aim at an interactive, inductive/abductive, schema-guided synthesizer of recursive programs, that takes the initiative and minimally queries a (possibly computationally naive) specifier for evidence in her/his conceptual language.

Example 1. After analyzing my previous work (in a different mindset [9, 10, 12]), I decided on the following target scenario. Assume a (possibly computationally naive) specifier somehow has (an equivalent of) the following informal specification in mind:

`sort(L,S)` iff `S` is a non-decreasing permutation of `L`,
where `L`, `S` are integer-lists.

Now imagine a logic program synthesizer that takes this specifier through the following annotated dialogue, where questions are in **teletype** font, default answers (if any) are between curly braces "{...}", the specifier's actual answers are in *italics*, the comma "," stands for conjunction, and the semi-colon ";" stands for disjunction:

Predicate declaration? $sort(L : list(int), S : list(int))$

If the specifier is ever to use a logic program for `sort`, s/he *must* be able to give such a predicate declaration, because the predicate symbol, the sequence of formal parameters, and their types must be known to her/him. Minimal knowledge about the system, its syntax, and its type system is thus unavoidable.

Induction parameter? {L} L

Result parameter? {S} S

Decomposition operator? {L=[HL|TL]} $L = [HL|TL]$

The last three queries seem to require some programming knowledge (see Section 2 for the terminology), which would go counter a scenario with a computationally naive specifier. However, note that the system proposes default answers, so that such a specifier may indeed ignore these queries by simply accepting their default answers.

What conditions on <S> must hold such that sort([],S) holds? $S = []$

The specifier *must* know what the sorted version of the empty list is, because otherwise s/he wouldn't even have the need for a `sort` program.

What conditions on <A,S> must hold such that sort([A],S) holds? $S = [A]$

Also, the specifier *must* know what the sorted version of the one-element list is.

What conditions on <A,B,S> must hold such that sort([A,B],S) holds? $S = [A, B], A \leq B; S = [B, A], A > B$

Finally, the specifier *must* know *what* the sorted version of a two-element list is, and *why* it is so. The answer may look complicated (due to the use of variables, conjunction, and disjunction), but note that it only embodies minimal knowledge about `sort`, which is *independent* of any sorting algorithms. Note how the specifier was "forced" to use the $\leq/2$ and $>/2$ predicates, as they are essential to the concept of number sorting. Soon afterwards, the system reports:

```
A possible logic program for sort is:
   sort(L,S) ← L=[],S=[]
   sort(L,S) ← L=[HL|TL],sort(TL,TS), insert(HL,TS,S)
   insert(I,L,R) ← L=[],R=[I]
   insert(I,L,R) ← L=[HL|TL],I≤HL, R=[I,HL|TL]
   insert(I,L,R) ← L=[HL|TL],I>HL, insert(I,TL,TR),R=[HL|TR]
```

This is insertion-sort. Obviously (to us), the implicit informal specification of `insert` is as follows:

`insert(I,L,R)` iff `R` is `L` with `I` inserted into it,
where `I` is an integer and `L`, `R` are non-decreasing integer-lists.

But the specifier need not know this. The system doesn't know this either, and actually uses predicate symbol `dpcSort` instead (see the naming scheme in Section 2), as well as other parameter names.

`Do you want another logic program for sort? {yes}` *Yes, please!*
`Backtracking ...`
`Decomposition operator? {L=[HL|T],partition(T,HL,TL1,TL2)}`
$L = [HL|T], partition(T, HL, TL_1, TL_2)$

Assume that `partition` is a system primitive (whose program is known to the system) informally specified as follows:

`partition(L,P,S,B)` iff `S` (resp. `B`) contains the elements of `L` that are smaller than (resp. bigger than or equal to) `P`,
where `L`, `S`, `B` are integer-lists and `P` is an integer.

After a short while, the system announces:

```
A possible logic program for sort is:
   sort(L,S) ← L=[],S=[]
   sort(L,S) ← L=[HL|T], partition(T,HL,TL1,TL2), sort(TL1,TS1),
                  sort(TL2,TS2), combine(HL,TS1,TS2,S)
   combine(E,A,B,C) ← A=[],C=[E|B]
   combine(E,A,B,C) ← A=[HA|TA], combine(E,TA,B,TC),C=[HA|TC]
```

This is the quick-sort program. The implicit informal specification of `combine` is as follows:

`combine(E,A,B,C)` iff `append(A,[E|B],C)`,
where `A`, `B`, `C` are lists and `E` is a term.

Again, the system actually uses predicate symbol `dpcSort` instead of `combine`. Also note that no further queries were posed to the specifier, and yet the system came up with a completely different program.

`Do you want another logic program for sort? {yes}` *Yes, sure!*
`Backtracking ...`
`Decomposition operator? {L=[_,_|_],halves(L,TL1,TL2)}`
$L = [\_, \_|\_], halves(L, TL_1, TL_2)$

Assume that `halves` is a system primitive informally specified as follows:

`halves(L,F,S)` iff `F` is the first half of `L`, and `S` is the second half of `L`, such that `S` has maximum one element more than `F`, where `L`, `F`, `S` are lists.

Soon, the system proclaims:

```
A possible logic program for sort is:
sort(L,S) ← L=[],S=[]
sort(L,S) ← L=[_],S=L
sort(L,S) ← L=[_,_|_],halves(L,TL1,TL2),sort(TL1,TS1),
              sort(TL2,TS2),merge(TS1,TS2,S)
merge(A,B,C) ← C=[],A=[],B=[]
merge(A,B,C) ← C=[HC|TC],merge(TA,TB,TC),TB=[],A=[HC|TA],B=TB
merge(A,B,C) ← C=[HC|TC],merge(TA,TB,TC),TB=[H|_],HC≤H,
                   A=[HC|TA],B=TB
merge(A,B,C) ← C=[HC|TC],merge(TA,TB,TC),TA=[],A=TA,B=[HC|TB]
```

```
merge(A,B,C) ← C=[HC|TC],merge(TA,TB,TC),TA=[H|_],H>HC,
                    A=TA,B=[HC|TB]
```

This is the merge-sort program. The implicit informal specification of `merge` is as follows:

`merge(A,B,C)` iff `C` is the merger of `A` and `B`,
where `A`, `B`, `C` are non-decreasing integer-lists.

The system actually uses the predicate symbol `dpcSort` instead of `merge`. Again note that although no further queries were posed to the specifier, the system produced yet another completely new program.

`Do you want another logic program for sort? {yes}` *No*

This ends the target scenario. ◇

In the remainder of this paper, I first discuss, in Section 2, the notion of logic program schema, and then, in Section 3, I show how such schemata are the key to building the DIALOGS system (Dialogue-based Inductive and Abductive LOGic program Synthesizer), such that it has all the wanted features. The refinement of DIALOGS is incremental, introducing more advanced features only as the need arises and as the basic mechanism is already explained. Finally, in Section 4, I look at related work, outline future work, and conclude.

2 Logic Program Schemata

Programs can be classified according to their synthesis methodologies, such as divide-and-conquer, generate-and-test, top-down decomposition, global search, and so on, or any composition thereof. Informally, a *program schema* consists, first of all, of a *template* program with a fixed dataflow, but without specific indications about the actual computations, except that they must satisfy certain *constraints*, which are the second component of a schema. A program schema thus abstracts a whole family of particular programs that can be obtained by instantiating the place-holders of its template to particular computations, using the program synthesized so far and the specification, so that the constraints of the schema are satisfied. It is therefore interesting to guide program synthesis by a schema that captures the essence of some synthesis methodology. This reflects the conjecture that experienced programmers actually instantiate schemata when programming, which schemata are summaries of their past programming experience. For a more complete treatise on this subject, please refer to my survey [11]. In ILP, for instance, schemata are used as a form of declarative bias by XOANON [22], MOBAL [16], CLINT/CIA [6], GRENDEL [5], SYNAPSE [9, 12], MISST [21], CILP [17], METAINDUCE [15], and others.

For the purpose of illustration only, I will focus on the divide-and-conquer synthesis methodology (which yields recursive programs), and I will restrict myself to predicates of maximum arity 3.

A *divide-and-conquer program* for a predicate `R` over parameters `X`, `Y`, and `Z` works as follows. Assume `X` is the induction parameter, `Y` the (optional) result parameter, and `Z` the (optional) auxiliary parameter. If `X` is minimal, then `Y`

is directly computed from X, possibly using Z. Otherwise, that is if X is non-minimal, decompose (or: *divide*) X into a vector **HX** of hx heads $\mathtt{HX}_i$ and a vector **TX** of t tails $\mathtt{TX}_i$, the tails $\mathtt{TX}_i$ being each of the same type as X, as well as smaller than X according to some well-founded relation. The tails **TX** are recursively associated with a vector **TY** of t tails $\mathtt{TY}_i$ of Y, the auxiliary parameter Z being unchanged in recursive calls (this is the *conquer* step). The heads **HX** are processed into a vector **HY** of hy heads $\mathtt{HY}_i$ of Y, possibly using Z. Finally, Y is composed (or: *combined*) from its heads **HY** and tails **TY**, possibly using Z. For X non-minimal, it is sometimes unnecessary or insufficient (if not wrong) to perform a recursive call, because Y can be directly computed from **HX** and **TX**, possibly using Z. One then has to discriminate between such cases, according to the values of **HX**, **TX**, Y, and Z. If the underlying relation is non-deterministic given X, then such discriminants may be non-complementary. In the non-recursive non-minimal case, several (say v) subcases with different solving operators may emerge; conversely, in the recursive case, several (say w) subcases with different processing and composition operators may emerge: one then has to discriminate between all of these subcases.

Each of the $1+v+w$ clauses of logic programs synthesized by this divide-and-conquer methodology is covered by one of the second-order clause templates of Template 1. Note that an "accidental" consideration of a parameter W as a result parameter rather than as an auxiliary parameter does not prevent the *existence* of a program (but the converse is true): W will be found to be always equal to its tail TW, and post-synthesis transformations can yield the version that would have been synthesized with W being considered as an auxiliary parameter. For convenience, if hx, t, hy, v, or w is particularized to constant 1, then I will often drop the corresponding indices. Also, I will often refer to the predicate variables, or their instances, as *operators*.

```
R(X,Y,Z) ←
    Minimal(X),
    SolveMin(X,Y,Z)
R(X,Y,Z) ←
    NonMinimal(X),
    Decompose(X,HX,TX),              % HX=HX1,...,HXhx
    Discriminatej(HX,TX,Y,Z),        % TX=TX1,...,TXt
    SolveNonMinj(HX,TX,Y,Z)
R(X,Y,Z) ←
    NonMinimal(X),
    Decompose(X,HX,TX),
    Discriminatek(HX,TX,Y,Z),
    R(TX1,TY1,Z),...,R(TXt,TYt,Z),
    Processk(HX,HY,Z),               % HY=HY1,...,HYhy
    Composek(HY,TY,Y,Z)              % TY=TY1,...,TYt
```

Template 1: Divide-and-conquer clause templates ($1 \leq j \leq v$, $v < k \leq v + w$)

The constraints to be verified by first-order instances of this template are

listed elsewhere [11]. The most important one is that there must exist a well-founded relation "<" over the domain of the induction parameter, such that the instance of **Decompose** guarantees that $\mathtt{TX}_i$ "<" X, for every $1 \leq i \leq t$. Other important constraints will be seen in Section 3.2.

Note that, at the logic program level (and at the schema level), I'm not interested in the control flow: these are *not* Prolog programs, and there is complete independence of the execution mechanism.

Example 2. The insertion-sort program of Example 1 is a rewriting of the program obtained by applying the second-order substitution

```
{ R/λA,B,C.sort(A,B), % projection: no auxiliary parameter!
Minimal/λA.A=[], SolveMin/λA,B,C.B=[],
NonMinimal/λA.∃H,T.A=[H|T], Decompose/λA,H,T.A=[H|T],
Discriminate/λH,T,B,C.true,
Process/λA,B,C.B=A, Compose/λH,T,B,C.insert(H,T,B) }
```

to the $\{v/0, w/1, hx/1, t/1, hy/1\}$-particularization of Template 1. This means that there is no non-recursive non-minimal case, and one recursive case, which features decomposition of the induction parameter L into one head, HL, and one tail, TL, the latter giving rise to one tail, TS, of the result parameter S. There is no auxiliary parameter. ◇

Example 3. The **insert** program of Example 1 is a rewriting of the program obtained by applying the second-order substitution

```
{ R/λA,B,C.insert(C,A,B), % re-ordering of formal parameters!
Minimal/λA.A=[], SolveMin/λA,B,C.B=[C],
NonMinimal/λA.∃H,T.A=[H|T], Decompose/λA,H,T.A=[H|T],
Discriminate1/λH,T,B,C.C≤H, SolveNonMin/λH,T,B,C.B=[C,H|T],
Discriminate2/λH,T,B,C.C>H,
Process/λA,B,C.B=A, Compose/λH,T,B,C.B=[H|T] }
```

to the $\{v/1, w/1, hx/1, t/1, hy/1\}$-particularization of Template 1. This means that there is one non-recursive non-minimal case and one recursive case, both featuring decomposition of the induction parameter L into one head, HL, and one tail, TL, the latter giving rise to one tail, TR, of the result parameter R. Auxiliary parameter I is used in the discriminants and in the solving operators, and passed around unchanged in the recursive calls; it is however not used in the process and compose operators of the recursive case. ◇

A more general template is needed to cover the **combine** program of Section 1; it would cover logic programs for n-ary predicates with arbitrary numbers of result parameters and auxiliary parameters. Such a template is actually to be used by any serious implementation of the synthesis mechanism exposed hereafter.

In the following, Template 1 will turn out to have too much information, as we will not be able to distinguish between the instances of the operators in the first two clause templates, nor between the instances of **NonMinimal**, the $\mathtt{Discriminate}_k$, the $\mathtt{Process}_k$, and the $\mathtt{Compose}_k$ in the third clause template: I'll thus unite these operators into $\mathtt{DS}_j$ (with parameters X, Y, Z) and $\mathtt{DPC}_k$ (with

parameters **HX**, **TY** , Y, Z; note that **HY** has disappeared altogether, and that discrimination must now be on **TY**), respectively. Moreover, I will want to identify the predicate, say R, in whose logic program a certain operator appears, and this by just looking at the predicate symbol of that operator: therefore, I'll keep every operator name short and suffix their names by "-R" or "R", at the template level and at the instance level. Since nothing in λ-calculus mechanizes such a naming scheme when moving to the instance level, I will enforce it manually. Also note the convenient naming scheme of the internal variables of each clause: every head or tail of some formal parameter has a name syntactically dependent on the name of that parameter (heads are prefixed by "H" and tails by "T"); this helps tracing the role of each variable. If a predicate is declared by the specifier as r(A,B,C), then I will automatically apply the renaming substitution {X/A, Y/B, Z/C, **HX/HA**, **TX/TA**, **TY/TB**} to instances of the template (assuming A is chosen as induction parameter, B as result parameter, and C as auxiliary parameter), so that the specifier (and reader) can relate to such instances. All this yields Template 2 as a version that is more adequate for my present purposes. I'll refer to instances of its first clause template as *primitive cases*, and to instances of the other one as *non-primitive cases*.

```
R(X,Y,Z) ←
    DS-R_j(X,Y,Z)
R(X,Y,Z)←
    DecR(X,HX,TX)                          % HX=HX_1,...,HX_hx
    R(TX_1,TY_1,Z),...,R(TX_t,TY_t,Z),     % TX=TX_1,...,TX_t
    DPC-R_k(HX,TY,Y,Z)                     % TY=TY_1,...,TY_t
```

Template 2: Divide-and-conquer clause templates ($1 \leq j \leq v$, $1 \leq k \leq w$)

Example 4. The insertion-sort program of Example 1 is a slight rewriting of the program obtained by applying the second-order substitution

{ R/λA,B,C.sort(A,B), DS-R/λA,B,C.A=[],B=[],
DecR/λA,H,T.A=[H|T], DPC-R/λH,T,B,C.dpcSort(H,T,B) }

to the $\{v/1, w/1, hx/1, t/1\}$-particularization of Template 2, provided the first-order renaming substitution {X/L,Y/S,HX/HL,TX/TL,TY/TS} is indeed automatically applied in this process. ◇

Example 5. The **insert** program of Example 1 is a slight rewriting of the program obtained by applying the second-order substitution

{ R/λA,B,C.insert(C,A,B), DS-R$_1$/λA,B,C.A=[],B=[C],
DS-R$_2$/λA,B,C.∃H,T.A=[H|T],B=[C,H|T],C≤H,
DecR/λA,H,T.A=[H|T], DPC-R/λA,B,C,D.∃H,T.B=[H|T],C=[A,H|T],D>A }

to the $\{v/2, w/1, hx/1, t/1\}$-particularization of Template 2. ◇

3 The DIALOGS System

A DIALOGS synthesis is divided into two phases. The first phase performs a *full particularization* of Template 2 (instantiation of *all* its form variables, namely

hx, *t*, *v*, and *w*, which yields a second-order logic program) and an instantiation of *some* of its predicate variables (all except the $\texttt{DS-R}_j$ and the $\texttt{DPC-R}_k$), and is explained in Section 3.1. The second phase performs an instantiation of the $\texttt{DS-R}_j$ and the $\texttt{DPC-R}_k$ (that is the computations constructing the result parameter in each case), and is explained in Section 3.2.

3.1 Full Particularization and Partial Instantiation of the Template

Predicate declaration. DIALOGS first prompts the specifier for a predicate declaration. Assume, without loss of generality, that the specifier answers with a predicate declaration for a ternary predicate, say $\texttt{p(A:T}_1\texttt{,B:T}_2\texttt{,C:T}_3\texttt{)}$, where `p` is a new predicate symbol, `A`, `B`, `C` are different variable names, and the types $\texttt{T}_i$ are in the set {`atom`, `int`, `nat`, `list(_)`,...}. The actual type system is of no importance here, and the reader may guess the meanings of these type names.

Dialogue issues. DIALOGS needs to obtain a full particularization of Template 2. This means that the form variables *hx*, *t*, *v*, and *w* need to be bound to integers. These are technical decisions, but they must be feasible without technical knowledge, because the specifier might be computationally naive or might not even exist (which is an extreme case of naiveté)! Let me explain: the need for a program for `p` might arise during the synthesis/learning of a program that uses `p`, in which case nobody can answer queries phrased in terms of `p`. (Of course, giving a predicate declaration for `p` is always possible.) This situation arises when a synthesizer/learner detects or conjectures the necessity of a new predicate `p`; for instance, a $\texttt{Compose}_k$ operator of a divide-and-conquer program might itself have a recursive program, so the synthesizer could call itself to find this program. So I need to devise a dialogue mechanism, for this first phase, with at least three features: (*i*) the provision of "reasonable" default answers; (*ii*) the runnability in two modes, namely *aloud* (where a computationally naive specifier may simply select the default answers, and any other specifier may answer with personal preferences) and *mute* (where a non-existing specifier is simulated by automatic selection of the default answers), and (*iii*) backtrackability, because there might be several reasonable default answers to certain queries, or because an answer may lead to failure at the second phase.

Choice of the parameter roles. The first step towards particularization of *hx* and *t* is the choice of the roles of the parameters: one of them must be the induction parameter, the others may be either result or auxiliary parameters, if any. Choosing an induction parameter can be done heuristically: any parameter of an inductively defined type such as `nat` or `list(_)` is a good candidate. From the predicate declaration, DIALOGS can create a sequence of potential induction parameters, keep the first one as the (first) default answer, and the remaining ones as default answers upon backtracking. Similarly for the result parameter (if any), which is also likely to be of an inductively defined type: from the remaining parameters (if any), DIALOGS can create a sequence of potential result parameters, keep the first one as the (first) default answer, and the remaining ones as

default answers upon backtracking. Finally, DIALOGS can propose as the auxiliary parameter (if any) the remaining parameter (if any). Note that an auxiliary parameter is likely, but not certain, not to be of an inductively defined type, a good counter-example being `I` of **`insert`**, which is an integer, but has nothing to do with the "inductive nature" of inserting something into a list. Also remember, from Section 2, that an auxiliary parameter may inadvertently be considered as a result parameter, without any influence on the existence of a correct program (but the synthesis is likely to be a bit slower). In the following, I will implicitly drop all occurrences of `Z` in Template 2 in case there is no auxiliary parameter.

Instantiation of `R`. Assuming, without loss of generality, that `B` is chosen as induction parameter, `C` as result parameter, and `A` as auxiliary parameter, DIALOGS can now apply the second-order substitution {`R/λU,V,W.p(W,U,V)`} and the renaming substitution {`X/B,Y/C,Z/A,`**`HX/HB,TX/TB,TY/TC`**} to Template 2, hence (partly) instantiating the heads and the recursive calls of the templates.

Instantiation of **`DecR`** *and particularization of hx and t.* The choice of an instance of **`DecR`** will finally particularize *hx* and *t*. DIALOGS can simply use a type-specific predefined sequence of potential instances of **`DecR`**, keep the first one as the (first) default answer, and the remaining ones as default answers upon backtracking. Assuming induction parameter `B` is of type `list(int)`, the sequence could be

`DecR/λL,H,T.L=[H|T]`	*hx*/1, *t*/1
`DecR/λL,`H_1`,`H_2`,T.L=[`H_1`,`H_2`|T]`	*hx*/2, *t*/1
...	...
`DecR/λL,H,`T_1`,`T_2`.∃T.L=[H|T],partition(T,H,`T_1` ,`T_2`)`	*hx*/1, *t*/2
`DecR/λL,`T_1`,`T_2`.L=[_,_|_],halves(L,`T_1`,`T_2`)`	*hx*/0, *t*/2
...	...

Similar sequences are pre-defined for every type, such that they enforce the well-foundedness constraint.

Particularization of v and w. Definitely the hardest particularization is to decide, in advance, how many subcases there are for each case. A safe approach is to conjecture that there is one primitive case ($v = 1$), as well as one non-primitive case ($w = 1$), and to have the remainder of synthesis refine this: if either of these cases turns out to have subcases, which means that the instance of `DS-R` or `DPC-R` is a disjunctive formula, then set v or w to the number of disjuncts in this instance and rewrite the overall program accordingly.

So far so good. This terminates the first phase: in Template 2, all form variables and all predicate variables except `DS-R` and `DPC-R` are by now instantiated. From a programming point of view, all creative decisions have been taken, but alternative decisions are ready for any occurrence of backtracking (either because some decision leads to failure of the second phase, or because the specifier wants another program after successful completion of the second phase). The remaining instantiations are performed by the second phase, which is discussed in the next subsection.

3.2 Instantiation of the Solving Computations

The instantiation of the remaining predicate variables (namely DS-R and DPC-R) also is interactive and is based on the notions of abduction through (naive) unfolding and querying, and induction through computation of most-specific generalizations (or: least-general generalizations). [1]

Basic principle. In a nutshell, the basic principle is as follows. Assume, for concreteness and simplicity, that the first phase produced the following instantiation of Template 2 (without auxiliary parameter), with list A being the induction parameter, divided by head-tail decomposition, and B being the result parameter:

p(A,B) ← DS-p(A,B)
p(A,B) ← A=[HA|TA],p(TA,TB),DPC-p(HA,TB,B)

The possible computation "traces" for various most-general values of the induction parameter are:

p([],D_1) ← DS-p([],D_1)
p([E_1],F_1) ← DS-p([E_1],F_1)
p([E_1],F_1) ← p([],F_1),DPC-p(E_1,F_1,F_1)
p([G_1,G_1],H_1) ← DS-p([G_1,G_1],H_1)
p([G_1,G_1],H_1) ← p([G_1],H_1),DPC-p(G_1,H_1,H_1)
...

The strategy is to (*a*) query the specifier for an instance of the last atom of each trace, using previous answers to resolve recursive calls, (*b*) inductively infer an instance of DS-p from some of the answers, and (*c*) inductively infer an instance of DPC-p from the other answers. The criterion of how to establish such a partition of the answers follows from the dataflow constraints of the schema (see below).

The specifier *must* know what B is when A is the empty list. A query is generated by instantiating the first clause to

p([],D_1) ← DS-p([],D_1) (1)

Unfolding of second-order atoms is impossible, so the unfolding process stops here. The query

What conditions on <D_0> must hold such that p([],D_0) holds?

can be extracted from this clause. The answer should thus be a formula $\mathcal{F}[D_0]$, where only D_0 may be free, explaining how to compute D_0 from [] such that p([],D_0) holds. In other words, DS-p([],D_0) should be "equivalent" to $\mathcal{F}[D_0]$. Instantiating the second clause when A is the empty list would lead to failure of the unfolding process at the equality atom.

The specifier *must* also know what B is when A has one element. A query is generated by instantiating the second clause to

p([E_1],F_1) ← [E_1]=[HA|TA],p(TA,TB),DPC-p(HA,TB,F_1)

Unfolding the equality atom gives

[1] Term g is *more general than* term s if there is a substitution θ such that $s = g\theta$. We also say that s is *more specific than* g. The *most-specific generalization* (abbreviated msg) of terms a and b is a term m that is more general than both a and b, and such that no term more specific than m (up to renaming) is more general than both a and b. The msg of a non-empty set of terms is defined similarly. See [19] for more details.

p([E_1],F_1) ← p([],TB),DPC-p(E_1,TB,F_1)

Unfolding the p atom, using clause (1) with the newly obtained evidence of DS-p as a "shortcut," gives

p([E_1],F_1) ← F[TB],DPC-p(E_1,TB,F_1)

Recursively unfolding all the atoms in $\mathcal{F}$[TB] eventually reduces this clause to

p([E_1],F_1) ← DPC-p(E_1,tb_1,F_1) (2)

where tb_0 represents the value of TB after this "execution" of $\mathcal{F}$[TB].The query

What conditions on <E_1,F_1> must hold such that p([E_1],F_1) holds?

can be extracted from this clause. The answer should thus be a formula $\mathcal{G}$[E_1,F_1], where only E_1 and F_1 may be free, explaining how to compute F_1 from [E_1] such that p([E_1],F_1) holds. In other words, DPC-p(E_1,tb_0,F_1) should be "equivalent" to $\mathcal{G}$[E_1,F_1]. Instantiating the first clause when A is a one-element list would yield the same query, so we can directly establish that DS-p([E_1],F_1) should also be "equivalent" to $\mathcal{G}$[E_1,F_1].

Next query the specifier for what B is when A has two elements. Again, s/he *must* know the answer. A query is generated by now instantiating the second clause to

p([G_1,G_1],H_1) ← [G_1,G_1]=[HA|TA],p(TA,TB),DPC-p(HA,TB,H_1)

Unfolding the equality atom gives

p([G_1,G_1],H_1) ← p([G_1],TB),DPC-p(G_1,TB,H_1)

Unfolding the p atom, using clause (2) with the newly obtained evidence of DPC-p as a "shortcut," gives

p([G_1,G_1],H_1) ← G[G_1,TB],DPC-p(G_1,TB,H_1)

Recursively unfolding all the atoms in $\mathcal{G}$[G_2,TB] will reduce this clause to

p([G_1,G_1],H_1) ← DPC-p(G_1,tb_1,H_1)

where tb_1 represents the value (possibly using G_2) of TB after this "execution" of $\mathcal{G}$[G_2,TB]. The query

What conditions on <G_1,G_2,H_2> must hold such that p([G_1,G_2],H_2) holds?

can be extracted from this clause. The answer should thus be a formula $\mathcal{H}$[G_1,G_2,H_2], where only G_1, G_2, and H_2 may be free, explaining how to compute H_2 from [G_1,G_2] such that p([G_1,G_2],H_2) holds. In other words, DPC-p(G_1,tb_1,H_2) should be "equivalent" to $\mathcal{H}$[G_1,G_2,H_2]. Instantiating the first clause when A is a two-element list would yield the same query, so we can directly establish that DS-p([G_1,G_2],H_2) should also be "equivalent" to $\mathcal{H}$[G_1,G_2,H_2].

One may continue like this for an arbitrary number of times, gathering more and more evidence of DS-p and DPC-p. As of now, I do not have a clear heuristic for when to stop gathering evidence. The current implementation simply goes through the loop a constant number of times and lets the specifier give "skip" answers (at her/his risk!) when tired or bored. Overcoming this is considered future work. Sooner or later thus, some inductive inference has to be done from this evidence. For example, if $\mathcal{G}$, $\mathcal{H}$, ... are conjunctions of literals (for other situations, see below), then it "often" (see below) suffices to compute the most-specific generalization of an "adequate" subset of the tuple set (considering all predicate symbols and the connectives "," and "¬" as functors)

$\{\langle \mathtt{E}_1, tb_0, \mathtt{F}_1, \mathcal{G}\rangle, \langle \mathtt{G}_1, tb_1, \mathtt{H}_2, \mathcal{H}\rangle, \ldots\}$, say $\langle ha, tb, b, \mathcal{M}\rangle$, and the binding of `DPC-p` to λ`T,U,V.T=`ha`,U=`tb`,V=`b`,`$\mathcal{M}$ can then complete the synthesis of the second clause. Similarly, compute the msg of the "counterpart complementary subset" of the tuple set $\{\langle \mathtt{[]}, \mathtt{D}_0, \mathcal{F}\rangle, \langle \mathtt{[E}_1\mathtt{]}, \mathtt{F}_1, \mathcal{G}\rangle, \langle \mathtt{[G}_1\mathtt{,G}_2\mathtt{]}, \mathtt{H}_2, \mathcal{H}\rangle, \ldots\}$, say $\langle a, b, \mathcal{M}\rangle$, and the binding of `DS-p` to λ`T,U.T=`a`,U=`b`,`$\mathcal{M}$ can then complete the synthesis of the first clause. I call this (and its refinement hereafter) the *MSG Method* [9, 12, 8].

This presentation of the basic principle is of course very coarse, as it sidetracks or leaves open many important issues, which will be discussed next. In any case, notice how query generation and answering actually *abduce* evidence of the still missing operators.

Unfolding issues. In general thus, the principle of query generation is to successively instantiate every clause for most-general values of the induction parameter and to unfold its first-order body atoms (until only a second-order atom remains), so that a query in terms of the target predicate only can be extracted, hiding the fact that the specifier actually has to answer a query about the second-order atom. Answers to previously posed queries are made available during this unfolding process as shortcuts, avoiding thus that the same query is generated twice. Naive unfolding is sufficient here, as I am only interested in the logic, not in the control, of logic programs. Also, I assume there is a system program for every primitive (such as $=/2$).

As usual, unfolding uses *all* applicable clauses (except when shortcuts are available, in which case only the shortcut clauses are used), so that several clauses may result from an unfolding step; unfolding then continues from *all* of these clauses, with the same stopping criterion and the same spawning process. Moreover, it is sometimes unnecessary to recursively unfold until only a second-order atom is left.

Example 6. Both of these phenomena can be illustrated by means of the `delOdds` predicate, which is informally specified as follows:
`delOdds(L,R)` iff `R` is `L` without its odd elements, where `L`, `R` are integer-lists.
Suppose `L` is chosen as induction parameter, which is divided by head-tail decomposition, and `R` is chosen as result parameter. The following first two queries are posed to the specifier:

`What conditions on <`$\mathtt{R}_0$`> must hold such that delOdds(`$[]$`,`$\mathtt{R}_0$`) holds?` $R_0 = []$

`What conditions on <`$\mathtt{A}_1$`,`$\mathtt{R}_1$`> must hold such that delOdds(`$[\mathtt{A}_1]$`,`$\mathtt{R}_1$`) holds?` $odd(A_1), R_1 = []; \neg odd(A_1), R_1 = [A_1]$

Note that the second answer is disjunctive, and that it not only says *how* the result is computed, but also *when/why* it is so. Now, during the generation of the query about what happens when `L` has two elements, the following clauses are obtained after some unfolding:

$\mathtt{delOdds([}B_1\mathtt{,}B_1\mathtt{],}R_1\mathtt{)} \leftarrow \mathtt{odd(}B_1\mathtt{),DPCdelOdds(}B_1\mathtt{,[]\,,}R_1\mathtt{)}$

$\mathtt{delOdds([}B_1\mathtt{,}B_1\mathtt{],}R_1\mathtt{)} \leftarrow \neg\mathtt{odd(}B_1\mathtt{),DPCdelOdds(}B_1\mathtt{,[}B_1\mathtt{]\,,}R_1\mathtt{)}$

Note that the unfolding yielded two clauses (using the shortcuts established from the second query). The primitive predicate `odd` being introduced by the specifier, we need not unfold it. Therefore, the queries

What conditions on <B_1,B_2,R_2> must hold such that delOdds([B_1,B_2],R_2) holds, assuming odd(B_2)?
$odd(B_1), R_2 = []; \neg odd(B_1), R_2 = [B_1]$
What conditions on <B_1,B_2,R_2> must hold such that delOdds([B_1,B_2],R_2) holds, assuming ¬odd(B_2)?
$odd(B_1), R_2 = [B_2]; \neg odd(B_1), R_2 = [B_1, B_2]$

should be extracted: note the new sub-sentences introduced by the keyword **assuming**. ◇

Instantiation of DS-R *and* DPC-R *through the MSG Method.* Above, I wrote that it "often" suffices to compute msgs in order to help instantiate DS-R and DPC-R (in case their evidence involves only conjunctions of literals); so what is the criterion for doing so? And how to choose the "adequate" tuple subsets over which msgs are computed? To answer these, we first have to analyze the dataflow of divide-and-conquer programs in even greater detail than so far, namely *inside* the DS-R and DPC-R operators [9, 12, 8].

Let's start with the discriminate-process-compose operator. Essentially, it is Y that is "constructed from" **HX**, **TY**, and Z. "Constructing" a term "from" others means that its constituents (constants and variables) are taken from the constituents of these other terms; functors can safely be ignored here, due to their "decorative" role in logic programming. For example, in insert(HL,TS,S), which is the DPC-R operator of the insertion-sort program in Section 1, result S is constructed from HL and TS. But we know more: *all* the constituents of **TY** *must* be used for constructing Y or for discriminating between different constructions of Y, because otherwise the recursive computations of **TY** would have been useless; but the constituents of **HX** and Z only *might* be used in this construction of Y. For example, in insert(HL,TS,S), result S is indeed constructed from the "entire" TS, but also from HL; however, in R=[HL|TR], which is the DPC-R operator of the insert program in Section 1, result R is indeed constructed from TR, and from HL, but not from auxiliary parameter I; and there are programs with constructions of Y that involve **TY** and Z but not **HX**, or even only **TY**. Finally: Y can *only* be constructed from the constituents of **HX**, **TY**, and Z, but may not "invent" other constituents, except maybe for the type-specific constants (such as 0, nil, ...), although this is not always the case. All these observations can be gathered in the following definition (which is a particular case of Erdem's version [8], which itself is a powerful and generic extension of my old version [9, 12]): a tuple $\langle \mathbf{hx}, \mathbf{ty}, y, z, \mathcal{F} \rangle$ is *admissible* (for building a discriminate-process-compose operator) iff

$$constituents(\mathbf{ty}) \subseteq constituents(\langle y, \mathcal{F} \rangle) \ \wedge$$
$$constituents(y) \subseteq constituents(\langle \mathbf{hx}, \mathbf{ty}, z \rangle) \cup \{0, nil, \ldots\}$$

where terms **ty**, y, and z are optional, and first-order formula $\mathcal{F}$ is a conjunction of literals without any equality atoms. From such an admissible tuple, we can build an *admissible instance* of DPC-R by binding this predicate variable to λ**T**,**U**,**V**,**W**.**T**=**hx**,**U**=**ty**,**V**=y,**W**=z,$\mathcal{F}$.

Let's continue with the discriminate-solve operator. Essentially, it is Y that is constructed from X and Z. But the constituents of X and Z only *might* be used in this construction of Y. Finally, Y may even "invent" new constituents: I here restrict invented constituents to the type-specific constants (0, nil, ...), although this is not always the case. All these observations can be gathered in the following definition [8]: a tuple $\langle x, y, z, \mathcal{F} \rangle$ is *admissible* (for building a discriminate-solve operator) iff

$$constituents(y) \subseteq constituents(\langle x, z \rangle) \cup \{0, nil, \ldots\}$$

where terms y and z are optional, and first-order formula $\mathcal{F}$ is a conjunction of literals without any equality atoms. From such an admissible tuple, we can build an admissible instance of DS-R by binding this predicate variable to λT,U,V.T=x,U=y,V=z,$\mathcal{F}$.

Admissibility of the instances of the DS-R$_j$ and the DPC-R$_k$ gives us thus other (dataflow) constraints of the divide-and-conquer schema. They are enforced as follows:

1. partition the tuple set for DPC-R into a minimal number of subsets (called *cliques*) of which any two elements have an admissible msg;
2. analyze every such clique: if the msg of the counterpart subset of the tuples for DS-R is also admissible, then delete the clique from the tuples for DPC-R; otherwise delete that counterpart subset from the tuples for DS-R;
3. take the msgs of the remaining cliques for building admissible instances of the DPC-R$_k$, and set w to the number of these cliques;
4. partition the remaining tuple set for DS-R into a minimal number of cliques, build admissible instances of the DS-R$_j$ from their msgs, and set v to the number of these cliques.

This is essentially my old MSG Method [9, 12], but run with the extended definitions of admissibility.

Example 7. The synthesis of delOdds, as started in Example 6, continues as follows. The first answer abduces the following evidence of DSdelOdds (left column) and DPCdelOdds (right column):

1.⟨[],[],true⟩	(not applicable)

The second answer abduces the following evidence of DSdelOdds and DPCdelOdds:

2.⟨[A$_1$],[],odd(A$_1$)⟩	⟨A$_1$,[],[],odd(A$_1$)⟩
3.⟨[A$_1$],[A$_1$],¬odd(A$_1$)⟩	⟨A$_1$,[],[A$_1$],¬odd(A$_1$)⟩

The third and fourth answers abduce the following evidence of DSdelOdds and DPCdelOdds:

4.⟨[B$_1$,B$_2$],[],(odd(B$_1$),odd(B$_2$))⟩	⟨B$_1$,[],[],odd(B$_1$)⟩
5.⟨[B$_1$,B$_2$],[B$_1$],(¬odd(B$_1$),odd(B$_2$))⟩	⟨B$_1$,[],[B$_1$],¬odd(B$_1$)⟩
6.⟨[B$_1$,B$_2$],[B$_2$],(odd(B$_1$),¬odd(B$_2$))⟩	⟨B$_1$,[B$_2$],[B$_2$],odd(B$_1$)⟩
7.⟨[B$_1$,B$_2$],[B$_1$,B$_2$],(¬odd(B$_1$),¬odd(B$_2$))⟩	⟨B$_1$,[B$_2$],[B$_1$,B$_2$],¬odd(B$_1$)⟩

Note that tuples 4 and 5 for DPCdelOdds are just variants of its tuples 2 and 3, respectively; they could thus be eliminated. In fact, DIALOGS detects this during query generation and never even poses the third query to the specifier; the corresponding tuples are non-interactively abduced using the answer to the second query as shortcut. At step (1), the msg of all the tuples for DPCdelOdds is ⟨HL,TR,R,P⟩. Since there is a predicate variable in the fourth slot, namely P, this tuple is not admissible. So we should partition the tuple set into a minimal number of cliques of which any two elements have an admissible msg. A partition into two cliques of three elements each (with tuples 2, 4, 6, and 3, 5, 7, respectively) achieves this, with the following msgs:

⟨[HL|TL],R,P⟩	⟨HL,TR,TR,odd(HL)⟩
⟨[HL|TL],R,Q⟩	⟨HL,TR,[HL|TR],¬odd(HL)⟩

There are no other partitions yielding two cliques. The partitions yielding three to six cliques are obviously uninteresting, as each of their cliques is properly contained in some clique of the bi-partition.

At step (2), the counterpart six pieces of evidence of DSdelOdds can be deleted, because their two msgs (in the left column above) are not admissible (due to the presence of predicate variables).

At step (3), w is set to 2, and $\texttt{DPCdelOdds}_1$ is bound to λT,U,V.T=HL,U=TR, V=TR,odd(HL), while $\texttt{DPCdelOdds}_2$ is bound to λT,U,V.T=HL,U=TR,V=[HL|TR], ¬odd(HL).

At step (4), v is left to be 1, and DSdelOdds is bound to λT,U.T=[],U=[], true, using the only remaining evidence for DSdelOdds. ◇

What if the answers to the queries are not conjunctions of literals? For simplicity, and without loss of power, I restrict the answer language to the connectives *not* ("¬"), *and* (","), and *or* (";"), and I require answers to be in disjunctive normal form, with the variables appearing in the query being implicitly free, all others being implicitly existentially quantified. Therefore, it suffices to break up disjunctive answers into their conjunctions of literals, and to apply the MSG Method. This was actually illustrated in the delOdds example.

Instantiation of DPC-R *through recursive synthesis.* Instantiating DPC-R via the MSG Method assumes that there is a finite non-recursive axiomatization of that operator. But such is not always the case; take for example the insert predicate used in the insertion-sort program in Section 1: its program is recursive and hence not synthesizable through the MSG Method. So another method needs to be devised for detecting and handling such situations of necessary predicate invention [20, 10]. Since the MSG Method has been devised to always succeed (indeed, in the worst case, it partitions a tuple set into cliques of one element each), a heuristic is needed for rejecting the results of the MSG Method and thus conjecturing the necessity of predicate invention. A good candidate heuristic is [9, 8]: if there are "too few" cliques for DPC-R, then reject the results of the MSG Method. The interpretation of "too few" is implementation-dependent, and could

be user-controlled by system-confidence parameters; the current implementation only rejects when w is 0.

Example 8. After the three queries of the insertion-sort synthesis of Example 1 (assuming `L` is chosen as induction parameter, which is divided by head-tail decomposition, and `S` is chosen as result parameter), the abduced tuples for `DSsort` and `DPCsort` respectively are (after some renaming):

⟨[],[],true⟩	(not applicable)
⟨[A_1],[A_1],true⟩	⟨A_1,[],[A_1],true⟩
⟨[B_1,B_2],[B_1,B_2],$B_1 \leq B_2$⟩	⟨B_1,[B_2],[B_1,B_2],$B_1 \leq B_2$⟩
⟨[B_1,B_2],[B_2,B_1],$B_1 > B_2$⟩	⟨B_1,[B_2],[B_2,B_1],$B_1 > B_2$⟩

The MSG Method partitions, at step (1), the three tuples for `DPCsort` into three cliques of one element each; at step (2), these tuples are removed because their counterparts for `DSsort` are admissible as well; at step (3), no evidence is left for `DPCsort`, so w is set to 0; finally, at step (4), the four tuples for `DSsort` are partitioned into three cliques, so v is set to 3. This result is however rejected by the heuristic above: it is conjectured that `DPCsort` cannot be instantiated through the MSG Method (that is, a program for `insert` cannot be found by this way). ◇

So how to proceed? This is a situation of necessary predicate invention, which is precisely one of the situations targeted by DIALOGS, which is a recursion-synthesizer (due to its foundation on Template 2). So the idea is for DIALOGS to re-invoke itself, under the assumption that a divide-and-conquer program exists for the missing operator.

The instantiations done by steps (3) and (4) of the MSG Method need to be undone. The latter is thus revised as follows: steps (3) and (4) only *create* the instances, but the actual bindings are deferred until acceptance by the rejection heuristic.

Using Template 2 and the declaration of the current predicate (see below), the variable `DPC-R` is bound to λ`T,U,V,W.dpcR(T,U,V,W)`, and the predicate declaration `dpcR(H:`T_4`,T:`T_3`,R:`T_3`,A:`T_1`)` is elaborated (assuming that the elements of induction parameter `B:`T_2 are of type T_4, that $hx = t = 1$, and that `C:`T_3 is the result parameter and `A:`T_1 the auxiliary parameter). Indeed, under these assumptions, the call to the new predicate will be `dpcR(HB,TC,C,A)`. Note that this doesn't necessarily create a predicate of maximum arity 3, but, as said earlier, a generalization of Template 2 should be used for any serious implementation. Moreover, the variable `DS-R` is instantiated according to the msgs of the tuples that have no counterparts among the tuples for `DPC-R`. For the insertion-sort synthesis, this gives the declaration `dpcSort(I:int,L:list(int),R:list(int))`, while variable `DSsort` is bound to λ`A,B,C.A=[],B=[]`, and variable `DPCsort` is bound to λ`H,T,B,C.dpcSort(H,T,B)`, just like in Example 4.

The first phase of the sub-synthesis *must* be run in mute mode, as the specifier doesn't know what kind of program the system is synthesizing and therefore can't

be expected to answer queries about its operators, let alone about the operators used in synthesizing these operators.

However, some hints for the first phase of this sub-synthesis could be expressed: in general, it seems reasonable to hint at `T` as induction parameter, `R` as result parameter, and `H`, `A` as auxiliary parameters. A reasonable hint could also be expressed for instantiation of `DecR`, but I do not go into these details here. In any case, these hints beg a fourth feature of the dialogue mechanism (see "Dialogue issues" above), namely: (*iv*) preference of hints (if any) over defaults in mute mode. In general, DIALOGS is thus also called with a possibly empty hint list, rather than with only a predicate declaration.

The second phase of this sub-synthesis should not generate queries about the new predicate. It shouldn't even synthesize a program for the new predicate by explicit induction on the parameter hinted at, because not every value of that induction parameter is "reachable" by values of the induction parameter of the super-synthesis: queries about the new predicate can't always be formulated in terms of the old one. For example, a factorial program needs to invent a multiplication predicate, but actually only uses a sparse subset of the multiplication relation [17]. The "trick" to make DIALOGS generate queries about the top-level predicate (see below) such that the answers actually pertain, unbeknownst to the specifier, to that new predicate is quite simple: the first phase of the sub-synthesis should *add* the obtained clauses to those of the super-synthesis, rather than work with these new clauses only.

Thus, in general, DIALOGS is called with a *start program* as an additional argument: this is the empty set in the case of a new synthesis (for the *top-level predicate*), or a set of clauses for a (unique) *top-level predicate* and its (directly or indirectly) used predicates, in case DIALOGS is used (possibly by itself) for a necessary invention of a predicate that is (directly or indirectly) used by the top-level predicate. The first phase gets a predicate declaration for the *current predicate* and builds the *current program* by adding the new clauses to the start program. Query generation in the second phase is always done for the top-level predicate, but unfolding will eventually "trickle down" to a missing operator of the current predicate and extract a question for it in terms of the top-level one. The answers to queries help instantiate a missing operator of the current predicate, through either the MSG Method or further recursive synthesis.

Example 9. Let's continue the synthesis of the insertion-sort program (from Example 1 and Example 8). DIALOGS calls itself recursively in mute mode with

```
sort(L,S) ← L=[],S=[]
sort(L,S) ← L=[HL|TL],sort(TL,TS),dpcSort(HL,TS,S)
```

as start program, `sort` as top-level predicate, `dpcSort(I:int,L:list(int), R:list(int)` as declaration for current predicate `dpcSort`, parameter `L` as preferred induction parameter, parameter `R` as preferred result parameter, and parameter `I` as preferred auxiliary parameter. Assume the first phase builds the current program by adding to the start program the following clauses:

```
dpcSort(I,L,R) ← DSdpcSort(I,L,R)
dpcSort(I,L,R) ← L=[HL|TL],dpcSort(I,TL,TR),DPCdpcSort(HL,TR,R,I)
```

In the second phase, query generation for most-general one-element and two-element lists as induction parameter L of the top-level predicate **sort** leads, without interaction (due to the second and third queries of the super-synthesis), to the following tuples for **DSdpcSort** and **DPCdpcSort**, respectively:

$\langle A_1, [], [A_1], \mathtt{true}\rangle$	(not applicable)
$\langle B_1, [B_2], [B_1, B_2], B_1 \leq B_2\rangle$	$\langle B_2, [B_1], [B_1, B_2], B_1, B_1 \leq B_2\rangle$
$\langle B_1, [B_2], [B_2, B_1], B_1 > B_2\rangle$	$\langle B_2, [B_1], [B_2, B_1], B_1, B_1 > B_2\rangle$

This is scanty evidence to continue from, so one could decide to generate a query about what happens when induction parameter L of the top-level predicate **sort** has three elements. This would yield an extension to the target scenario of Example 1; the ensuing computations are too long to reproduce here, but they eventually lead to the correct binding (just as in Example 5) of $\mathtt{DSdpcSort}_1$ to $\lambda A,B,C.\, A=[],B=[C]$, of $\mathtt{DSdpcSort}_2$ to $\lambda A,B,C.\, \exists H,T.\, A=[H|T],B=[C,H|T],C\leq H$, and of **DPCdpcSort** to $\lambda A,B,C,D.\, \exists H,T.\, B=[H|T],C=[A,H|T],D>A$. Note that v is 2, and w is 1. A more "daring" move would be to directly infer these instances from the tuples above, and thus to stay within the targeted scenario. Indeed, the first tuple can directly lead to the instantiation of $\mathtt{DSdpcSort}_1$, based on the observation that there is no counterpart evidence of **DPCdpcSort**; the second tuple can directly lead to the instantiation of $\mathtt{DSdpcSort}_2$ (by generalization of constant **nil** to a variable), based on the observation that the counterpart evidence of **DPCdpcSort** forces the "breaking up" of the second parameter in order to construct the third one; conversely, the third tuple can directly lead to the instantiation of **DPCdpcSort** (by generalization of constant **nil** to a variable), based on the observation that the counterpart evidence of **DSdpcSort** forces the "breaking up" of the second parameter in order to construct the third one. Formalizing this, and hence reducing dialogues, is considered future work. ◇

A high-level DIALOGS algorithm can be found in the Appendix.

4 Conclusion

In this paper, I have first motivated and then incrementally reconstructed the reasoning that led to the design of the DIALOGS system, which is a dialogue-based, inductive/abductive, schema-guided synthesizer of recursive logic programs, that takes the initiative and minimally queries a (possibly computationally naive) specifier for evidence in her/his conceptual language. DIALOGS can be used by any learner (including itself) that detects, or merely conjectures, the necessity of invention of a new predicate.

Queries are kept entirely in terms of the specifier's conceptual language, and are simple, because they only ask what "happens" when some parameter has a finite number of "elements." Even better, the specifier *must* know the answers to such queries, because otherwise s/he wouldn't even feel the need for the synthesized program. Answers are thus also in the specifier's conceptual language, and are *independent* of the synthesized program. Answers are stored

so that synthesis can proceed with minimal querying. Indeed, a query can be generated more than once, albeit with different "intentions" (that is, aiming at gathering evidence of different operators): the aimed-at operators are either the ones of the top-level predicate or the ones of the current predicate (when the top-level predicate needs to invent the current predicate).

A *competent specifier assumption* only holds in the second phase, because of the backtrackability feature of the dialogue in the first phase: the specifier (if any!) can answer just about anything during the first phase, because wrong answers will lead to failure in the second phase.

Note the elegant ways by which DIALOGS avoids the "background knowledge re-use bottleneck" [13]: first, it only tries to re-use the $=/2$ primitive (by the MSG Method); moreover, other primitives (such as $\leq/2$ or `odd`) used by the specifier in answers to queries end up in the synthesized program (which prevents the sometimes automa-g-ic flavor of inductive synthesis); finally, the system re-uses the primitives occurring in its knowledge base for `DecR`. Overall thus, these primitives do not "compete" in re-use situations.

Due to its foundation on an extremely powerful codification of a "recursion-theory" (by means of the template and constraints of a divide-and-conquer schema), the current prototype implementation needs very little evidence and is very fast. An even faster and more powerful implementation is planned.

The time-complexity of synthesis is essentially linear in the complexity of the synthesized program, due to the repeated unfolding of the synthesized program for various most-general values of some parameter. Steps (1) and (4) of the MSG Method amount to partitioning a graph into a minimal number of cliques, which is known to be an NP-complete problem; however, this should not be an issue, as the graphs under investigation only have a few nodes.

The class of synthesizable programs is a subset of the class of divide-and-conquer programs. It seems to depend on the knowledge base for `DecR`, but a "Devil's Advocate" argument against its completeness with respect to that class may be countered by appealing to the ingenuity of a non-naive specifier when answering the `DecR` question. The current (relaxable) assumptions are that `DS-R` is non-recursively defined, and that `DPC-R` has a divide-and-conquer instance, *if* a new predicate needs to be invented for it.

DIALOGS falls into the category of *trace-based inductive synthesizers* [9] (such as [3], GRENDEL [5], SYNAPSE [9, 12], METAINDUCE [15], CILP [17], ...), because it first explains its examples in terms of computation traces (that fit a certain template), and then generalizes these traces into a recursive program. The main innovation here is that DIALOGS generates its own, generalized examples. Note that SPECTRE [4] and TRACY [2] are *not* trace-based synthesizers, as they don't construct their candidate clauses in a truly schema-guided way. However, they do use a form of declarative bias to enumerate and analyze (that is, accept or reject) potential clauses, and they also feature unfolding/resolution in the process of verifying the coverage of examples.

DIALOGS is most closely related to SYNAPSE [9, 12]: this non-interactive schema-guided inductive/abductive synthesizer expects some positive (ground)

examples as well as Horn clause equivalents (called *properties*) of at least the answers that DIALOGS would query for. In other words, DIALOGS is a simplification of SYNAPSE, without any loss of power, but with less burden on the specifier and with faster synthesis. The Proofs-as-Programs Method (which should have been called Abductive Method) of SYNAPSE has disappeared, as it has become the driving synthesis mechanism of the second phase of DIALOGS.

The CILP [17] and METAINDUCE [15] systems essentially feature subsets of the functionality of SYNAPSE and DIALOGS, in the sense that they have only examples as input language, rely on a simpler divide-and-conquer schema, and use less powerful MSG Methods, which cannot infer disjunctively defined operators.

The CLINT [6] and CLINT/CIA systems [7], although they are *model-based inductive synthesizers* [9], are also related to DIALOGS, in the sense that they are also interactive, sometimes guided by (mono-clausal) templates, and have an extended evidence language (full clauses, called *integrity constraints*). However, these integrity constraints are not used constructively during a synthesis, but only to accept or reject candidate programs.

As said before, a stopping criterion for the dialogue loop of the second phase needs to be identified. Co-routining the abduction, induction, and evaluation steps of that phase seems an approach towards this, as the loop can then be exited when the msgs stop changing.

Future work will also aim at increased schema independence (it's already largely achieved in the second phase, except for the hardwired verification of the constraints), at least via the coverage of an even more powerful divide-and-conquer schema (with support of compound induction parameters, ...) and of other schemata (tupling generalization [14], descending generalization [14], ...). Ideally, the schema would be a parameter of the system, and thus constitute a real declarative bias.

Another plan is to integrate DIALOGS with a post-synthesis transformation/optimization tool; the preference will of course go to using schema-guided transformers [14], as these can exploit much of the additional information (such as "what is the instance of each operator?") generated by DIALOGS.

Acknowledgments

Many thanks to Esra Erdem for numerous stimulating discussions about the MSG Method. The anonymous reviewers were constructive in suggesting some improvements of the presentation. Esra Erdem, Halime Büyükyıldız, and Serap Yılmaz provided useful feedback on an earlier version of this paper, and contributed to the implementation of a first prototype of the DIALOGS system, as well as to the ordeal of typesetting this document in LaTeX.

References

1. Angluin, D.: Queries and concept learning. *Machine Learning* 2(4):319–342, 1988.
2. Bergadano, F., Gunetti, D.: Learning clauses by tracing derivations. In S. Wrobel (ed), *Proc. of ILP'94*, pp. 11–29. GMD-Studien Nr. 237, Sankt Augustin, 1994.

3. Biermann, A.W.: Dealing with search. In A.W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 375–392. Macmillan, 1984.
4. Boström, H., Idestam-Almquist, P.: Specialization of logic programs by pruning SLD-trees. In S. Wrobel (ed), *Proc. of ILP'94*, pp. 31–48. GMD-Studien Nr. 237, Sankt Augustin, 1994.
5. Cohen, W.C.: Compiling prior knowledge into an explicit bias. In *Proc. of ICML'92*, pages 102–110. Morgan Kaufmann, 1992.
6. De Raedt, L., Bruynooghe, M.: Belief updating from integrity constraints and queries. *Artificial Intelligence* 53(2-3):291–307, February 1992.
7. De Raedt, L., Bruynooghe, M.: Interactive concept learning and constructive induction by analogy. *Machine Learning* 8:107–150, 1992.
8. Erdem, E.: *An MSG Method for Inductive Logic Program Synthesis.* Senior Project Final Report, Bilkent University, Ankara (Turkey), May 1996.
9. Flener, P.: *Logic Program Synthesis from Incomplete Information.* Kluwer, 1995.
10. Flener, P.: *Predicate Invention in Inductive Program Synthesis.* TR BU-CEIS-9509, Bilkent University, Ankara (Turkey), 1995. Submitted.
11. Flener, P.: *Synthesis of Logic Algorithm Schemata.* TR BU-CEIS-96xx, Bilkent University, Ankara (Turkey), 1996. Update of TR BU-CEIS-9502. In preparation.
12. Flener, P., Deville, Y.: Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation* 15(5–6):775–805, May/June 1993.
13. Flener, P., Popelínský, L.: On the use of inductive reasoning in program synthesis. In L. Fribourg and F. Turini (eds), *Proc. of META/LOPSTR'94.* LNCS 883:69–87, Springer-Verlag, 1994.
14. Flener, P., Deville, Y.: *Logic Program Transformation through Generalization Schemata.* TR BU-CEIS-96yy, Bilkent University, Ankara (Turkey), 1996. In preparation. Extended abstract in M. Proietti (ed), *Proc. of LOPSTR'95.* LNCS 1048:171–173, Springer-Verlag, 1996.
15. Hamfelt, A., Fischer-Nilsson, J.: Inductive metalogic programming. In S. Wrobel (ed), *Proc. of ILP'94*, pp. 85–96. GMD-Studien Nr. 237, Sankt Augustin, 1994.
16. Kietz, J.U., Wrobel, S.: Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 335–359. Volume APIC-38, Academic Press, 1992.
17. Lapointe, S., Ling, C., Matwin, S.: Constructive inductive logic programming. In S. Muggleton (ed), *Proc. of ILP'93*, pp. 255–264. TR IJS-DP-6707, J. Stefan Institute, Ljubljana (Slovenia), 1993.
18. Muggleton, S., Buntine, W.: Machine invention of first-order predicates by inverting resolution. In *Proc. of ICML'88*, pages 339–352. Morgan Kaufmann, 1988.
19. Plotkin, G.D.: A note on inductive generalization. In B. Meltzer and D. Michie (eds), *Machine Intelligence* 5:153–163. Edinburgh University Press, 1970.
20. Stahl, I.: *Predicate invention in ILP: An overview.* TR 1993/06, Fakultät Informatik, Universität Stuttgart (Germany), 1993.
21. Sterling, L.S., Kirschenbaum, M.: Applying techniques to skeletons. In J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 127–140. John Wiley, 1993.
22. Tinkham, N.L.: *Induction of Schemata for Program Synthesis.* Ph.D. Thesis, Duke University, Durham (NC, USA), 1990.

Appendix: The DIALOGS Algorithm

```
% Interactive synthesis of a recursive (divide-and-conquer) pgm.
dialogs <-
  set interaction mode to `aloud´,
  read(PredDecl), % declaration for r, the top-level predicate
  Hints = {},
  StartPgm = {},
  dialogs(PredDecl,Hints,StartPgm,Pgm),
  write(Pgm),
  if the specifier wants more programs then fail else true.

% Synthesis (in case of detected or conjectured necessary
% predicate invention) of a recursive (divide-and-conquer) pgm
% for the current predicate declared in PredDecl, using Hints
% (if any), which program is used by and thus added to the
% context program StartPgm to yield the final program Pgm.
dialogs(PredDecl,Hints,StartPgm,Pgm) <-
 % phase 1
  NewClauses = a set of divide-and-conquer clauses (according to
    Template 2) for the current predicate (which is declared in
    PredDecl), where only the DecR operator has been instantiated,
    according to Hints (if any),
  CurrPgm = StartPgm union NewClauses,
 % phase 2
  abduce(CurrPgm,DSev,DPCev),
  induce(DSev,DPCev,DSinsts,DPCinsts),
  evaluate(DSinsts,DPCinsts,CurrPgm,Pgm).

% Interactive abduction of evidence sets DSev and DPCev for the
% uninstantiated operators DS-R and DPC-R in program Pgm.
abduce(Pgm,DSev,DPCev) <-
  DSev = {},  DPCev = {},     % initializations
  as often as ``needed´´ do
    construct Goal,  % a goal for the top-level predicate
    demo(Pgm,Goal,Assumptions,Residue),
    ask(Goal,Assumptions,Residue,DS-exs,DPC-exs),
    DSev = DSev union DS-exs,
    DPCev = DPCev union DPC-exs
  od.

% An SLD-refutation of <- Goal in theory Pgm (augmented with
% shortcut clauses from previous queries) generates the conj
% Assumption, but is blocked by the unresolvability of the
% unit-goal Residue, because it has a predicate variable.
demo(Pgm,Goal,Assumption,Residue) <- ...
```

```
% The set DS-exs (resp. DPC-exs) contains the tuples for the
% predicate variable DS-R (resp. DPC-R) in second-order atom
% Residue, which tuples are extracted from the answer (by the
% specifier or by an oracle based on previous answers) to the
% query under what conditions atom Goal must hold, assuming
% that conjunction Assumption holds.
ask(Goal,Assumption,Residue,DS-exs,DPC-exs) <- ...

% Inductive generalization of the evidence sets DSev and DPCev
% into lists of ``plausible'' instances (according to the
% admissibility criteria) DSinsts and DPCinsts for the operators
% DS-R and DPC-R.
induce(DSev,DPCev,DSinsts,DPCinsts) <-      % revised MSG Method
  partition(DPCev,DPCcliques),              % step 1 (as in text)
  prune(DPCcliks,NewDPCcliks,DSev,NewDSev), % step 2 (as in text)
  buildInsts(NewDPCcliks,DPCinsts),         % step 3 (revised)
  partition(NewDSev,DScliks),               % step 4 (revised)
  buildInsts(DScliks,DSinsts).              % step 4 (cont'd)

% Heuristic-based acceptance or rejection of the induced
% instances DSinsts and DPCinsts for the uninstantiated operators
% DS-R and DPC-R in second-order logic program CurrPgm, so as to
% instantiate the latter into a first-order program Pgm.
evaluate(DSinsts,DPCinsts,CurrPgm,Pgm) <-
  if #DPCinsts=0 then  % reject!
    construct NewPredDecl    % decl. for dpcR, the new curr. pred.
    construct NewHints,      % hints for dpcR
    in the last two clauses of CurrPgm do
      instantiate the DS-Rj (as described in text),
      particularize v accordingly,
      instantiate DPC-R to dpcR,
      particularize w to 1
    yielding NewStartPgm,
    set interaction mode to `mute',
    dialogs(NewPredDecl,NewHints,NewStartPgm,Pgm) % recursion!
  else   % accept!
    in the last two clauses of CurrPgm do
      particularize v to #DSinsts,
      for 1<=j<=v do instantiate DS-Rj using DSinsts[j],
      particularize w to #DPCinsts,
      for 1<=k<=w do instantiate DPC-Rk using DPCinsts[k]
    yielding Pgm.
```