Predicate Invention in Inductive Program Synthesis

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

In Inductive Logic Programming, predicate invention is the process of introducing a hitherto unknown predicate, and its description, into the description of the currently learned predicate. This is only necessary when a finite axiomatization of the current predicate is otherwise impossible, in which case the description of the invented predicate is recursive. So necessary predicate invention is a program synthesis task, and had thus best be done by a synthesizer rather than by a general purpose learner, because one can then re-use the vast body of knowledge about recursive algorithm design. Taking a schema-guided approach, I show how predicate invention can be performed (if not avoided) by intelligent re-use from structured background knowledge, by necessarily successful intelligent dialogue with the user, and by structural or computational generalization of the original problem.

1 Introduction

During the learning of a hypothesis H for some positive evidence E^+ and negative evidence E^-, in the presence of some background knowledge B, the expression *concept invention* designates the processes of (*i*) introducing into H some concept(s) that do not appear in E^+, E^-, or B, and (*ii*) learning hypotheses for these new concepts. This corresponds to the usage of *constructive rules* of inductive inference (where the inductive consequent may involve symbol(s) that are not in the antecedent), as opposed to selective ones. Such constructive induction thus doesn't (simplistically) assume that the preliminary learning tasks of representation and vocabulary choice have already been solved, and represents thus a crucial field in learning.

I will here explore some issues around concept invention in the setting of machine learning where the hypotheses (concept descriptions), evidence, and background (prior) knowledge are expressed in (some subset of) first-order logic. This is commonly referred to as Inductive Logic Programming (ILP). A concept is then represented by a predicate-symbol (abbreviated predicate hereafter), and described by a logic program. I will ignore for a while the notion of learning bias.

Let's first have a look at two introductory examples, some arising questions, and the related work, before I make my objectives more explicit.

Example 1: Assume the following positive evidence for the *sort*/2 relation (where *sort*(*L*,*S*) holds iff integer-list *S* is a non-decreasingly ordered permutation of integer-list *L*):

```
sort([],[])
sort([0],[0])
sort([2,1],[1,2])
sort([4,3,5],[3,4,5])
```

Assume there is no negative evidence, nor any background knowledge. Then the learning of the following logic program for *sort*/2:

```
sort([],[])
sort([HL|TL],S) ←
   sort(TL,TS),
   insert(H,TS,S)
```

involved the invention of the *insert*/3 predicate (where *insert*(*I,L,R*) holds iff integer-list *R* is non-decreasingly ordered integer-list *L* with integer *I* inserted), whose logic program hereafter is a by-product:

```
insert(I,[],[I])
insert(I,[HL|TL],[I,HL|TL]) ←
   I≤HL
insert(I,[HL|TL],[HL|TR]) ←
   ¬(I≤HL),
   insert(I,TL,TR)
```

Note that this required in turn the invention of the ≤/2 predicate (whose specification and program are omitted here). Let's ignore for a while that this learning session would be a quite sensational result. ♦

Example 2: Assume the following positive and negative evidence of the *grandDaughter*/2 relation (where *grandDaughter*(*G,P*) holds iff person *G* is a grand-daughter of person *P*):

```
grandDaughter(ayşe,murat)
grandDaughter(bulut,lâle)
¬grandDaughter(ali,murat)
```

and the following background knowledge (where *parent*(*P,Q*) holds iff person *P* is a parent of person *Q*, *female*(*P*) holds iff person *P* is female, and *male*(*P*) holds iff person *P* is male):

```
parent(murat,birtane)        female(birtane)        male(murat)
parent(lâle,sibel)           female(lâle)           male(ali)
parent(birtane,ali)          female(sibel)
parent(birtane,ayşe)         female(ayşe)
parent(sibel,bulut)          female(bulut)
```

Then the learning of the following logic program for *grandDaughter*/2:

```
grandDaughter(G,P) ←
   parent(P,Q),
   daughter(G,Q)
```

involved the invention of the *daughter*/2 predicate (where *daughter*(*D,P*) holds iff person *D* is a daughter of person *P*), whose logic program hereafter is a by-product:

```
daughter(D,P) ←
   parent(P,D),
   female(D)
```

Note that no other predicate needed to be invented in turn. ♦

Let's now have a look at a few issues around predicate invention.

When Is Predicate Invention Necessary? In Example 1, once synthesis was committed to the recursive call *sort*(*TL,TS*), where *TL* is the tail of *L*, the predicate *insert*/3 *had to* be invented, especially that its program *cannot* be unfolded into the program for *sort*/2. If committed to some other recursive call(s), another predicate would *have to* be invented. Otherwise, the background knowledge being empty, *sort*/2 would have to be implemented at most in terms of itself only, which is impossible without generating the non-terminating program *sort*(*L,S*) ← *sort*(*L,S*), or without generating an infinite program. In Example 2, the invention of *daughter*/2 was not necessary, as its program can simply be unfolded into the program for *grandDaughter*/2. However, the *daughter*/2 concept is still interesting in its own right. Generally speaking:

Definition 1: Predicate invention is *necessary* iff there is no finite logic program for the observational concepts in the (positive and negative) evidence that uses only the fixed vocabulary of predicates from the evidence and the background knowledge.

However, detecting this is an undecidable question [23], so heuristics are needed in general. Fortunately, a theorem by Kleene shows that appropriate necessary predicate invention always achieves finite axiomati-

zation (see [23]). Necessarily-invented predicates are always recursively defined, because otherwise their programs could be unfolded. Moreover, I propose the following notion of usefulness (unrelated to Muggleton's utility [18], which is about search space pruning during predicate invention):

Definition 2: Given a partially constructed logic program for the observational concepts in the evidence, predicate invention is *useful* iff there is a way to complete the program by inventing a predicate whose logic program is recursive.

(It is up to the reader to decide whether this concept introduction was necessary, or useful, or neither!) For instance, if the background knowledge of Example 1 included programs for *permutation*/2 and *ordered*/1, then predicate invention would *not* be necessary (as naive-sort can be learned). But, once committed to the recursive call *sort*(*TL,TS*), where *TL* is the tail of *L*, the invention of *insert*/3 is useful, because it avoids having to implement the functionality of *insert*/3 in terms of *permutation*/2 and *ordered*/1 (which would wreak havoc on the computational complexity of the logic program under construction). In Example 2, the invention of *daughter*/2 was neither necessary nor useful, but "interesting." Usefully-invented predicates are thus also recursively defined, which again prevents the unfolding of their programs.

Why Is Predicate Invention Difficult? In Example 1, once committed to the recursive call *sort*(*TL,TS*), where *TL* is the tail of *L*, the implicit evidence for the necessary and useful predicate *insert*/3 is as follows:

```
insert(0,[],[0])
insert(1,[],[1])            % abduced via oracle
insert(2,[1],[1,2])
insert(5,[],[5])            % abduced via oracle
insert(3,[5],[3,5])         % abduced via oracle
insert(4,[3,5],[3,4,5])
```

The problem is that this evidence seems quite insufficient to learn programs for both *insert*/3 itself and the then necessary and useful predicate $\leq$/2. In general, the implicit evidence tends to be quite sparse [17], unless a lot of evidence was given for the given initial concepts.

What Are the Standard Techniques of Predicate Invention? The CLINT-CIA learner [4] performs constructive induction by analogy: previously learned clauses are abstracted into second-order clause schemata that are then used to speed up future learning sessions (by initially restricting the focus to clauses that are instances of known schemata) as well as to invent new predicates (by trying to group learned conjunctions of literals according to bodies of schemata and asking the teacher to name, if possible, the so defined concept by a predicate). However, this technique *cannot* perform useful or necessary predicate invention, and aims thus "merely" at more compact or comprehensible hypotheses.

In specific-to-general learning, predicate invention (*e.g.*, [2], INDEX [7], CIGOL [19], ITOU [21], and others surveyed in [23]) is usually performed by applying the *W*-operators of inverse resolution [19]. However, the *W*-operators *cannot* produce recursive clauses for the invented predicates, so they *cannot* perform useful predicate invention by themselves. To do so, the *V*-operators of inverse resolution may then be invoked, or the teacher may be asked to rename the invented predicate by a known one. Moreover, these operators of inverse resolution induce a very large hypothesis space, so that only some form of "disciplined" search would have any chance of yielding "good" hypotheses in reasonable time (but see [18]). Finally, inverting resolution is not powerful enough, as the abduced pieces of evidence are often more than one resolution step apart. Generally speaking now: inverse resolution is not complete with respect to induction.

The CILP system [17] therefore uses inverse implication instead to invent predicates. A similar technique, applicable in general-to-specific learning, is performed by SYNAPSE [8,10–12], SIERES [25], and METAINDUCE [15]: when the existence of necessary or useful predicate invention is detected (or merely conjectured), then explicitly abduce the corresponding evidence and call the learner recursively from that evidence (I will later refer to this as the *Synthesis Method*). Better: unless the "current" learner is itself specialized to learning recursive programs, call such a specialized learner instead of calling the "current" general purpose learner recursively, because the specialized one can take into advantage the knowledge (or conjecture) that a recursive program has to be learned. The mentioned SYNAPSE, ITOU, CILP, SIERES, and METAINDUCE systems are such specialized learners, and they all rely on some recursive program schema [9] to achieve better performance on the class of recursive programs than general purpose learners. They are themselves recursively defined, and may be called by general purpose learners for necessary or useful predicate invention.

Objectives of this Paper. I can now state my objectives more precisely: I will here focus on the process of useful predicate invention in a very relevant niche [13] of ILP, namely the learning of recursive logic programs. By *computational concepts* I will mean concepts that have (among others) recursive[1] logic programs as concept descriptions; and by (*inductive*) *synthesis* I will mean the learning of recursive logic programs for computational concepts. The *sort*/2 relation is a computational concept, as it can be implemented by recursive logic programs, such as insertion-sort, merge-sort, quick-sort, and so on. It can actually also be implemented by non-recursive logic programs, such as naive-sort, which is however written in terms of some computational concepts. For computational concepts, non-recursive programs tend to be very naive and inefficient, that is they are the kind of formal specifications often provided for deductive synthesis. But the *grandDaughter*/2 concept is non-computational: I, at least, cannot imagine a recursive description of grand-daughters (whose recursive clauses are non-redundant with the non-recursive ones). Synthesis is thus the branch of learning that yields programs that actually compute something, whereas the learning of non-recursive programs (be it for computational concepts or not) yields programs that "merely" classify data. A *specifier* is a teacher who teaches computational concepts. A s*ynthesizer* is a learner that can only synthesize recursive logic programs for computational concepts. Examples of inductive synthesizers are the aforementioned SYNAPSE, ITOU, CILP, SIERES, and METAINDUCE. Their applicability as predicate inventors for general purpose learners further strengthens my former arguments [13] for their being a relevant niche of ILP, in addition to their relevance to (inductive) software engineering in general.

In this paper, I will show that the knowledge (or mere conjecture) that there *is* a recursive program for the invented predicate gives us some extra leverage over conventional methods of predicate invention, precisely because we may appeal to the vast body of knowledge generated by recursive algorithm design research. In the following, I will assume that we try to synthesize a recursive program, regardless of whether this synthesis task arose from a useful or necessary predicate invention in a general purpose ILP task or from another synthesis task. In any case, the synthesis of a recursive program is likely to generate its own useful or necessary predicate inventions, so I cover the whole spectrum of possibilities.

The remainder of this paper is then organized as follows. In Section 2, I briefly introduce the concept of logic program schema, which is crucial to the efficient guidance of synthesis and to the reader's understanding of the remaining sections of this paper. Section 3 shows various techniques of performing predicate invention through re-use, including the reliance on structured background knowledge (as opposed to the traditionally unstructured background knowledge in ILP). In Section 4, I argue that predicate invention can also be performed by mere queries to the specifier. Section 5 discusses problem generalization techniques and shows whether they can be used to perform predicate invention. Finally, in Section 6, I conclude on predicate invention in inductive program synthesis.

2 Logic Program Analysis via 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* is a template program with a fixed control and data flow, but without specific indications about the actual parameters or the actual computations, except that they must satisfy certain constraints. A program schema thus abstracts a whole family of particular programs that can be obtained by instantiating its place-holders to particular parameters or computations, using the specification, the program synthesized so far, and the constraints of the schema. It is therefore interesting to guide program synthesis by a schema that captures the essence of some 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 [9].

In this section, I will introduce a schema for divide-and-conquer logic programs (Section 2.1), then argue for schema-guided synthesis (Section 2.2), and finally identify the place-holders of the divide-and-conquer schema whose instantiations are most likely to require useful predicate invention (Section 2.3).

1. Recursion being the only "looping" construct in logic programming, this discussion would have to be broadened in (structured) imperative programming, so as to encompass for/while/repeat iteration.

2.1 A Divide-and-Conquer Logic Program Schema

In this sub-section, 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, for pedagogical purposes only, to binary predicates. A *divide-and-conquer program* for a binary predicate R over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is (usually) easily found by directly solving the problem. Otherwise, that is if X is non-minimal, decompose (or: divide) X into a vector $\boldsymbol{HX}$ of h heads of X and a vector $\boldsymbol{TX}$ of t tails of X, the tails being of the same type as X, as well as smaller than X according to some well-founded relation. The t tails $\boldsymbol{TX}$ recursively yield t tails $\boldsymbol{TY}$ of Y (this is the conquer step). The heads $\boldsymbol{HX}$ are processed into a vector $\boldsymbol{HY}$ of h' heads of Y. Finally, Y is composed (or: combined) from its heads $\boldsymbol{HY}$ and tails $\boldsymbol{TY}$. Suppose m subcases with different processing and composition operators emerge: one discriminates between them according to the values of $\boldsymbol{HX}$, $\boldsymbol{TX}$, and Y. The m+1 clauses of logic programs synthesized by this divide-and-conquer methodology are covered by the second-order clause schemata of Schema 1, where $\boldsymbol{R(TX,TY)}$ stands for the conjunction of the $R(TX_j,TY_j)$, for $1 \leq j \leq t$.

```
R(X,Y) ←
   Minimal(X),
   Solve(X,Y)
R(X,Y) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   Discriminate_k(HX,TX,Y),
   R(TX,TY),
   Process_k(HX,HY),
   Compose_k(HY,TY,Y)
```

Schema 1: Divide-and-conquer schema (where $1 \leq k \leq m$)

The constraints to be verified by first-order instances of this schema are listed in [9]. The most important one is that there must exist a well-founded relation "<" over the domain of the induction parameter, such that the instantiation of *Decompose* guarantees that TX_j "<" X, for every $1 \leq j \leq t$.

The insertion-sort program of Example 1 is a rewriting of the program obtained by applying the substitution {*Minimal/isEmptyList*, *NonMinimal/isNonEmpty*, *Decompose/headTail*, $Discriminate_1$*/true*, *Solve*/=, $Process_1$/=, $Compose_1$*/insert*, *m*/1, *h*/1, *h'*/1, *t*/1} to Schema 1, with the primitives defined as follows:

```
isEmptyList([])
isNonEmpty([_|_])
headTail([H|T],H,T)
```

and *insert*/3 is defined as in Example 1. Note that the logic program for *insert*/3 is not an instance of Schema 1: covering it would require a generalization to n-ary predicates, the handling of *non*-recursive non-minimal clauses, and the handling of auxiliary parameters (such as I), which can't be induction parameters. Such extensions are discussed in [8] [9].

2.2 Schema-Guided Synthesis

An interesting idea is to devise stepwise synthesis strategies where each step synthesizes instance(s) of some place-holder(s) of a given schema. The schema thus actively guides the synthesis, and constitutes a synthesis bias. A synthesis *strategy* determines the order of instantiation of the place-holders of its attached schema, and hence the order of "navigation" through the web of constraints attached to that schema.

A *tool-box* of schema-independent *methods* can then be developed for the instantiation of place-holders. Such methods may be merely based on databases of common instances. More sophisticated methods would perform actual computations for inferring such instances, based on the specification, the constraints, and the program synthesized so far. Several methods of such a tool-box might be applicable at each step, thus yielding opportunities for user interaction, or for the application of synthesis heuristics. I thus here advocate a very disciplined approach to synthesis: rather than use a uniform method for instantiating *all* place-holders of a given schema (possibly without any awareness of such a schema), one should deploy for each place-holder the *best*-suited method. I thus propose to view research on synthesis as (also see [22]):

(1) the search for adequate schemata;
(2) the development of useful methods of place-holder instantiation; and
(3) the discovery of interesting mappings between these methods and the place-holders of these schemata, these mappings being encoded in strategies.

As many methods would be schema-independent, or even place-holder-independent, one should also investigate synthesizers that are parameterized on schemata. In other words, the first synthesis step would then be to select a schema, and the subsequent steps would be either a hardwired sequence (specific to the selected schema) of applications of methods, or a user-guided selection of place-holders and methods. My grand view of synthesizers is thus one of a large *workbench* with a disparate set of highly specialized methods and a set of schemata that cover (as much as possible of) the space of all possible programs.

Given the specification of the top-level problem, and assuming the divide-and-conquer methodology has been selected, the synthesis of a divide-and-conquer program thus amounts to *first* reducing this specification to a set of $4+3\cdot m$ new specifications of sub-problems, and *then* synthesizing (not necessarily using the divide-and-conquer methodology!) programs for these specifications, before *finally* assembling these programs according to a divide-and-conquer schema. It is important to realize that instances of the place-holders are not (necessarily) directly derived, but are rather the results of auxiliary syntheses from derived specifications. A possible ordering strategy for Schema 1 would be to instantiate the place-holders as they appear in the schema; this is a very natural sequence, especially for human programmers.

2.3 Predicate Invention and the Divide-and-Conquer Schema

Regarding our investigation of predicate invention now, the instantiation of virtually all place-holders of the divide-and-conquer schema may require predicates that do not appear in the evidence. They could then be retrieved, in the best case, from the background knowledge, but this would be a very undisciplined way of proceeding and would inevitably result in a combinatorial explosion of the search space, unless that background knowledge is restricted to the necessary predicates (or a small superset thereof), which is a very unrealistic scenario.

The most unlikely place-holders to require predicate invention are *Minimal*, *NonMinimal*, and *Decompose*: for every datatype, there are standard ways of instantiating these manipulations of the induction parameter. This actually constitutes the *only* creative part of synthesis, because all the other instantiations necessarily and even deterministically follow from these decisions. Backtracking to these decision points and making new choices will yield other, but of course equivalent, programs. So it would be but good software engineering practice to start synthesis by choosing among such standard instantiations. I will discuss this re-use issue in Section 3.

Next, the $Discriminant_k$ place-holders are a bit more likely to require predicate invention: they are usually "tests" rather than "computations," and may thus require predicates that do not appear in the evidence. Of course, "tests," such as $\leq/2$ over natural numbers, actually *are* computational concepts, so they could be synthesized in turn, but it seems more reasonable to either re-use such standard instantiations from background knowledge (if available) or to query the user for this purpose. I will discuss this latter dialogue issue in Section 4.

Finally, the most likely place-holders to require predicate invention are *Solve*, the $Process_k$, and the $Compose_k$. Especially the latter two are even likely to require useful predicate invention, because the logic program under construction may feature nested "loops." In case of recursive descriptions of the instantiations, the synthesizer may use any standard method or just recursively call itself from the abduced evidence (this is the Synthesis Method), or it may apply one of the (new) methods shown in Section 5. Otherwise, there may be re-use methods (as shown in Section 3) or query methods (as shown in Section 4).

3 Predicate Invention through Re-Use

In my humble opinion, one of the biggest open problems of ILP in particular, and of Machine Learning in general, is the lack of effective and efficient use of the background knowledge. Such a knowledge base is potentially very large, and even grows after each learning session by assimilation of the newly learned knowledge. So, with current learning methodologies, the search space would simply grow exponentially. In

order to cope with this problem, learning sessions are usually started from very little background knowledge (often exactly the one that is "needed," or a small superset thereof). While this is a reasonable research scenario (because it may quickly establish the existence of a solution in the smallest search space, and hence in any large superset thereof), this is an unrealistic scenario in real life, where the background knowledge is *all* the knowledge amassed so far.

Unless the background knowledge is structured in some way, just like in real life (we don't *normally* think of daughters when synthesizing a sorting program, and vice-versa, but see below!), learning sessions will remain unrealistic, despite their establishing the feasibility of learning. Of course, picking the "relevant" background knowledge *is* a form of knowledge structuring (by assigning a non-zero relevance factor to all deemed-to-be-relevant predicates and a zero factor to all the other known predicates), but it is not always easy to do so adequately. It may even be counterproductive to do so, especially in synthesis. While the teaching of naive-sort, by putting *permutation*/2 and *ordered*/1 (only) into the background knowledge, may be a valid objective, such is not the case with specifying it: one specifies problems, not solutions! Also: why teach a computer something we already know? By doing so, one might bypass potentially more "beautiful" (or unknown) solutions, such as quick-sort, which would exist if only the background knowledge were large enough and/or predicate invention powerful enough. The hitch of course is that, if the background knowledge is large enough, then "ugly" solutions may be learned, such as the previously mentioned insertion-sort whose *insert*/3 is implemented in terms of *permutation*/2 and *ordered*/1. That's why I proposed the notion of useful predicate invention and why background knowledge needs to be structured. Some form of "discipline" needs to be brought into learning methodologies in order to cope with these problems.

In synthesis, there are good opportunities to do so. Especially schema-guided synthesis strategies provide us with neat opportunities for structuring the background knowledge according to standard instantiations of the place-holders of schemata. Strictly speaking, this is of course no real predicate invention.

Inventing *Minimal*, *NonMinimal*, *Decompose*, and the $Compose_k$ through Re-Use. My SYNAPSE synthesizer features sub-knowledge bases that are specific to the *Minimal*, *NonMinimal*, and *Decompose* place-holders of the divide-and-conquer schema. Its *Re-Use Method* for instantiating these place-holders consists of first detecting the type (or domain) of the chosen induction parameter, and then non-deterministically re-using a type-specific instantiation from the corresponding sub-knowledge base. In any case, it is good software engineering practice to start synthesis by re-using such standard operators. The other place-holders of the divide-and-conquer schema are less "predictable," and hence less appropriate for effective re-use via a structured knowledge base. Exceptions to this may be the $Compose_k$, because of their symmetry with *Decompose* and because of the symmetrical role of parameters X and Y: composing Y from its heads and tails can indeed be seen as *de*composing Y into these heads and tails (provided these tails are smaller than Y according to some well-founded relation, which is not necessarily the case for all composition operators), and decomposing X into its heads and tails can be seen as *com*posing X from these heads and tails, because of the reversibility of logic programs. So, given a "simple" instantiation of *Decompose*, the invention of "complex" $Compose_k$ can sometimes be avoided by changing the induction parameter and re-using some "complex" *Decompose*, because the $Compose_k$ might then have "simple" instantiations that are easier to invent.

Inventing the $Process_k$ and $Compose_k$ through Re-Use. Another method of predicate invention through re-use is commonly found in synthesizers, especially for the simultaneous instantiation of the $Process_k$ and $Compose_k$ place-holders, say $ProcComp_k(\boldsymbol{HX},\boldsymbol{TY},Y)$, of the divide-and-conquer schema: the detection of the existence of useful predicate invention usually results from some failure to find a non-recursive logic program for the invented predicate. At present, the search for such a non-recursive description is restricted to re-using the =/2 primitive only: compute the most-specific generalization (msg) of the abduced evidence (assuming it consists of positive facts only); if that msg satisfies some (dataflow) criteria, then it can be rewritten as a logic program for the invented predicate. For instance, if the msg of the abduced evidence is $procComp(A,B,[A|B])$, then the clause $procComp(A,B,C) \leftarrow C=[A|B]$ may be conjectured as a description of the invented predicate *procComp*/3. The used dataflow criteria here are that the Y parameter *must* be constructed in terms of *all* the $\boldsymbol{TY}$ parameters (otherwise recursion would have been useless) and *may* in addition be constructed in terms of the $\boldsymbol{HX}$ parameters. Similar criteria can be derived for other place-holders. The CILP, SIERES, and METAINDUCE synthesizers feature such a method. The *MSG Method* of my SYNAPSE synthesizer is a bit more evolved in that it can instantiate multiple different $ProcComp_k$ in terms of =/2, which means that there may be more than just one recursive clause ($m \geq 1$). However, if their

msg approach to predicate invention through re-use fails, then all these synthesizers hastily conjecture the existence of useful predicate invention and call themselves recursively from the abduced evidence (this is the Synthesis Method). Obviously, this is not always adequate, as there might be non-recursive descriptions of the invented predicate that are not in terms of the =/2 primitive only. Some generalization of the msg approach is thus needed, so that other background knowledge predicates can also be re-used.

4 Predicate Invention through Queries

In useful predicate invention, we look for a recursive logic program for the invented predicate, which thus represents a computational concept. So we may apply the entire body of knowledge of recursive algorithm design to this objective, which knowledge can be captured in program schemata and their attached constraints. Divide-and-conquer logic programs are an important subclass of recursive logic programs, and it turns out that, quite often, some of its place-holders can be easily instantiated by mere dialogue with the specifier, because s/he *must* know the predicate that would otherwise have to be invented. "Knowing a concept" means that one can act as a decision procedure for answering membership queries for that concept, but it doesn't necessarily imply the ability to actually write that decision procedure.[2]

Inventing *Solve* through Queries. For instance, a specifier who wants a logic program synthesized for the *sort*/2 relation must know what the sorted version of the empty list is. In general, given a minimal form of the induction parameter, the specifier must know the corresponding values of the other parameters, because otherwise s/he wouldn't even have the need for the overall program. The *Solve* operator can thus be instantiated by looking for appropriate information among the evidence or by simply querying the specifier for it:

```
Q: What is the formula Solve/2 such that sort([],S) ← Solve([],S) ?
A: Solve([],S) ← S=[]
```

In general, *Solve* can be instantiated by any (conjunctive) formula. Such predicate invention through queries is not always directly applicable: for instance, if synthesis for the sorting problem goes into the direction of insertion-sort, then *insert*/3 may be usefully invented, but the specifier cannot be queried for the instantiation of *Solve* for *insert*/3, 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. So such a query would have to be rephrased in terms of the *sort*/2 predicate:

```
Q: What is the formula Solve/3 such that sort([I],R) ← Solve(I,[],R) ?
A: Solve(I,[],R) ← R=[I]
```

Whatever predicates are invented, it thus seems possible to discover their *Solve* operators by asking appropriate questions in terms of the top-level predicate for which a program is being synthesized.

Inventing the $Discriminate_k$ through Queries. Similarly, a specifier who wants a logic program synthesized for the *sort*/2 relation must also know what the sorted version of a two-element list is, and *why* it is so. In other words, the ≤/2 predicate must be known to the specifier. In general, if the synthesizer discovers at least two different ways ($m \geq 2$) of process-composing Y from the $\boldsymbol{HX}$ and $\boldsymbol{TY}$, then the specifier must know how to discriminate between them, because otherwise s/he wouldn't even have the need for the overall program. The $Discriminate_k$ operators (of the top-level predicate, or of a predicate usefully invented for it) can thus often be instantiated by looking for appropriate information among the evidence or by simply querying the specifier for it. The following dialogue results from the synthesis of *insert*/3, which is invented if synthesis for the sorting problem goes into the direction of insertion-sort; the queries however had to be rephrased in terms of the top-level predicate *sort*/2:

```
Q: What is the formula Discriminate1/4 such that
     sort([A,B],[A,B]) ← Discriminate1(B,[],A,[A,B]) ?
A: Discriminate1(B,[],A,[A,B]) ← A≤B
Q: What is the formula Discriminate2/4 such that
     sort([A,B],[B,A]) ← Discriminate2(B,[],A,[B,A]) ?
A: Discriminate2(B,[],A,[B,A]) ← ¬(A≤B)
```

2. It would be interesting to examine specifiers (oracles) that are capable of answering other kinds of queries (subset, superset, … [1]) and to investigate other meanings of the phrase "knowing a concept."

Very similar questions would be asked in case synthesis goes towards merge-sort; they would help instantiate the discriminants of an invented *merge*/3 predicate. In general, the need for discriminants tends to appear at "lower" levels of the predicate hierarchy resulting from a synthesis.

Inventing the $Process_k$ through Queries. Let *multN*(*L,N,R*) hold iff integer-list *R* is integer-list *L* where all elements are multiplied by integer *N*. A specifier who wants a logic program synthesized for the *multN*/3 relation must know what the result for a one-element list is. In general, given the "smallest" non-minimal form of the induction parameter, the specifier must know the corresponding values of the other parameters, because otherwise s/he wouldn't even have the need for the overall program. The $Process_k$ operators can thus often be instantiated by looking for appropriate information among the evidence or by simply querying the specifier for it:

```
Q: What is the formula Process1/3 such that
     multN([A],N,[B]) ← Process1(A,N,B) ?
A: Process1(A,N,B) ← product(A,N,B)
```

where *product*(*X,Y,Z*) holds iff integer *Z* is the product of integers *X* and *Y*. The assumption here was that $Compose_1$ had already been instantiated. This need not always be the case, but doesn't pose any problems, because both operators can usually be instantiated simultaneously.

Inventing the $Compose_k$ through Queries. Let *minList*(*L,M*) hold iff integer *M* is the minimum element of integer-list *L*. A specifier who wants a logic program synthesized for the *minList*/2 relation must know what the result for a two-element list is. In general, given a "small" non-minimal form of the induction parameter, the specifier must know the corresponding values of the other parameters, because otherwise s/he wouldn't even have the need for the overall program. The $Compose_k$ place-holders can thus often be instantiated by looking for appropriate information among the evidence or by simply querying the specifier for it:

```
Q: What is the formula Compose1/3 such that
     minList([A,B],M) ← Compose1(A,B,M) ?
A: Compose1(A,B,M) ← min(A,B,M)
```

where *min*(*X,Y,Z*) holds iff integer *Z* is the minimum of integers *X* and *Y*. The assumption here was that $Process_1$ had already been instantiated.

Evaluation of Predicate Invention through Queries. It turns out that surprisingly many place-holders of the divide-and-conquer schema (namely all but the ones that had better be instantiated by a re-use method) can be instantiated by mere queries to the specifier. These queries can be kept entirely in terms of the specifier's conceptual language, and are simple, because they need only ask what "happens" when the induction parameter is of a minimal form or of a "small" non-minimal form. Even better, the specifier *must* know the answers to these queries, because otherwise s/he wouldn't even feel the need for the synthesized program. The answers are thus also in the specifier's conceptual language, and the predicates they contain are either part of the background knowledge (in which case there is a "nice" knowledge transfer, without the usual efficiency problems of re-use and the sometimes automa-g-ic flavor of inductive synthesis) or new predicates (in which case auxiliary syntheses need to be started for them).

All this indicates that synthesis may start from very little evidence, as the synthesizer can query the specifier for the missing minimal information. Of course, if the specifier has given (some of) that information, then the synthesizer must be able to locate it and appropriately use it. If all the information was given, then synthesis should even be fully automatic. In any case, the evidence language should be at least as sophisticated as the answer language for the queries. For instance, my SYNAPSE synthesizer features Horn-clauses as evidence language, and expects as evidence some facts as well as *all* the answers (called properties) to such queries. Its *Proofs-as-Programs Method*[3] "transfers" predicates from the clause bodies into the appropriate place-holders of the synthesized program. An interactive version that asks for missing information, rather than expect all the minimal information, is in preparation. The CLINT learner [3] also features an evidence language that is more expressive than just positive and negative facts (namely clauses), but it works in a completely different way, because it is a general purpose learner.

3. This method actually performs some kind of abduction, and might be renamed in the future.

5 Predicate Invention through Problem Generalization

When synthesizing a divide-and-conquer program, it sometimes becomes "difficult," if not impossible, to process-compose *Y* from the ***HX*** and ***TY***. Besides changing the induction parameter, or the well-founded relation over its domain (and hence the instances of *Minimal*, *NonMinimal*, and *Decompose*), or the guiding schema/methodology, one can also generalize the initial specification and then synthesize a recursive program from the generalized specification as well as a non-recursive program for the initial problem as a particular case of the generalized one. Paradoxically, the new synthesis then becomes "easier," if not possible in the first place. As an additional and beneficial side-effect, programs for generalized problems are often more efficient, because they feature (semi-)tail recursion (which is transformed by optimizing interpreters into iteration) and/or because the generalization provoked a complexity reduction by loop merging. The latter phenomenon is of interest here, because if the composition loop can be merged with the loop of the top-level predicate, then the composition operator need not be invented.

For a detailed overview of problem generalization techniques, the reader is invited to consult [5] or [6]. I here summarize Deville's presentation and borrow some of his examples, but adapt and extend where appropriate. Basically, there are two generalization approaches:

- In *structural generalization*, one generalizes the structure (type) of some parameter. For instance, an integer parameter could be generalized into a integer-list parameter, and the intended relation must then be generalized accordingly. This is called *tupling generalization*, and I shall restrict the discussion of structural generalization to it.
- In *computational generalization*, one generalizes a state of computation in terms of "what has already been done" and "what remains to be done." If information about what has already been done is not needed, then it is called *descending generalization* (partial results are reduced to their predecessors by only considering a suffix of execution), otherwise it is *ascending generalization* (partial results are extended to their successors by also considering a prefix of execution).

In Sections 5.1 to 5.3 hereafter, I will investigate whether these generalization techniques can be successfully applied for the purposes of predicate invention in inductive synthesis.

5.1 Predicate Invention through Structural Generalization

In this sub-section, I first briefly illustrate tupling generalization on an example, and then examine its potential for predicate invention.

Example 3: Let the constant *void* represent the empty binary tree, and the compound term *btree*(*L,E,R*) represent a binary tree of root *E*, left subtree *L*, and right subtree *R*. Let *flat*(*B,F*) hold iff list *F* contains the elements of binary tree *B* as they are visited by a prefix traversal of *B*. We also say that *F* is the *prefix representation* of *B*. A corresponding "naive" divide-and-conquer logic program could be:

```
flat(void,[])
flat(btree(L,E,R),F) ←
   flat(L,U), flat(R,V),
   H=[E],
   append(U,V,I), append(H,I,F)
```

where *append*(*A,B,C*) holds iff list *C* is the concatenation of list *B* to the end of list *A*. If *n* is the number of elements in tree *B*, then this logic program has an $O(n^2)$ time complexity (as opposed to the linear complexity one might expect), because composition is done through *append*/3, whose time complexity is linear in the number of elements in its first parameter (and not constant, as one might hope). This is all an inevitable consequence of the definition of lists. Worse, if *h* is the height of *B*, then this logic program builds a stack of *h* pairs of recursive calls, and it creates $2 \cdot n$ intermediate data structures, so it also has a very bad space complexity. In the (+,–) mode, the call to the composition operator *append*/3 cannot be moved in front of any of the recursive calls (well, at least not without significantly further degrading the time/space efficiency), so the *flat*/2 logic program is not (semi-)tail-recursive (assuming a left-to-right computation rule).

Let's now perform a tupling generalization of the initial specification: let *flats*(*Bs,F*) hold iff list *F* is the concatenation of the prefix representations of the elements of binary tree list *Bs*. Expressing the initial problem as a particular case of the generalized one yields:

```
flat(B,F) ←
   flats([B],F)
```

and a logic program for the generalized problem is:

```
flats([],[])
flats([void|Bs],F) ←
   flats(Bs,F)
flats([btree(L,E,R)|Bs],[E|TF]) ←
   flats([L,R|Bs],TF)
```

Note that the calls to *append*/3 have disappeared: the *append*/3 loops have been merged into the *flat*/2 loop. So the conjunction of the last four clauses yields the ideal $O(n)$ time complexity. This logic program builds a stack of $2 \cdot n+1$ recursive calls, and it creates as many intermediate data structures; fortunately, the logic program for *flats*/2 can be made tail-recursive in the mode (+,–), as the last two clauses are exclusive. ♦

Inventing the $Compose_k$ through Structural Generalization. Regarding predicate invention now, the most interesting phenomenon here is the possible disappearance of the composition operator: if, at synthesis time, an adequate structural generalization can be found, then the invention of that operator may be avoided! (Tail recursion optimization is "only" a run-time feature, and can also be exploited by mode-specific post-synthesis transformation.) However, it can be shown [14] that the *eureka* needed to adequately generalize the initial specification comes directly from the composition operator of the initial program, provided it is associative and has a left/right-identity element, say *e*. Trying to avoid having to invent a composition operator thus actually requires already knowing it! For instance, *flats*/2 was specified in terms of a *concatenation* precisely because the composition operator of *flat*/2 was *append*/3. Structural generalization is very easy if a (naive) program is already given, as this technique is very suitable for mechanical transformation: all operators of the generalized program are operators of the initial program, and the generalized program is again covered by the divide-and-conquer schema, namely (for simplicity, I restrict the presentation to the particular case where the induction parameter is decomposed into one head and two tails, and where there is only one non-minimal clause, that is $h=h'=m=1$ and $t=2$; generalizing this is quite straightforward) [14]:

```
R(X,Y) ←
   Rs([X],Y)
```

where:

```
Rs(Xs,Y) ←
   Xs=[],                          % Minimal-Rs
   Y=e                             % Solve-Rs
Rs(Xs,Y) ←
   Xs=[_|_],                       % NonMinimal-Rs
   Xs=[X|TXs],                     % Decompose_1-Rs
   Minimal(X),                     % Discriminate_1-Rs
   Rs(TXs,TY),
   Solve(X,HY),                    % Process_1-Rs
   Compose(HY,TY,Y)                % Compose_1-Rs
Rs(Xs,Y) ←
   Xs=[_|_],                       % NonMinimal-Rs
   Xs=[X|TXs],                     % Decompose_2-Rs (part 1)
   NonMinimal(X),                  % Discriminate_2-Rs
   Decompose(X,HX,TX_1,TX_2),      % Decompose_2-Rs (part 2)
   Rs([TX_1,TX_2|TXs],TY),
   Process(HX,HY),                 % Process_2-Rs
   Compose(HY,TY,Y)                % Compose_2-Rs
```

Schema 2: Tupling generalization schema, expressed using the divide-and-conquer operators

Moreover, if *Solve*/2 converts *X* into a constant "size" *Y*, then the conjunction `Solve(X,HY), Compose(HY,TY,Y)` of the second clause can be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. Often, this partial evaluation even results in an equality atom for *Y*, which can then be forward-compiled (into the head

of the second clause). The second and third clauses being mutually exclusive (by virtue of a constraint that *Minimal*/1 and *NonMinimal*/1 must be complementary over the domain of the induction parameter *X*), the recursive call `Rs(TXs,TY)` in the second clause can then be made iterative (by, *e.g.*, placing a cut after the call to *Minimal*/1). For instance, the prefix representation of the empty tree is the empty list (of size 0), so partial evaluation gives *F* = *TF*, which can indeed be compiled into the head of the second clause.

Finally, if *Process*/2 converts *HX* into a constant "size" *HY*, then the conjunction `Process(HX,HY), Compose(HY,TY,Y)` of the third clause can also be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. Often, this partial evaluation even results in an equality atom for *Y*, which can then be forward-compiled (into the head of the third clause), so that the recursive call `Rs([TX1,TX2|TXs],TY)` in the third clause also becomes iterative. For instance, the list [*E*] obtained through processing of the root *E* being of size/length 1, partial evaluation gives *F* = [*E*|*TF*], which can indeed be compiled into the head of the third clause.

If the last two conditions simultaneously hold (which is not unusual), then *Compose*/3 effectively disappears altogether and would thus not have to be invented. So one could perform structural generalization "blindly," that is without prior knowledge of the composition operator. One would then maintain a knowledge base of composition operators satisfying the associativity and identity requirements. A "devil's advocate" argument of course shows that such a knowledge base can never be complete. One would first try to re-use such a known composition operator (upon failure, one would instantiate it by other means) during the synthesis of a first program, and then (if appropriate) mechanically transform it [14], using Schema 2, into another program that reflects a structural generalization. This is certainly preferable to the alternative of first re-using any such standard operator for a "blind" structural generalization of the specification, and then synthesizing a program for it, because one wouldn't know in advance whether the chosen operator is adequate or not, nor even whether such an adequate operator even exists for the problem at hand. This latter approach would certainly be very difficult in the case of inductive synthesis from incomplete evidence, because the new evidence for the structurally generalized problem (derived by applying the chosen composition operator to the evidence of the initial problem) grows exponentially with the initial evidence. For instance, if the initial evidence consists of *n* facts and the chosen operator is commutative, then the new evidence could consist of 2^n-1 facts. Relaxing the commutativity condition makes things even worse, and it is not even clear how non-factual evidence can be structurally generalized.

As a partial conclusion, structural generalization is a very good technique for post-synthesis optimizing transformation, especially that it can be completely mechanical and that it can be decided in advance whether or not it leads to optimizations, and even to which optimizations it would lead [14]. Its only reasonable suitability for predicate invention seems to be via re-use from a knowledge base of standard composition operators that have certain properties, because the otherwise necessary *eureka* requires knowing what one actually hoped not having to know.

5.2 Predicate Invention through Ascending Generalization

In this sub-section, I first briefly illustrate ascending generalization on an example, and then examine its potential for predicate invention.

Example 4: Let *maxPlatLen*(*L*,*M*) hold iff integer *M* is the length of the longest plateau (sequence of identical elements) of list *L*. Constructing a "naive" logic program could yield:

```
maxPlatLen([],0)
maxPlatLen([H|T],M) ←
   maxPlatLen(T,N),
   Compose(H,N,M)
```

where the *Compose*/3 operator is yet to be instantiated. Unfortunately, it is impossible to do so without leaving the divide-and-conquer methodology and schema, because *T* is needed to decide whether *M* is *N* or *N*+1. However, decomposing *L* by splitting it after its first plateau leads to a successful synthesis:

```
maxPlatLen([],0)
maxPlatLen([H|T],M) ←
   firstPlateau([H|T],F,S),
   maxPlatLen(S,N),
```

```
    length(F,P),
    max(P,N,M)
```

where *firstPlateau*(*L,F,S*) holds iff list *F* is the first plateau of non-empty list *L*, and list *S* is the corresponding suffix of *L*; *length*(*L,N*) holds iff integer *N* is the number of elements of list *L*; and *max*(*A,B,M*) holds iff integer *M* is the maximum of integers *A* and *B*. If *n* is the number of elements in *L*, then this logic program has the ideal $O(n)$ time complexity (actually $O(2 \cdot n)$, but it can easily be linearized by merging the *firstPlateau*/3 and *length*/2 loops, which share parameter *F*). If *q* is the number of plateaus in *L*, then this logic program builds a stack of *q* recursive calls and creates *q* intermediate data structures, so it also has a very bad space complexity. In the (+,–) mode, the call to the composition operator *max*/3 cannot be moved in front of the recursive call (but the call to the processing operator *length*/2 can), so the logic program for *maxPlatLen*/2 is not tail-recursive.

Let's now perform an ascending generalization of the initial specification: let *maxPlatLenAsc*(*S,M,I,E,N*) hold iff there exist lists *L* and *P*, such that *L* is the concatenation of *P* and *S*, integer *I* is the length of the longest plateau in *P*, except for the last plateau of *P*, which has *N* occurrences of term *E*, and integer *M* is the length of the longest plateau in *L*. Parameters *E* and *N* constitute information about the prefix of execution. This specification can be significantly simplified, but the initial problem would then barely be recognizable. Expressing the initial problem as a particular case of the generalized one yields:

```
maxPlatLen(L,M) ←
   maxPlatLenAsc(L,M,0,ayşe,0)
```

because the witness for prefix *P* is then the empty list, which indeed has a longest plateau of length 0 and a last plateau of 0 occurrences of term *ayşe* (which establishes that we *may* after all think of daughters when synthesizing programs!). A logic program for the generalized problem is:

```
maxPlatLenAsc([],M,I,_,N) ←
   max(I,N,M)
maxPlatLenAsc([H|T],M,I,E,N) ←
   H=E,
   P is N+1,
   maxPlatLenAsc(T,M,I,E,P)
maxPlatLenAsc([H|T],M,I,E,N) ←
   H≠E,
   max(I,N,J),
   maxPlatLenAsc(T,M,J,H,1)
```

Note that the calls to *max*/3 have not disappeared (the disappearance of the calls to *firstPlateau*/3 and *length*/2 is only due to the fact that decomposition is no longer done by splitting off the first plateau). The conjunction of the last four clauses yields the ideal $O(n)$ time complexity. This logic program builds a stack of n+1 recursive calls, but creates no intermediate data structures; fortunately, the logic program for *maxPlatLenAsc*/5 can be made tail-recursive in the mode (+,–,+,+,+), as the last two clauses are mutually exclusive. ♦

Inventing the $Compose_k$ through Ascending Generalization. Regarding predicate invention now, the most interesting phenomenon here would again be the possible disappearance of the composition operator. However, it is at present unclear to me whether this is possible at all, and even less how the necessary *eureka* can be automatically found. The sheer necessity of introducing information about the prefix of execution seems to prevent any kind of automation, but I have included this sub-section in the hope that some reader has the required *meta-eureka*!

5.3 Predicate Invention through Descending Generalization

In this sub-section, I first briefly illustrate descending generalization on an example, and then examine its potential for predicate invention.

Example 5: Let *reverse*(*L,R*) hold iff list *R* is the reverse of list *L*. A corresponding "naive" divide-and-conquer logic program could be:

```
reverse([],[])
reverse([HL|TL],R) ←
   reverse(TL,TR),
   HR=[HL],
   append(TR,HR,R)
```

If n is the number of elements in list L, then this logic program has an $O(n^2)$ time complexity (as opposed to the linear complexity one might expect), because composition is done through *append*/3, whose time complexity is linear in the number of elements in its first parameter. Worse, this logic program builds a stack of n recursive calls, and creates n intermediate data structures, so it also has a very bad space complexity. In the (+,−) mode, the call to the composition operator *append*/3 cannot be moved in front of the recursive call (well, at least not without significantly further degrading the time/space efficiency), so the *reverse*/2 logic program is not tail-recursive (assuming a left-to-right computation rule).

Let's now perform a descending generalization of the initial specification: let *reverseDesc*(*L,R,A*) hold iff list R is the concatenation of list A to the end of the reverse of list L. Expressing the initial problem as a particular case of the generalized one yields:

```
reverse(L,R) ←
   reverseDesc(L,R,[])
```

and a logic program for the generalized problem is:

```
reverseDesc([],R,R)
reverseDesc([HL|TL],R,A) ←
   reverseDesc(TL,R,[HL|A])
```

Note that the call to *append*/3 has disappeared: the *append*/3 loop has been merged into the *reverse*/2 loop. So the conjunction of the last three clauses yields the ideal $O(n)$ time complexity. This logic program also builds a stack of n recursive calls, but it creates no intermediate data structures; fortunately, the logic program for *reverseDesc*/3 is even tail-recursive in the mode (+,−,+). ♦

Descending generalization thus introduces an accumulator parameter, which is progressively extended to the final result. The pair of parameters R and A can also be seen as representing the difference-list $R \setminus A$, which itself represents the difference between lists R and A, where A is a suffix of R. But descending generalization yields something more general than transformation to difference-list manipulation, because it is by no means restricted to creating difference-lists only: any form of difference-structures can be created. Another example would be difference-integer $I \setminus J$, which could represent $I–J$ or $max(I,J)$ or whatever.

Logic programs for descendingly generalized problems are *not* covered by the divide-and-conquer schema (Schema 1), because the accumulator is *extended* for recursive calls rather than reduced. The corresponding (tail-recursive!) schema is as follows [5] [14]:

```
R-desc(X,Y,A) ←
   Minimal(X),
   ExtendMin(X,A,Y)
R-desc(X,Y,A) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   ExtendNonMin(HX,A,NewA),
   R-desc(TX,Y,NewA)
```

Schema 3: Descending generalization schema

Inventing the $Compose_k$ through Descending Generalization. Regarding predicate invention now, the most interesting phenomenon here is again the possible disappearance of the composition operator: if, at synthesis time, an adequate descending generalization can be found, then the invention of that operator may be avoided! However, it can again be shown [5] [14] that the *eureka* needed to adequately generalize the initial specification comes directly from the composition operator of the initial program, provided it is associative and has a left-identity element, say e, and provided the initial problem exhibits a functional dependency from induction parameter X to parameter Y. Trying to avoid having to invent a composition operator thus actually requires already knowing it! For instance, *reverseDesc*/3 was specified in terms of a *concatenation* precisely because the composition operator of *reverse*/2 was *append*/3. Its formal specification:

$$reverseDesc(TL,R,HR) \Leftrightarrow \exists TR \;\; reverse(TL,TR) \wedge append(TR,HR,R)$$

has a right-hand side built of two atoms extracted from the logic program for *reverse*/2. Generally speaking now (for simplicity, I restrict the presentation to the particular case where the induction parameter has one head and one tail, and where there is only one non-minimal clause, that is $h{=}h'{=}t{=}m{=}1$; generalizing this is quite straightforward), the *eureka* can be mechanically found [5] [14] [20] by searching in the program for *R*/2 for a (not necessarily consecutive) sub-formula of the form `R(X,S), Compose(A,S,Y)`, so that the following formal specification for *R-desc*/3 can be postulated:

$$R\text{-}desc(X,Y,A) \Leftrightarrow \exists S \;\; R(X,S) \wedge Compose(A,S,Y) \tag{1}$$

The key principle here is that both parts of the sub-formula share some variable *S*. This search is easy if the program for *R*/2 was constructed in the first place so as to be an instance of Schema 1. Note that it is thus crucial that the *Process*/2 and *Compose*/3 operators are not merged (yet). The same principle can be employed for other loop mergers, but I am here only interested in descending generalization. For our current purpose, it doesn't even matter whether *Compose*/3 is already instantiated or not. Descending generalization is very easy if a (naive) program is already given, as this technique is very suitable for mechanical transformation: all operators of the generalized (tail-recursive!) program are operators of the initial program:

```
R(X,Y) ←
   R-desc(X,Y,e)
```

where:

```
R-desc(X,Y,A) ←
   Minimal(X),
   ( Solve(X,S), Compose(A,S,Y) )
R-desc(X,Y,A) ←
   NonMinimal(X),
   Decompose(X,HX,TX),
   ( Process(HX,HI), Compose(A,HI,NewA) ),
   R-desc(TX,Y,NewA)
```

Schema 4: Descending generalization schema, expressed using the divide-and-conquer operators

Moreover, if the intended relation behind *R*/2 maps the minimal form of parameter *X* into *e*, and if *e* is also a right-identity element of *Compose*/3, then the conjunction `Solve(X,S), Compose(A,S,Y)` of the first clause can be simplified into `Y=A`. For instance, the reverse of the empty list is the empty list, which is indeed the right-identity element of *append*/3, and even the minimal form of the second parameter.

Finally, if *Process*/2 converts *HX* into constant "size" *HY*, then the atom `Compose(A,HI,NewA)` in the second clause can be partially evaluated, which usually results in the disappearance of that call to *Compose*/3, and thus in a merging of the *Compose*/3 loop into the *R*/2 loop. For instance, the processing operator of *reverse*/2 maps the first element *HL* of a non-empty list into the singleton list [*HL*], so the atom `append([HL],A,NewA)` can indeed be partially evaluated, namely into `NewA=[HL|A]`. Further transformations yield the *reverseDesc*/3 program above. However, for *minList*/2 (see Section 4), the processing operator, namely =/2, maps the head of the list to itself, that is to an arbitrary integer, so the call to the composition operator, namely *min*/3, cannot be partially evaluated away. Generally speaking, if the elements of *X* are of the same type as *Y*, then that call to *Compose*/3 cannot be partially evaluated away, because the elements of *X* are usually not of a constant size, and hence are not processed into constant size *HY*.

If the last two conditions simultaneously hold (which is not unusual), then *Compose*/3 effectively disappears altogether and would thus not have to be invented. So one could perform descending generalization "blindly," that is without prior knowledge of the composition operator. One would then maintain a knowledge base of composition operators satisfying the associativity and identity requirements, and re-use them, along the lines laid out in Section 5.1.

Fortunately, in the case of *inductive* synthesis, there exists a more convenient other approach to "blind" descending generalization. The technique is based on the detection of the left-identity element of the composition operator, and this by mere inspection of the given evidence (I here assume that the left-identity also is a right-identity). If no re-use method succeeds in instantiating the *Compose*/3 operator, one could *conjecture* the existence of useful predicate invention, and either invoke the synthesizer recursively on abduced

evidence in order to synthesize a logic program for *Compose*/3 (this is the Synthesis Method), or proceed through the following tasks (called the *Descending Generalization Method*):

(1) discover the left-identity element of *Compose*/3 for *R*/2;
(2) express the initial problem *R*/2 in terms of its generalization *R-desc*/3;
(3) generalize the evidence for *R*/2 into evidence for *R-desc*/3;
(4) synthesize a logic program for *R-desc*/3.

The synthesized logic program for the initial problem *R*/2 then consists of the conjunction of the clause obtained from the second task and the program synthesized at the fourth task. Such avoidance of the potentially useful invention of *Compose*/3 of course requires the invention of *R-desc*/3, but this is much easier and may even result in increased efficiency of the overall synthesized logic program.

The first task is intriguing: how can left-identity e be discovered, without even knowing *Compose*/3?! Suppose (for simplicity) that the type $\mathcal{C}$ of induction parameter X is defined using base a and constructor $c/2$ (which takes as arguments an element of some type $\mathcal{E}$ and a tail of type $\mathcal{C}$), and that the type $\mathcal{D}$ of the other parameter Y is defined using base b and constructor $d/2$ (which takes as arguments an element of some type $\mathcal{F}$ and a tail of type $\mathcal{D}$). Then, due to the functional dependency requirement, the intended relation $\mathcal{R}/2$ behind predicate $R/2$ reflects a function $f: \mathcal{C} \to \mathcal{D}$ such that:

$$\mathcal{R}(X,Y) \;\Leftrightarrow\; Y = f(X)$$

where:

$$f(X) = g \;\text{ if }\; X = a$$
$$f(X) = k(p(H), f(T)) \;\text{ if }\; X = c(H,T)$$

where p: $\mathcal{E} \to \mathcal{D}$ is a function converting the elements of X into working values that are combined into Y by the binary function k: $\mathcal{D} \times \mathcal{D} \to \mathcal{D}$, which is associative and has left/right-identity e. (Note that $p/1$ would be the function underlying the *Process*/2 operator, and that $k/2$ would be the function underlying the *Compose*/3 operator of a possible logic program for $R/2$.) Now, e is not necessarily the base b, nor the image g under f of the base a or of any other term of type $\mathcal{C}$, but it *is* often the case that $e = b$, or even that $e = b = g$. As seen above, the latter is the case for *reverse*/2. So it suffices to scan the Y terms in *all* factual ground evidence of $\mathcal{R}/2$ for a common sub-term s and to postulate that $s = e$. Indeed, if $e = b$, then e must be common to all Y terms, because they are of type $\mathcal{D}$, which has base b. If this eventually leads to failure, then it could actually be that $b \neq e = f(a) = g$, and it suffices to find the fact $R(a,g)$, or to query the specifier for it. For *minList*/2, we have that $g = +\infty$, but, as seen above, the call to *min*/3 cannot be eliminated, so the Descending Generalization Method to avoiding the invention of *Compose*/3 will fail.

The second task is straightforward now, and can be done mechanically using the following formula:

$$R(X,Y) \;\Leftrightarrow\; R\text{-}desc(X,Y,e) \tag{2}$$

Indeed, unfolding the right-hand side using (1), and simplifying the result considering that e is a left-identity of *Compose*/3, provides the justification for this formula.

The third task is then trivial as well: one could use (2) again, but it would be a bad idea to do so. Indeed, the ground term e would then be common to *all* generalized facts, so it would probably still be present in the program synthesized for *R-desc*/3, which is obviously not what is wanted. Fortunately, the following theorem establishes that each given factual evidence $R(X,Y)$ can be generalized even further than "only" into $R\text{-}desc(X,Y,e)$, namely into $R\text{-}desc(X,Y[e/Z],Z)$, where Z is a new variable. As a consequence, the third parameters of all generalized facts would then all be different, as desired.

Theorem 1: If the intended relation $\mathcal{R}/2$ behind predicate $R/2$ reflects a function $f/1$ defined in terms of a processing function $p/1$ (underlying *Process*/2) and a composition function $k/2$ (underlying *Compose*/3), which has left-identity e, then $R(X,Y) \;\Leftrightarrow\; R\text{-}desc(X,Y[e/Z],Z)$, if variable Z does not occur in term Y.

Proof 1: We have:

$R\text{-}desc(X,Y[e/Z],Z)$
$\Leftrightarrow\; \exists S\;\; R(X,S) \wedge Compose(Z,S,Y[e/Z])$, by unfolding using (1), where S is a new variable;
$\Leftrightarrow\; \exists S\;\; R(X,S) \wedge Compose(Z,S,Y[e/Z])[Z/e]$, by universal instantiation;
$\Leftrightarrow\; \exists S\;\; R(X,S) \wedge Compose(e,S,Y)$, because Z does not occur in terms S and Y;
$\Leftrightarrow\; \exists S\;\; R(X,S) \wedge Y = S$, because e is the left-identity of *Compose*/3;
$\Leftrightarrow\; R(X,Y)$, by properties of =/2. ❑

Formula (2) is a corollary of this theorem, namely when Z is instantiated to e. Note that this theorem cannot be used instead of formula (2) for the second task, because the necessary substitution cannot be expressed at the clause level. Generalizing non-factual evidence is done by applying this theorem to all atoms involving predicate $R/2$.

The fourth task is a synthesis one, and is thus a "mere" recursive invocation of the entire synthesizer on the generalized evidence.

This Descending Generalization Method is related to the *Variable Addition Technique* of Summers [24] (but also see [16]). His THESYS system is a trace-based synthesizer [8] of LISP programs from positive facts, and is in a sense the ancestor of the mentioned SYNAPSE, CILP, SIERES, and METAINDUCE synthesizers, and also of ITOU. He presents his technique without an explanation of why and when it works, which is remedied here. Moreover, he directly changes the currently synthesized divide-and-conquer program into a descending generalization one, rather than explicitly generalizing the given evidence and starting a new synthesis therefrom.

We have now finished the theoretical overview of how to perform predicate invention by descending generalization. Let's illustrate the four tasks on our *reverse*/2 problem, and then discuss implementation issues.

Example 6: Suppose the given evidence for *reverse*/2 consists of the following ground facts:

```
reverse([],[])
reverse([a],[a])
reverse([b,c],[c,b])
reverse([d,e,f],[f,e,d])
```

The abduced evidence for the composition operator would then consist of the following ground facts (other abducible facts would be semantically redundant with the ones given here):

```
compose(a,[],[a])
compose(b,[c],[c,b])
compose(d,[f,e],[f,e,d])
```

The expected instantiation, namely *compose*(*HL,TR,R*) $\Leftrightarrow$ *append*(*TR*,[*HL*],*R*), cannot be found by the MSG Method (see Section 3), *i.e.*, by re-use of =/2 only. It could be found by the Synthesis Method from this abduced evidence, which would lead to the naive program of Example 5. But one could also apply the Descending Generalization Method. At the first task, constant [] is found to be common to all second arguments of the original facts. So we conjecture that e = []. At the second task, we use formula (2) to express the initial problem *reverse*/2 in terms of its generalization *reverseDesc*/3:

```
reverse(L,R) ←
  reverseDesc(L,R,[])
```

At the third task, using Theorem 1, we generate the following non-ground facts for *reverseDesc*/3:

```
reverse([],W,W)
reverse([a],[a|X],X)
reverse([b,c],[c,b|Y],Y)
reverse([d,e,f],[f,e,d|Z],Z)
```

At the fourth task, we call the synthesizer recursively on this generalized evidence, which should lead to the efficient program of Example 5. ♦

This requires that the synthesizer can be guided not only by the divide-and-conquer schema, but also by the descending generalization schema. Moreover, it must be able to handle non-ground factual evidence. The former requirement necessitates only a straightforward adaptation of my SYNAPSE, namely (see Section 2.2) the preliminary selection of a schema and the encoding of a synthesis strategy for the descending generalization schema (fortunately, just as I predicted in [8], the generic methods used by my divide-and-conquer strategy are sufficient for this). This generalization of SYNAPSE is even desirable, as the class of divide-and-conquer programs does not cover all possible recursive programs, so the initially specified problem could well be a computational generalization of some other problem and hence unsolvable by means of a divide-and-conquer program. The latter requirement necessitates only a change of the definitions of examples and properties in SYNAPSE: the old restriction to ground examples was arbitrary, especially that the system already worked without modification from non-ground examples (bodiless properties).

As a partial conclusion, descending generalization is also a very good technique for post-synthesis optimizing transformation, for the same reasons as structural generalization. Fortunately, for inductive synthesis, there is even another approach to predicate invention than "blind" re-use from a knowledge base of standard composition operators that have certain properties, namely a method based on the detection of the left-identity element of the composition operator, and this by mere inspection of the given evidence.

6 Conclusion

Predicate invention is only necessary when a finite axiomatization of the current predicate is otherwise impossible, which implies that the description of a necessarily invented predicate is recursive. So necessary predicate invention is a program synthesis task, and had thus best be done by a synthesizer rather than by a general purpose learner, because one can then re-use the vast body of knowledge about recursive algorithm design. Taking a schema-guided approach, I showed how predicate invention can be performed (if not avoided) by intelligent re-use from structured background knowledge, by necessarily successful intelligent dialogue with the specifier, and by generalization of the original problem. The shown methods are easy to graft onto most existing inductive synthesizers. The apparently "miraculous" synthesis of a sorting program from just four examples in Example 1 is definitely within reach of such an extended synthesizer.

Acknowledgments

Many thanks to Yves Deville for laying the groundwork on schema-guidance for structural and computational generalization, and to Axel van Lamsweerde for suggesting some of the mentioned extensions.

References

[1] Dana Angluin. Queries and concept learning. *Machine Learning* 2(4):319–342, April 1988.

[2] Ranan B. Banerji. Learning theoretical terms. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 93–112. Academic Press, 1992.

[3] Luc De Raedt and Maurice Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence* 53(2–3):291–307, February 1992.

[4] Luc De Raedt and Maurice Bruynooghe. Interactive concept learning and constructive induction by analogy. *Machine Learning* 8:107–150, 1992.

[5] Yves Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.

[6] Yves Deville and Jean Burnay. Generalization and program schemata. In E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'*89, pp. 409–425. The MIT Press, 1989.

[7] Peter Flach. Predicate invention in inductive data engineering. In P. Brazdil (ed), *Proc. of ECML'*93. LNAI 667:83–94, Springer-Verlag, 1993.

[8] Pierre Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publ., 1995.

[9] Pierre Flener. *Logic Program Schemata: Synthesis and Analysis*. Technical Report BU-CEIS-9502, Bilkent University, Ankara (Turkey), 1995. Submitted for publication.

[10] Pierre Flener and Yves Deville. Towards stepwise, schema-guided synthesis of logic programs. In T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'*91, pp. 46–64. Springer-Verlag, 1992.

[11] Pierre Flener and Yves Deville. Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 67–96. Wiley, 1993.

[12] Pierre Flener and Yves Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation* 15(5–6):775–805, May/June 1993.

[13] Pierre Flener and Luboš Popelínský. On the use of inductive reasoning in program synthesis: Prejudice and prospects. In L. Fribourg and F. Turini (eds), *Proc. of META'*94 *and LOPSTR'*94. LNCS nnn:xxx–yyy, Springer-Verlag, 1995.

[14] Pierre Flener and Yves Deville. *Logic Program Transformation through Generalization Schemata*. Technical Report BU-CEIS-95xx, Bilkent University, Ankara (Turkey), 1995. In preparation.

[15] Andreas Hamfelt and Jørgen Fischer-Nilsson. Inductive metalogic programming. In S. Wrobel (ed), *Proc. of ILP'*94 (GMD-Studien Nr. 237), pp. 85–96, Bad Honnef, Germany, 1994.

[16] Yves Kodratoff and Jean-Pierre Jouannaud. Synthesizing LISP programs working on the list level of embedding. In A.W. Biermann, G. Guiho, and Y. Kodratoff (eds), *Automatic Program Construction Techniques*, pp. 325–374. Macmillan, 1984.

[17] Stéphane Lapointe, Charles Ling, and Stan Matwin. Constructive inductive logic programming. *Proc. of ILP'*93 (Technical Report IJS-DP-6707, Jozef Stefan Institute, Ljubljana, Slovenia), pp. 255–264.

[18] Stephen Muggleton. Predicate invention and utility. *Journal of Experimental and Theoretical Artificial Intelligence* 6(1):127–130, 1994.

[19] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 261–280. Academic Press, 1992.

[20] Maurizio Proietti and Alberto Pettorossi. Synthesis of eureka predicates for developing logic programs. In N. Jones (ed), *Proc. of ESOP'*90. LNCS 432:306–325. Springer-Verlag, 1990.

[21] Céline Rouveirol. ITOU: Induction of first order theories. In S. Muggleton (ed), *Proc. of ILP'*91.

[22] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.

[23] Irene Stahl. *Predicate invention in ILP: An overview.* Technical Report 1993/06, Fakultät Informatik, Universität Stuttgart (Germany), 1993.

[24] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM* 24(1):161–175, January 1977.

[25] Rüdiger Wirth and Paul O'Rorke. Constraints for predicate invention. In S. Muggleton (ed), *Inductive Logic Programming*, pp. 299–318. Academic Press, 1992.