Logic Program Schemata: Synthesis and Analysis

Pierre Flener
Department of Computer Engineering and Information Science
Faculty of Engineering, Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@bilkent.edu.tr Voice: +90/312/266-4000 x1450

Abstract

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. A program schema and the constraints on its place-holders encode a programming methodology, and they are thus an interesting means of guiding many software engineering tasks. First, I propose a notation and terminology for schemata and their manipulations. Second, from a series of divide-and-conquer programs, I synthesize four increasingly powerful divide-and-conquer schemata and their constraints. Third, I outline a vision of schema-guided program synthesis and report on particular synthesis strategies that are guided by various schemata. Finally, I discuss the related work and outline directions for future work.

1 Introduction

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. 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.

In order to be more precise, we have to settle for a particular programming language (in Section 1.1), so that we may then define a corresponding language for schemata (in Section 1.2). I finish this introduction by specifying, in Section 1.3, all the programming problems referred to in this paper.

1.1 Logic Algorithms

I believe it is beneficial to decompose the program development stage into two sub-stages: (1) *synthesis* of an abstract algorithm, with prime concern about its logical correctness w.r.t. the specification, but not so much concern about its time or space complexity; and (2) *implementation* of the algorithm into a concrete program written in some programming language, with concern about achieving operational correctness w.r.t. the specification and about maximizing its time or space efficiency. Note that I use the term "synthesis" in a very large sense, that is without any bias as to what extent it is automated or manual. The advantages of this approach are twofold. First, an algorithm is not subjected to the computational model of the target programming language, nor to its deficiencies or even to its intended underlying machine architecture. Second, the algorithm implementation stage, now that it is clearly separated, may explicitly re-use the huge body of existing and ongoing research on algorithm transformation, algorithm implementation, and pro-

gram transformation. The truly creative effort thus goes into the synthesis of a first, correct algorithm. Moreover, the ensuing transformation is often easier to perform than the synthesis from scratch of an optimized algorithm. The disadvantage of this approach is however that algorithm complexity considerations cannot always be clearly dissociated from the algorithm synthesis process: sometimes, at the implementation time, it is "too late" to improve an algorithm. Special attention needs thus to be paid to make the synthesis of low-complexity algorithms possible, if not likely, rather than a mere pot-shot.

In this paper, I explore these issues in the logic programming framework, and focus on the concept of *logic algorithm*, as originally introduced by Deville [19] [20], and later also advocated by others [10] [11] [43] [74]. Informally, a logic algorithm is an algorithm expressed in (first-order) logic. For pragmatic reasons (for example, the ease of implementation into Prolog programs), I propose the following more restrictive formal definition:

Definition 1: A *logic algorithm* defining a predicate *r/n*, denoted $LA(r)$, is a closed well-formed formula (wff) of the form:

$$\forall X_1 \ldots \forall X_n \quad r(X_1,\ldots,X_n) \Leftrightarrow \mathcal{B}[X_1,\ldots,X_n]$$

where the X_i are distinct variables, and $\mathcal{B}$ is a wff in prenex disjunctive normal form, whose only free variables are $X_1,\ldots,X_n$. The atom $r(X_1,\ldots,X_n)$ is called the *head*, and $\mathcal{B}[X_1,\ldots,X_n]$ is called the *body* of the logic algorithm. ♦

Example 1: I now introduce a first programming problem that will be used throughout this paper. Informally speaking, a *plateau* is a non-empty list of identical elements. A *compact list* is a list of couples, where the first term of each couple, called the *value* of the couple, is different in two consecutive couples, and the second term of each couple, called the *counter* of the couple, is a positive integer. Throughout this paper, natural integers are represented as successors (using the functor *s*/1) of zero (denoted by the constant 0). We can now specify a list compression problem:

compress(*L*,*C*) iff *C* is a list of $\langle v_i, c_i \rangle$ couples, such that the ith plateau of *L* has c_i elements equal to v_i, where *L* is a list and *C* a compact list.

A possible logic algorithm for *compress*/2 is as follows:

```
∀L ∀C compress(L,C) ⇔ ∃HL ∃HL1 ∃HL2 ∃TL ∃TC ∃V ∃N ∃TTC
      L=[]              ∧ C=[]
    ∨ L=[HL]            ∧ C=[⟨HL,s(0)⟩]
    ∨ L=[HL1,HL2|TL]    ∧ HL1≠HL2
                        ∧ compress([HL2|TL],TC)
                        ∧ C=[⟨HL1,s(0)⟩|TC]
    ∨ L=[HL1,HL2|TL]    ∧ HL1=HL2
                        ∧ compress([HL2|TL],TC)
                        ∧ C=[⟨V,s(N)⟩|TTC] ∧ TC=[⟨V,N⟩|TTC]
```

Logic Algorithm 1: *LA*(*compress*) ♦

Such a logic algorithm is clearly easy to implement into, say, a Prolog program. A logic algorithm is truly multi-directional (multi-modal, reversible), but preserving this at the Prolog level is tricky and thus usually done by deriving correct permutations of literals and clauses for *each* directionality (mode) required by its specification [16]. (The specification above doesn't have any such information.)

In the following, I drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginning of bodies of logic algorithms. Moreover, for convenience and when appropriate, I often write logic algorithms in a more compact form, using De Morgan's laws in order to merge disjuncts. By extension, a logic algorithm $LA(r)$ can also designate a set of logic algorithms, provided the involved predicate-symbols feature a "uses"-hierarchy rooted in *r/n*.

Note that logic algorithms take the "natural" form of what one would expect to be a logic program. Indeed, the equivalence symbol is necessary to state when *r/n* is true as well as when *r/n* is false. Logic algorithms thus correspond in fact to the notion of completed normal programs [12]. Hence, reasoning on logic algorithms is equivalent to using normal programs, but reasoning on their completions. Implementing logic algorithms into normal programs is thus nothing but "de-completion" [10].

1.2 Logic Algorithm Schemata

We may now proceed to the definition of a schema language. Informally, in a first approximation, a *logic algorithm schema* is a second-order logic algorithm, and its place-holders are first/second-order variables. A particular logic algorithm, called an *instance* of the schema, is then obtained by instantiating the variables of the schema.

Example 2: Here is a logic algorithm schema for the generate-and-test methodology:

```
R(X,Y) ⇔ Generate(X,Y) ∧ Test(Y)
```

The logic algorithm *LA*(*sort*):

```
sort(L,S) ⇔ permutation(L,S) ∧ ordered(S)
```

is an instance of this generate-and-test schema, namely via the second-order substitution {*R/sort, X/L, Y/S, Generate/permutation, Test/ordered*}. ♦

Reality is more complex, however. Function-variables and predicate-variables (whose symbols start with uppercase letters, just like for first-order variables) may have any arity, and this calls for schema-variables (whose symbols start with lowercase letters, but they should not be confused with functors and predicate-symbols) to denote such arities. Conjunctions, disjunctions, and quantifications of any length may appear, and this calls for schema-variables to denote the lengths of such ellipses, and for notation-variables (whose symbols also start with lowercase letters) to range across such ellipses. Permutations of parameters/conjuncts/disjuncts/quantifications and unfold transformations may have to be performed in order to see why a logic algorithm is an instance of some schema.

Example 3: Given the logic algorithm schema:

$$\begin{array}{l} R(X_1,\ldots,X_n,Y) \Leftrightarrow \\ \qquad\qquad P(Y,Z_1,\ldots,Z_n) \\ \quad \wedge \bigwedge_{1\le i\le n} \; Q_i(X_i,Z_i) \end{array}$$

where *n* is a schema-variable and *i* is a notation-variable, it is not immediately clear why the logic algorithm *LA*(*foo*):

```
foo(S,B,A) ⇔
     permutation(A,SA)
   ∧ reverse(B,RB)
   ∧ append(SA,RB,S)
   ∧ ordered(SA)
```

is an instance of it. A possible schema substitution is {*R/foo, n/*2, X_1*/B*, X_2*/A, Y/S, P/append,* Z_1*/RB,* Z_2*/SA,* Q_1*/reverse,* Q_2*/sort*}. But the *sort*/2 atom must also be unfolded into the conjunction of the *permutation*/2 and *ordered*/1 atoms (as in *LA*(*sort*) of Example 2), and a series of permutations of parameters and conjuncts are furthermore required to actually obtain this instance. ♦

A wff-schema language is thus needed to write realistic logic algorithm schemata. The formal definitions of such a language and of its semantics are beyond the scope of this paper. Finally, there may be constraints on the possible instances of the predicate-variables (for instance in comparison to a specification): algorithm instances are guaranteed correct w.r.t. their specifications if the instantiated constraints hold.

Definition 2: A *logic algorithm schema* defining a predicate *R/n* is a wff-schema of the form:

$$\forall X_1 \ldots \forall X_n \quad R(X_1,\ldots,X_n) \Leftrightarrow \mathcal{B}[X_1,\ldots,X_n]$$

and a set $\mathcal{S}$ of wff-schemata, called *constraints*, relating *R* and the predicate-variables of $\mathcal{B}$, where *n* is a schema-variable or a constant, the X_i are distinct variables, *R* is a predicate-variable, and $\mathcal{B}$ is a wff-schema in prenex disjunctive normal form, whose free first-order variables are $X_1,\ldots,X_n$. All predicate-variables are implicitly existentially quantified. The atom-schema $R(X_1,\ldots,X_n)$ is called the *head*, and $\mathcal{B}[X_1,\ldots,X_n]$ is called the *body* of the logic algorithm schema. ♦

Hereafter, I drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginning of the bodies of logic algorithm schemata.

A logic algorithm schema without function-variables, predicate-variables, schema-variables, and notation-variables is a logic algorithm.

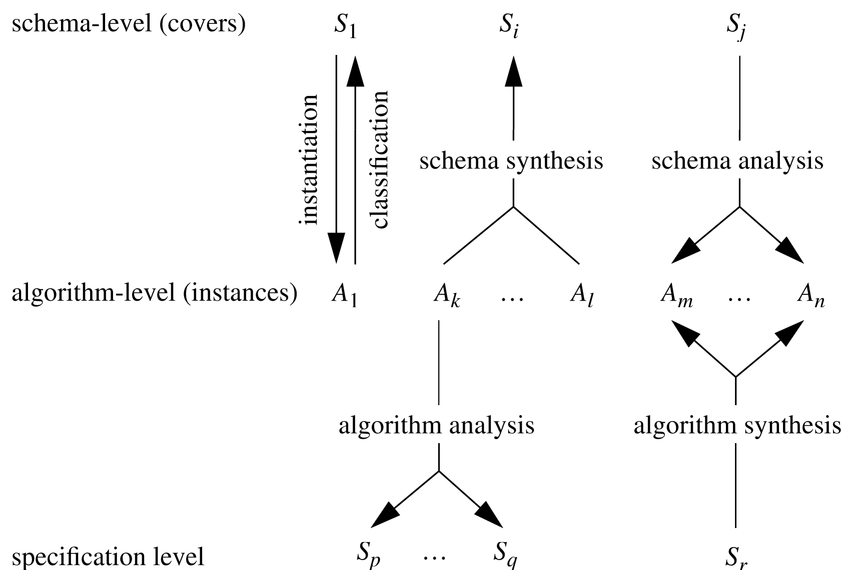

Figure 1: Specifications, algorithms, and schemata

Definition 3: An *instance* of a logic algorithm schema is a logic algorithm obtained by the following sequence of operations:

(1) permutation of the parameters, conjuncts, disjuncts, and quantifications of the schema;

(2) application of a schema substitution to the resulting schema, such that all function-variables, predicate-variables, and schema-variables are instantiated to first-order objects, and such that the constraints are satisfied;

(3) application of unfold transformations.

This process is called *instantiation* of a schema. The reverse process is called *classification* of the algorithm, and yields a schema called a *cover* of that algorithm. The process of classifying *several* algorithms into a same schema is called *schema synthesis* (or *algorithm abstraction*). Conversely, the process of instantiating a given schema into *several* algorithms is called *schema analysis*. The process of synthesizing several algorithms from a specification is called *algorithm synthesis*. This is a part of *software engineering*. Conversely, the process of "reverse-synthesizing" specifications from an algorithm is called *algorithm analysis*. This is a part of *reverse engineering*. See Figure 1 for an illustration of these processes. ♦

An algorithm synthesizer, whether human or (semi-)automated, should be able to synthesize a whole family of algorithms from a single specification. The different decisions taken at the choice-points during synthesis then provide an interesting algorithm classification. A popular idea is to benchmark the ability of a synthesizer on the sorting problem, whose specification is deceptively simple, and yet gives rise to a tremendous variety of different algorithms. Classification through synthesis of sorting algorithms has been done by Broy [9], Clark and Darlington [13], Follet [28], Green and Barstow [31], Lau [47] [49], Smith [62], and so on. Moreover, the ability to find new algorithms might be considered another benchmark of the ability of a synthesizer.

In order to facilitate the "visual" classification of logic algorithms, I introduce the following purely syntactic criterion for the writing of logic algorithms:

Definition 4: A *canonical representation* of a logic algorithm w.r.t. a covering schema exactly matches the layout of that schema.

Example 4: A possible canonical representation of *LA*(*foo*) with respect to the schema of Example 3 is:

```
foo(S,B,A) ⇔
     append(SA,RB,S)
   ∧ reverse(B,RB) ∧ ( permutation(A,SA) ∧ ordered(SA) ) ♦
```

Note that a canonical representation is not necessarily unique, because of the possible permutations of parameters. The permutations of parameters as in a schema can't be imposed at the instance level, because there is no possible control over how problems are actually specified. I thus can't introduce a concept such as *the* normalized representation of a logic algorithm. Almost all logic algorithms in this paper are canonical representations w.r.t. some schema in this paper.

Note that one may distinguish between *synthesis schemata* and *transformation schemata*: the former are useful for guiding algorithm synthesis, whereas the latter are useful for guiding algorithm transformation. But as nothing prevents the use of transformation schemata for guiding algorithm synthesis, the boundary between these two kinds of schemata is a subjective one, just as for the boundary between synthesis and transformation. For the sake of this paper, I assume the following purely syntactic criterion for distinguishing between synthesis and transformation: if the input and output languages are the same, then it is transformation, otherwise it is synthesis. Examples of transformation schemata are discussed in Section 4.5.

1.3 Specifications

A specification of a problem is a statement describing *what* the problem is and *how to use* a corresponding program. Ideally, a specification does not indicate *how to solve* the problem, but the question whether this is desirable, or even feasible!, is beyond the concerns of this paper, so I here admit virtually everything as a specification. Also, since I am here only concerned about the synthesis of algorithms, I drop most of the information about how to use a corresponding program, such as required call-modes. Throughout this paper, the used specification format (or schema!) is as follows:

$R(X_1,\ldots,X_n)$ iff / implies $\mathcal{R}[X_1,\ldots,X_n]$
where $\langle X_1,\ldots,X_n \rangle \in dom(\mathcal{R})$.

where R is a predicate-variable designating the predicate-symbol to be used, the X_i are distinct parameters, sentence $\mathcal{R}$ describes the *specified relation* (which may be different from the *intended relation*), and $dom(\mathcal{R})$ is the *domain* of $\mathcal{R}$. The symbols "iff", "implies", "where", "$\in$", … should not be taken literally here, as the "iff / implies" and "where" statements may be written in *any* language: if they are in (some subset of) natural language with *ad hoc* mathematical notations, then the specification is an *informal* specification, otherwise it is a *formal* specification. Equivalence specifications (with "iff") are the most frequent specifications. Implication specifications (with "implies") are for problems that just need to establish some post-condition $\mathcal{R}$: in that case, we are not interested in the algorithm that always fails, but in "weaker" algorithms (not necessarily the weakest algorithm, which would establish an equivalence with the post-condition). Whatever the underlying language, equivalence specifications can be abstracted as 3-tuples $\langle R, dom(\mathcal{R}), \mathcal{R}[X_1,\ldots,X_n] \rangle$. Whenever I want to be independent of the actual specification language, I will use this abstraction.

Formal specifications written in (some subset of) first-order logic have the description $\mathcal{R}[X_1,\ldots,X_n]$ of the specified relation as a wff, whose free variables are $X_1,\ldots,X_n$. The symbols "iff" and "implies" may then be replaced by "$\Leftrightarrow$" and "$\Rightarrow$", respectively. Such specifications are called *logic specifications*.

The domain $dom(\mathcal{R})$ of a relation $\mathcal{R}$ is, in general, a subset of the Cartesian product of the types of its parameters. Indeed, for example, the restriction of a parameter of type *list* to being non-empty need not necessarily appear in the "iff" statement, but may also appear in the "where" statement. The domain $dom(\mathcal{R}_X)$ of a parameter X in relation $\mathcal{R}$ is a subset of its type in $\mathcal{R}$. The domain of a relation is, in general, a subset of the Cartesian product of the domains of its parameters. But, for simplicity and without loss of generality, I here only consider relation domains that are equal to the Cartesian product of the domains of the relation's parameters. The reasons for this will become apparent in Section 2.4.1. By abuse of language, I sometimes talk about the domain of an algorithm or program.

Example 5: Suppose relation $\mathcal{R}$ has parameters X and Y of type *integer*, then we could have $dom(\mathcal{R}) = \{ \langle M, N \rangle \mid odd(M) \wedge integer(N) \wedge M < N \}$, and thus $dom(\mathcal{R}_X) = \{ M \mid odd(M) \}$, while $dom(\mathcal{R}_Y) = \{ N \mid integer(N) \}$. But $dom(\mathcal{R}) \subset dom(\mathcal{R}_X) \times dom(\mathcal{R}_Y)$. In order to handle $\mathcal{R}$ in the context of this paper, the constraint $M < N$ would have to be moved from the domain description to the relation description.

The main point here is that I treat domains as pre-conditions (that is, $\mathcal{R} \subseteq dom(\mathcal{R})$). This means that no algorithm has to verify whether its own parameters belong to its domain, though it must verify whether the parameters it passes to another algorithm belong to the domain thereof.

Example 6: Here are informal specifications (because that's enough for my purposes) of all the programming problems referred to in this paper:

append(*A,B,C*) iff *C* is the concatenation of *B* to the end of *A*,
where *A, B, C* are lists.

compress(*L,C*) iff *C* is a list of $\langle v_i, c_i \rangle$ couples, such that the ith plateau of *L* has c_i elements equal to v_i, where *L* is a list and *C* a compact list.

delOddElems(*L,R*) iff *R* is *L* without its odd elements,
where *L, R* are integer-lists.

efface(*E,L,R*) iff *R* is *L* without its first, existing occurrence of *E*,
where *E* is a term, *L* a non-empty list, and *R* a list.

firstPlateau(*L,P,S*) iff *P* is the first maximal plateau of *L*, and *S* is the corresponding suffix of *L*,
where *L* is a non-empty list, *P* a plateau, and *S* a list.

insert(*E,L,R*) iff non-descending list *R* is non-descending list *L* with *E*,
where *E* is an integer and *L*, *R* are integer-lists.

integer(*N*) iff *N* is an integer,
where *N* is a term.

intList(*L*) iff *L* is an integer-list,
where *L* is a term.

length(*L,N*) iff *N* is the length of *L*,
where *L* is a list and *N* a natural integer.

member(*E,L*) iff *E* is an element of *L*,
where *E* is a term and *L* a non-empty list.

member(*A,N,K,I*) iff *K* is the Ith element of *A*,
where *A* is a non-descending integer-array indexed from 1 to integer *N*, and *K* is an integer.

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

odd(*N*) iff *N* is odd,
where *N* is an integer.

ordered(*S*) iff *S* is non-descending,
where *S* is an integer-list.

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

permutation(*L,P*) iff *P* is a permutation of *L*,
where *L, P* are lists.

plateau(*N,E,P*) iff *P* is a plateau of *N* elements equal to *E*,
where *N* is a positive integer, *E* a term, and *P* a non-empty list.

reverse(*L,R*) iff *R* is the reverse of *L*,
where *L, R* are lists.

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

split(*L,F,S*) iff *F* is the first half of *L*, and *S* is the second half of *L*,
where *L, F, S* are lists.

sum(*L,S*) iff *S* is the sum of the elements of *L*,
where *S* is an integer and *L* an integer-list.

Note that these specifications are restricted to the objectives of algorithm synthesis, because no procedural requirements, such as required directionalities (modes), and no non-functional requirements, such as complexity thresholds, are given. ♦

The rest of this paper is now organized as follows. In Section 2, I incrementally synthesize logic algorithm schemata reflecting a divide-and-conquer synthesis methodology. In Section 3, I examine strategies for stepwise and schema-guided synthesis of logic algorithms. Related work is discussed in Section 4, before drawing some conclusions and outlining future work in Section 5.

2 Divide-and-Conquer Algorithm Schema Synthesis

In this paper, I mostly focus on the divide-and-conquer methodology, for the following reasons (cf. [62]):

- *diversity*: a wide variety of relations can be implemented by such algorithms;
- *complexity*: the resulting algorithms often have good time/space complexities;
- *simplicity*: the "simplicity" of this methodology makes it particularly convenient for (semi-)automated algorithm synthesis.

In essence, the divide-and-conquer synthesis methodology *solves* a problem by the following steps: [15]

(1) *divide* a problem into sub-problems, unless it can be trivially solved;
(2) *conquer* the sub-problems by solving them recursively;
(3) *combine* the solutions to the sub-problems into a solution to the initial problem.

Hence the name of the methodology. In the following, I focus on applying this methodology to data-structures, rather than to states of partial computations.

This methodology description is a little rough, though, and calls for further details. First, in Section 2.1, I manually synthesize a set of divide-and-conquer logic algorithms in a form suitable for schema synthesis. In Section 2.2, I then incrementally synthesize various versions of a divide-and-conquer logic algorithm schema from these logic algorithms. Next, in Section 2.3, I justify some of my schema synthesis decisions. Section 2.4 lists the constraints that instances of these schemata have to satisfy in order to actually be correct divide-and-conquer algorithms. In Section 2.5, I discuss various issues related to divide-and-conquer schemata. Finally, in Section 2.6, I compare the divide-and-conquer methodology to other methodologies.

2.1 A Set of Divide-and-Conquer Logic Algorithms

I now synthesize logic algorithms for some of the problems of Section 1.3, following the divide-and-conquer methodology, and using the other problems as primitives, whenever appropriate. Also, the logic algorithms may look at little bit convoluted sometimes, but this is on purpose, as it facilitates the subsequent schema synthesis, and actually makes these algorithms be in canonical representation w.r.t. some schema synthesized hereafter.

2.1.1 Logic Algorithms for the *compress*/2 Problem

Let's first synthesize a few logic algorithms for the *compress*/2 problem. I use this problem to go step-by-step through the divide-and-conquer methodology, and to define some terminology all along.

Step 1: Selection of an induction parameter. The first step is the selection of an induction parameter, that is the parameter to which the *divide* (decomposition) operator is applied. Only parameters of inductive types (such as *integer*, *list*, *tree*, …) are eligible as induction parameters. I hypothesize that no parameter is a tuple, that is that procedure declarations are "flattened" out.

Definition 5: A *simple induction parameter* is composed of exactly one parameter. A *compound induction parameter* is composed of at least two parameters.

Many problems can be implemented by algorithms with induction over a simple induction parameter. Selection heuristics, such as the Functionality Heuristic and the Directionality Heuristic, can be found in [20]. In the following, when talking about a logic algorithm $LA(r)$, I sometimes write $LA(r\text{-}X)$ to show that X was selected as induction parameter. Note however that the predicate defined by $LA(r\text{-}X)$ still is r, and not $r\text{-}X$.

Example 7: For the *compress*/2 problem, suppose that L is selected as (simple) induction parameter. Alternative decisions will be taken later in this sub-section.

Step 2: Synthesis of structural form-identifying formulas. Structural forms are terms. The second step starts with the selection of minimal and non-minimal forms of the induction parameter, such that every minimal form "represents" a problem that can be trivially solved, and every non-minimal form "represents" a

problem that needs to be solved by the *divide*, *conquer*, and *combine* operators. A minimal form is thus a form to which at least one computation starting from a non-minimal form eventually reduces the induction parameter. Moreover, all these forms must partition the domain of the induction parameter. This amounts to selecting a well-founded relation (wfr) over the domain of the induction parameter, as conveyed in the following definition:

Definition 6: Given a wfr "<" over the domain $\mathcal{D}$ of the induction parameter, the *minimal forms* of the induction parameter are the minimal elements of the well-founded set ($\mathcal{D}$,<), and the *non-minimal forms* of the induction parameter are the non-minimal elements of ($\mathcal{D}$,<). A *structural form-identifying formula* is a wff that decides whether or not a given term is of a given form.

The selected wfr is used in the total correctness proof of the synthesized algorithm with respect to its specification. Wfr selection heuristics, such as the Intrinsic Heuristic and the Extrinsic Heuristic, can be found in [20]. The second step concludes by the "synthesis" of a structural form-identifying formula for each obtained form. I use quotes here because this may amount to a selection from a database of predefined domain-specific wffs rather than to actual synthesis based on specification information.

Example 8: For the *compress*/2 problem, suppose that *is-the-tail-of* is selected as wfr over *list*, which is the domain of *L*. This means that the empty list, denoted [], is the unique minimal form of *L*, and the non-empty list, denoted [_|_], is the unique non-minimal form of *L*. Weaker relations, such as *has-less-elements-than* or *is-a -suffix-of*, are also suitable, but it is best to start with strong relations and relax them if needed. Possible form-identifying formulas are thus *L* = [] and *L* = [_|_] (or $L \neq$ [], or *length*(*L*,*N*) $\wedge$ $N > 0$), respectively.

Step 3: Synthesis of a *divide* operator. The third step consists of the synthesis of an operator that divides (decomposes) the induction parameter into values (called *tails*) that are smaller than it, according to the wfr selected at the second step. This is only applicable if the induction parameter is of a non-minimal form. Other values (called *heads*) allow the reconstruction of *X* from its tails. There are at least three strategies according to which the induction parameter, say *X*, can be decomposed:

- *intrinsic decomposition*: *X* is decomposed into $h \geq 1$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of *X*; for example, *L* = [*HL*|*TL*];
- *extrinsic decomposition*: *X* is decomposed into $h \geq 0$ heads and $t \geq 1$ tails in a manner reflecting the definition of the type of some other parameter than *X*, or reflecting the intended relation; for example, *partition*(*L,P,S,B*);
- *logarithmic decomposition*: *X* is decomposed into $h = 0$ heads and $t \geq 2$ tails of about equal size; for example, *split*(*L,F,S*).

An intrinsic decomposition reflects a wfr selected via the Intrinsic Heuristic, and an extrinsic or logarithmic decomposition reflects a wfr selected via the Extrinsic Heuristic (see [20]). None of these three strategies is superior to the others. In the following, when talking about a logic algorithm *LA*(*r*), I sometimes write *LA*(*r-int/ext/log-X*), in order to show that *X* was selected as induction parameter, and that the intrinsic/extrinsic/logarithmic decomposition strategy was applied. Again, the predicate defined by *LA*(*r-int/ext/log-X*) still is *r*, and not *r-int/ext/log-X*.

Once a decomposition strategy selected, a corresponding decomposition operator has to be "synthesized." I again use quotes here because this may amount to a selection from a database of predefined domain-specific operators rather than to actual synthesis based on specification information.

Example 9: For the *compress*/2 problem, suppose that the intrinsic decomposition strategy is selected, and that *L* = [*HL*|*TL*] is then selected as a decomposition operator. Alternative decisions will be taken later in this sub-section.

Step 4: Synthesis of the *conquer* operator. The fourth step consists of the synthesis of the *conquer* operator. This is a mere syntactic operation, as it reduces to the generation of a conjunction of recursive calls, one for each tail of the induction parameter that is computed by the *divide* operator. This yields tails of all the other parameters.

Example 10: For the *compress*/2 problem, with the decisions taken in the previous three examples, the atom *compress*(*TL*,*TC*) is synthesized, introducing a unique tail *TC* of the other parameter *C*. So far, the synthesized logic algorithm thus is:

```
compress(L,C) ⇔
     L=[]
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ compress(TL,TC)
```
♦

Step 5: Synthesis of the *combine* operators. The fifth and last step is the synthesis of the *combine* operators, which formalize how the other parameters relate to the induction parameter, for each of its structural forms. There may be alternative (but not necessarily mutually exclusive) ways to do so for a given form. The overall results are *structural cases*, and here take the form of conjunctions of literals.

Definition 7: A structural case is a *minimal case* if the induction parameter is of a minimal form, and a *non-minimal case* if the induction parameter is of a non-minimal form.

This step is (surprisingly) not creative either, as everything just follows from the previous steps and from the specification. Domain-checking literals have to be added to ensure that all predicates are correctly used. This is important in view of achieving correctness of the logic algorithm w.r.t. its specification.

Example 11: For the *compress*/2 problem, we proceed as follows:

- if L is of the selected minimal form ($L = []$), then C must be empty, too ($C = []$);
- if L is of the selected non-minimal form ($L = [_|_]$) and is intrinsically decomposed via $L = [HL|TL]$ such that *compress*(TL,TC) holds, then:
  - if TL is empty or TL is non-empty *and* starts with a term different from HL, then C must be $[\langle HL,s(0)\rangle|TC]$;
  - if TL is non-empty and starts with a term identical to HL, then C must be $[\langle V,s(s(N))\rangle|TTC]$, where $TC = [\langle V,s(N)\rangle|TTC]$.

The structural cases are mutually exclusive here, because given a list L, there is only one compact list C corresponding to L. No domain-checking literals have to be added. Hence the following logic algorithm for *compress*(L,C), after minor re-arrangements:

```
compress(L,C) ⇔
     L=[]    ∧ C=[]
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ (TL=[]) ∨ (TL=[HTL|TTL] ∧ HL≠HTL)
             ∧ compress(TL,TC)
             ∧ HC=⟨HL,s(0)⟩
             ∧ C=[HC|TC]
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ TL=[HTL|TTL] ∧ HL=HTL
             ∧ compress(TL,TC)
             ∧ HC=_
             ∧ C=[⟨V,s(s(N))⟩|TTC] ∧ TC=[⟨V,s(N)⟩|TTC]
```

Logic Algorithm 2: *LA*(*compress-int-L*) ♦

Example 12: The following other logic algorithm for *compress*(L,C) has been synthesized by extrinsic decomposition of C. I decomposed C into something smaller in a way reflecting the type of L, namely *list*. Since every element of L represents an increment by 1 of the counter in an element of C, the idea was to decompose C by decrementing, if possible, the counter of its first element by 1. This decomposition was non-trivial, but considerably facilitated the rest of the synthesis. Note that there is (surprisingly) no usage of ≠/2, because of the domains-as-preconditions approach: C is *assumed* to be a compact list.

```
compress(L,C) ⇔
     C=[]    ∧ L=[]
   ∨ C=[_|_] ∧ decompose(C,HC,TC)
             ∧ compress(TL,TC)
             ∧ HL=HC
             ∧ L=[HL|TL]
```

```
decompose(C,HC,TC) ⇔
     C=[⟨V,s(0)⟩|T] ∧ HC=V ∧ TC=T
   ∨ C=[⟨V,s(s(N))⟩|T] ∧ HC=V ∧ TC=[⟨V,s(N)⟩|T]
```

Logic Algorithm 3: *LA*(*compress-ext-C*) ♦

Example 13: The following other logic algorithm for *compress*(*L,C*) has been synthesized by intrinsic decomposition of *C*:

```
compress(L,C) ⇔
     C=[]    ∧ L=[]
   ∨ C=[_|_] ∧ C=[HC|TC]
             ∧ compress(TL,TC)
             ∧ HC=⟨E,N⟩ ∧ plateau(N,E,HL)
             ∧ firstPlateau(L,HL,TL)
```

Logic Algorithm 4: *LA*(*compress-int-C*) ♦

Example 14: The following last logic algorithm for *compress*(*L,C*) has been synthesized by extrinsic decomposition of *L*. I decomposed *L* into something smaller in a way reflecting the type of *C*, namely *compact list*. Since every element of *C* represents a "summary" of a plateau of *L*, the idea was to decompose *L* by extracting its first maximal plateau as head *HL* of *L*, and the corresponding suffix as tail *TL* of *L*. This decomposition was non-trivial, but considerably facilitated the rest of the synthesis.

```
compress(L,C) ⇔
     L=[]    ∧ C=[]
   ∨ L=[_|_] ∧ firstPlateau(L,HL,TL)
             ∧ compress(TL,TC)
             ∧ plateau(N,E,HL) ∧ HC=⟨E,N⟩
             ∧ C=[HC|TC]
```

Logic Algorithm 5: *LA*(*compress-ext-L*) ♦

Synthesis is not necessarily a linear process where one goes from Step 1 to 5. Especially the first three steps have a lot of choice-points to which synthesis may backtrack, either because it fails at a later step (for instance, because the selected wfr is too strong) or because alternative algorithms need to be synthesized.

2.1.2 Other Logic Algorithms

I now synthesize a few more logic algorithms for selected other problems. I give only brief information about the synthesis decisions, but add useful comments and terminology where appropriate.

Example 15: The following logic algorithm for *delOddElems*(*L,R*) has been synthesized by intrinsic decomposition of *L*:

```
delOddElems(L,R) ⇔
     L=[]    ∧ R=[]
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ odd(HL)
             ∧ delOddElems(TL,TR)
             ∧ HR=_
             ∧ R=TR
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ ¬odd(HL)
             ∧ delOddElems(TL,TR)
             ∧ HR=HL
             ∧ R=[HR|TR]
```

Logic Algorithm 6: *LA*(*delOddElems-int-L*)

This logic algorithm is typical for all *filtering problems*. It is a *complete-traversal logic algorithm*: all elements of the induction parameter are visited for a specific solution. Moreover, it is a *single-loop logic algorithm*: only one traversal of the induction parameter is being performed at any moment. Finally, it is a *semantic-manipulation logic algorithm*: some—if not all—constituents of the induction parameter are shuf-

fled around conditionally, as their values *are* relevant, or even new constituents are used, for the construction of the other parameters. The non-minimal cases are mutually exclusive, because given a list *L*, there is only one list *R* that is *L* where certain elements have been filtered out (the reverse is not true). A logic algorithm synthesized by intrinsic decomposition of *R* would be quite different from the one above. ♦

Example 16: The following logic algorithm for *efface*(*E,L,R*) has been synthesized by intrinsic decomposition of *L*:

```
efface(E,L,R) ⇔
     L=[_]      ∧ L=[HL] ∧ E=HL ∧ R=[]
   ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ HL=E
                ∧ E=HL ∧ R=TL
   ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ HL≠E
                ∧ efface(TE,TL,TR)
                ∧ HE=_ ∧ HR=HL
                ∧ E=TE ∧ R=[HR|TR]
```

Logic Algorithm 7: *LA*(*efface-int-L*)

Note that the non-minimal form gives rise to two structural cases, one of them without recursion. This logic algorithm is a *prefix-traversal logic algorithm*: only the first few elements of the induction parameter are visited for a specific solution. The non-minimal cases are mutually exclusive, because given a list *L* and a term *E*, there is only one list *R* that is *L* without its first occurrence of *E* (the reverse is not true). Also note that *E* could have been used instead of *TE* in the recursive atom, because *E* and *TE* are "later" unified anyway. Such a parameter is called an *auxiliary parameter*, because it has nothing to do with the "inductive nature" of the problem. Induction on an auxiliary parameter is obviously a bad idea. I will come back to auxiliary parameters in Section 2.5.2. The reader is invited to synthesize a logic algorithm by intrinsic decomposition of *R*, and to compare it with the one above. ♦

Example 17: The following logic algorithm for *member*(*E*,*L*) has been synthesized by intrinsic decomposition of *L*:

```
member(E,L) ⇔
     L=[_]      ∧ L=[A] ∧ E=A
   ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ true
                ∧ E=HL
   ∨ L=[_,_|_] ∧ L=[HL|TL]
                ∧ true
                ∧ member(TE,TL)
                ∧ HE=_
                ∧ E=TE
```

Logic Algorithm 8: *LA*(*member-int-L*)

Parameter *E* is an auxiliary parameter. Note that the non-minimal form gives rise to two cases, one of them without a recursive atom. This is also a prefix-traversal logic algorithm. However, the non-minimal cases are *not* mutually exclusive: a non-empty list may have more than one member. Overall, thus, all the elements of the induction parameter are *eventually* visited, but upon backtracking only. ♦

Example 18: The following logic algorithm for *merge*(*A,B,C*) has been synthesized by intrinsic decomposition of the compound induction parameter ⟨*A*,*B*⟩:

```
merge(A,B,C) ⇔
      A=[]                 ∧ C=B
   ∨             B=[]      ∧ C=A
   ∨  A=[_|_] ∧ B=[_|_]    ∧ decompose(⟨A,B⟩,HAB,⟨TA,TB⟩)
                           ∧ true
                           ∧ merge(TA,TB,TC)
                           ∧ HC=HAB
                           ∧ C=[HC,HC|TC]

decompose(⟨A,B⟩,HAB,⟨TA,TB⟩) ⇔
      A=[HAB|TA] ∧ B=[HB|_] ∧ HAB≤HB ∧ TB=B
   ∨  A=[HA|_] ∧ B=[HAB|TB] ∧ HAB<HA ∧ TA=A
```

Logic Algorithm 9: *LA*(*merge-int-*⟨*A,B*⟩)

Note that there are two minimal forms and one non-minimal form. Parameters *A* or *B* alone as induction parameter do not lead to linear-complexity algorithms. This is however possible with *C* alone. ♦

Example 19: The following logic algorithm for *plateau*(*N,E,P*) has been synthesized by intrinsic decomposition of *N*:

```
plateau(N,E,P) ⇔
      N=s(0)    ∧ E=_ ∧ P=[E]
   ∨  N=s(s(_)) ∧ N=s(TN) ∧ HN=N
                ∧ true
                ∧ plateau(TN,TE,TP)
                ∧ HE=A ∧ HP=A
                ∧ E=TE ∧ E=HE ∧ P=[HP|TP]
```

Logic Algorithm 10: *LA*(*plateau-int-N*)

Parameter *E* is an auxiliary parameter. Note the intricate way in which *HP* is unified with the appropriate term. A logic algorithm synthesized by intrinsic decomposition of *P* would be quite similar to the one above. ♦

Example 20: The following logic algorithm for *sort*(*L,S*) has been synthesized by intrinsic decomposition of *L*, yielding the Insertion-Sort algorithm:

```
sort(L,S) ⇔
      L=[]    ∧ S=[]
   ∨  L=[_|_] ∧ L=[HL|TL]
              ∧ sort(TL,TS)
              ∧ HS=HL
              ∧ insert(HS,TS,S)
```

Logic Algorithm 11: *LA*(*sort-int-L*) {Insertion-Sort}

The following other logic algorithm for *sort*(*L,S*) has been synthesized by extrinsic decomposition (reflecting the intended relation) of *L*, yielding the Quick-Sort algorithm:

```
sort(L,S) ⇔
      L=[]    ∧ S=[]
   ∨  L=[_|_] ∧ L=[HL|T] ∧ partition(T,HL,TL1,TL2)
              ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
              ∧ HS=HL
              ∧ append(TS1,[HS|TS2],S)
```

Logic Algorithm 12: *LA*(*sort-ext-L*) {Quick-Sort}

The following third logic algorithm for *sort*(*L,S*) has been synthesized by logarithmic decomposition of *L*, yielding the Merge-Sort algorithm:

```
sort(L,S) ⇔
     L=[]      ∧ S=L
   ∨ L=[_]     ∧ S=L
   ∨ L=[_,_|_] ∧ split(L,TL1,TL2)
               ∧ true
               ∧ sort(TL1,TS1) ∧ sort(TL2,TS2)
               ∧ true
               ∧ merge(TS1,TS2,S)
```

Logic Algorithm 13: *LA*(*sort-log-L*) {Merge-Sort}

Note that there are two minimal forms and one non-minimal form.

All these sorting algorithms are *multiple-loop logic algorithms*: at least two nested traversals of the induction parameter are being performed at some moment. ♦

Example 21: The following logic algorithm for *split*(*L,F,S*) has been synthesized via a slightly different methodology, namely first generalization of the initial problem by introduction of an additional parameter, then synthesis by structural induction for the new problem, and finally expression of the old problem in terms of the new one.

```
split(L,F,S) ⇔
     split(L,L,F,S)
split(L,M,F,S) ⇔
     M=[]      ∧ F=[] ∧ S=L
   ∨ M=[_]     ∧ F=[] ∧ S=L
   ∨ M=[_,_|_] ∧ M=[HM1,HM2|TM] ∧ L=[HL|TL]
               ∧ true
               ∧ split(TL,TM,TF,TS)
               ∧ HF=HL ∧ HS=_
               ∧ F=[HF|TF] ∧ S=TS
```

Logic Algorithm 14: *LA*(*split*)

Note that there are two minimal forms and one non-minimal form for *split*/4. This is a *structural-manipulation logic algorithm*: some—if not all—constituents of the induction parameter are shuffled around unconditionally, because their values are irrelevant, for the construction of the other parameters. ♦

2.2 Divide-and-Conquer Logic Algorithm Schema Synthesis

I now incrementally synthesize four versions of a divide-and-conquer logic algorithm schema, starting from the logic algorithms of the previous section. Each new version covers a larger set of logic algorithms.

2.2.1 First Version: Binary Predicates, One Case Each, No Subcases

Let's first restrict ourselves to binary predicates, and examine a most basic methodology that already yields solutions to many problems.

A *divide-and-conquer algorithm* 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 *X* into a vector ***HX*** of heads of *X* and a vector ***TX*** of 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 tails ***TX*** recursively yield tails ***TY*** of *Y*. The heads ***HX*** are processed into a vector ***HY*** of heads of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***.

For further discussion, let's quantify the vectors as follows. There are *h* heads of *X*, *h'* heads of *Y*, and *t* tails of *X*, hence *t* tails of *Y*. Thus:

$$\#\boldsymbol{HX} = h$$

$$\#\boldsymbol{HY} = h'$$

$$\#\boldsymbol{TX} = \#\boldsymbol{TY} = t$$

Note that *h, h', t* are schema-variables, not constants.

```
R(X,Y) ⇔
            Minimal(X)     ∧ Solve(X,Y)
  ∨         NonMinimal(X)  ∧ Decompose(X,HX,TX)
                           ∧ R(TX,TY)
                           ∧ Process(HX,HY)
                           ∧ Compose(HY,TY,Y)
```

Logic Algorithm Schema 1: Divide-and-conquer (version 1)

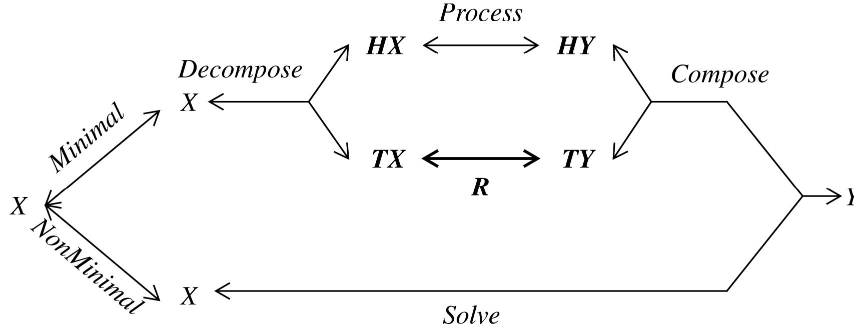

Figure 2: Dataflow and control-flow in divide-and-conquer algorithms

Logic algorithms synthesized by this basic divide-and-conquer methodology are covered by Schema 1, where ***R*(*TX*,*TY*)** stands for $\wedge_{1 \leq j \leq t} R(TX_j, TY_j)$, and *j* is a notation-variable. The dataflow and control-flow can be graphically represented as in Figure 2.

Example 22: Logic algorithms covered by Schema 1 are *LA*(*compress-ext-C*) (LA 3), *LA*(*compress-int-C*) (LA 4), *LA*(*compress-ext-L*) (LA 5), *LA*(*sort-int-L*) (LA 11), and *LA*(*sort-ext-L*) (LA 12).

Technically speaking, the instance of a predicate-variable *P/n* of a schema is a predicate-symbol *p/n*. By abuse of language, I will here also consider any definition of *p/n* as an instance of *P/n*, or even of $P(\boldsymbol{X})$. A *definition* of a predicate-symbol *p/n* is either the relation description $\mathcal{P}[\boldsymbol{X}]$ of its specification, or the body $\mathcal{B}[\boldsymbol{X}]$ of some logic algorithm for *p/n*. Note that every logic algorithm body is a (first-order logic) relation description, but not vice-versa. As allowed by operation (3) of Definition 3 (schema instantiation), I unfold predicate-variable instances as often as possible into logic algorithms (namely when they are non-recursive conjunctions of literals).

Example 23: The instance of *Minimal*/1 in *LA*(*compress-ext-C*) (LA 3) actually is some predicate-symbol, say *minCompress*/1, such that *minCompress*(*C*) is defined by *C* = [ ]. By abuse of language, the wff *C* = [ ] is also seen as an instance of *Minimal*/1, or even of *Minimal*(*X*), for *LA*(*compress-ext-C*). In *LA*(*compress-ext-C*), the call to *minCompress*/1 has effectively been transformed away by unfolding its definition.

I prefer the verb "decompose" to "divide", and that the "combine" operation is here actually split into a "process" operation and a "compose" operation for non-minimal forms, and renamed "solve" for minimal forms. The reasons for this departure from the "classical" terminology are that the new terminology has more concepts and that it actually reveals the often symmetric roles of parameters *X* and *Y* (see Figure 2).

Also note the convenient parameter naming-scheme suggested by the schema (and actually used throughout this paper): given the user-determined names of the parameters of *R*, all parameters introduced by the schema have names derived from them by prefixing them with either "*H*" (for "head") or "*T*" (for "tail") and possibly suffixing them with integers (to identify them in case there is more than one head or tail). The nouns "head" and "tail" are to be taken in a very general sense here: heads and tails are whatever is obtained through decomposition of parameters, with tails (but not necessarily heads) being of the same type as the decomposed parameter, and tails being used in recursive calls. So the head and tail of a list are not necessarily the first element and remaining elements of that list. This convention and the layout of the schema, if used systematically, allow a very straightforward understanding of my algorithms. Other variables may appear inside the definitions of the predicate-variables of the schema: unless these definitions are themselves covered by some divide-and-conquer schema, the naming of these *internal variables* is unconstrained here.

```
R(X,Y) ⇔
                  Minimal(X)     ∧ Solve(X,Y)
   ∨ ∨_{1≤k≤c}    NonMinimal(X)  ∧ Decompose(X,HX,TX)
                                 ∧ Discriminate_k(HX,TX,Y)
                                 ∧ R(TX,TY)
                                 ∧ Process_k(HX,HY)
                                 ∧ Compose_k(HY,TY,Y)
```

Logic Algorithm Schema 2: Divide-and-conquer (version 2)

For example, the choice of names for the internal variables *E* and *N* of the instantiation of *Process* in *LA*(*compress-int-C*) (LA 4) was arbitrary.

2.2.2 Second Version: Adding Subcases and Discriminants

Many logic algorithms are not covered by Schema 1 because the non-minimal case is further partitioned into subcases, each featuring a different way of combining partial solutions. An enhanced methodology is as follows.

A *divide-and-conquer algorithm* 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 *X* into a vector ***HX*** of heads of *X* and a vector ***TX*** of 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 tails ***TX*** recursively yield tails ***TY*** of *Y*. The heads ***HX*** are processed into a vector ***HY*** of heads of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***. Subcases with different processing and composition operators may emerge: discriminate between them according to the values of ***HX***, ***TX***, *Y*.

If non-determinism of the problem requires alternative solutions, then discriminants should evaluate to true. Discriminants may also be used to make the non-minimal form more precise when there is a single non-minimal case. Logic algorithms synthesized by this enhanced divide-and-conquer methodology are covered by Schema 2. A schema-variable *c* represents the number of different subcases of the non-minimal case. This new schema supersedes the previous schema if *c* is instantiated to 1 and the $Discriminate_1$ predicate-variable is instantiated to *true*.

Example 24: Logic algorithms that are covered by Schema 2, but not by the previous schema, are *LA*(*compress-int-L*) (LA 2) and *LA*(*delOddElems-int-L*) (LA 6).

2.2.3 Third Version: Adding Non-Recursive Cases

Many logic algorithms are not even covered by Schema 2, because the non-minimal case is partitioned into a recursive and a non-recursive case, each of which is in turn partitioned into subcases. In the non-recursive case, *Y* is (usually) easily found by directly solving the problem, taking advantage of the decomposition of *X* into ***HX*** and ***TX***. Let's assume there are *v* non-recursive subcases and *w* recursive subcases, such that $v + w = c$, where *v*, *w* are new schema-variables. Logic algorithms synthesized by this enhanced divide-and-conquer methodology are covered by the following schema:

```
R(X,Y) ⇔
                  Minimal(X)     ∧ Solve(X,Y)
   ∨ ∨_{1≤k≤v}    NonMinimal(X)  ∧ Decompose(X,HX,TX)
                                 ∧ Discriminate_k(HX,TX,Y)
                                 ∧ SolveNonMin_k(HX,TX,Y)
   ∨ ∨_{c-w<k≤c}  NonMinimal(X)  ∧ Decompose(X,HX,TX)
                                 ∧ Discriminate_k(HX,TX,Y)
                                 ∧ R(TX,TY)
                                 ∧ Process_k(HX,HY)
                                 ∧ Compose_k(HY,TY,Y)
```

But this schema is very lengthy, and doesn't sufficiently show the commonalities between the recursive and the non-recursive subcases. I thus syntactically merge these subcases by separating their differences by a second-order exclusive-or connective, denoted "|". The result is Schema 3.

```
R(X,Y) ⇔
              Minimal(X)    ∧ Solve(X,Y)
  ∨ ∨1≤k≤c   NonMinimal(X) ∧ Decompose(X,HX,TX)
                            ∧ Discriminatek(HX,TX,Y)
                            ∧ (     SolveNonMink(HX,TX,Y)
                                |
                                    R(TX,TY)
                                 ∧ Processk(HX,HY)
                                 ∧ Composek(HY,TY,Y)                  )
```

Logic Algorithm Schema 3: Divide-and-conquer (version 3)

```
R(X,Y) ⇔
              Minimal(X)    ∧ Solve(X,Y)
  ∨ ∨1≤k≤c   NonMinimal(X) ∧ Decompose(X,HX,TX)
                            ∧ Discriminatek(HX,TX,Y)
                            ∧ (     SolveNonMink(HX,TX,Y)
                                |
                                    R(TX,TY)
                                 ∧ Processk(HX,HY)
                                 ∧ Composek(HY,TY,Y)                  )
```

Logic Algorithm Schema 4: Divide-and-conquer (version 4)

Example 25: A logic algorithm that is covered by Schema 3, but not by the previous two schemata, is *LA*(*member-int-L*) (LA 8).

2.2.4 Fourth Version: Generalizing to *n*-ary Predicates

Let's now relax the requirement that predicate *r* be binary. But let's keep the (so far implicit) restriction that the induction parameter be simple. Supposing predicate *r* is *n*-ary (where *n* is a schema-variable), this new setting implies that *Y* becomes a vector ***Y*** of $n-1$ variables Y_j, and that vector ***TY*** becomes a vector $\underline{\boldsymbol{TY}}$ of $n-1$ vectors $\boldsymbol{TY}_j$, each of which is a vector of *t* variables TY_{jl} (where *j, l* are new notation-variables). Similarly, ***HY*** becomes a vector $\underline{\boldsymbol{HY}}$ of $n-1$ vectors $\boldsymbol{HY}_j$, each of which is a vector of $h'(j)$ variables HY_{jl}, where $h'/1$ is a new schema function-variable. Thus:

$$\#\boldsymbol{HX} = h$$
$$\#\boldsymbol{HY}_j = h'(j) \quad (1 \le j \le n-1)$$
$$\#\boldsymbol{TX} = \#\boldsymbol{TY}_j = t \quad (1 \le j \le n-1)$$
$$\#\boldsymbol{Y} = \#\underline{\boldsymbol{HY}} = \#\underline{\boldsymbol{TY}} = n-1$$

Logic algorithms synthesized by this enhanced divide-and-conquer methodology are covered by Schema 4, where $\boldsymbol{R}(\boldsymbol{TX},\underline{\boldsymbol{TY}})$ stands for $\wedge_{1\le l\le t}\ R(TX_l,TY_{1l},\ldots,TY_{n-1l})$.

Example 26: Logic algorithms that are covered by Schema 4, but not by the previous three schemata, are *LA*(*efface-int-L*) (LA 7) and *LA*(*plateau-int-N*) (LA 10).

2.2.5 More Versions

Notations are already getting complicated with the fourth version. But covering *LA*(*sort-log-L*) (LA 13) calls for the support of arbitrary numbers of minimal and non-minimal cases (and not just 1 of each), while covering *LA*(*merge-int-⟨A,B⟩*) (LA 9) in addition calls for the support of compound induction parameters, and *LA*(*split*) (LA 14) would then still be uncovered (see [24]).

Another possible enhancement is to capture the idea of adding parameter(s) to the specified predicate and adequately generalizing the specification. This is called *descending generalization* and is discussed by, *e.g.*, Deville [20] and Summers [68]. The well-known non-naive algorithm for *reverse*/2 (namely with an accumulator parameter) is an application of this technique:

```
reverse(L,R) ⇔
     reverse(L,R,[])
```

```
reverse(L,R,A) ⇔
     L=[]    ∧ R=A
   ∨ L=[_|_] ∧ L=[HL|TL]
             ∧ NewA=[HL|A]
             ∧ reverse(L,R,NewA)
```

Note however that the logic algorithm for *reverse*/3 is *not* a divide-and-conquer one: the accumulator is "growing bigger" rather than "smaller". Also, using the "result" parameter or the accumulator parameter as induction parameter would defeat the purpose of the (introduction of the) accumulator. This shows that problems with accumulator parameters can*not* be solved by divide-and-conquer algorithms: another methodology and schema are needed for such problems. Of course, if an accumulator-free problem is given in the first place, then transformation by descending generalization can automatically infer the specification and algorithm for the accumulator-version [20] [21], without having to use that other methodology.

Summers [68] presents another technique of parameter introduction. However, the so-generalized problems *do* have (slightly generalized) divide-and-conquer algorithms. *LA*(*split*) (LA 14) is an example of application of this technique. The underlying schema could still be considered a synthesis schema, rather than a transformation schema, as it is not possible to synthesize a logic algorithm for *split*/3 that is covered by one of the given schemata.

Another observation is that no post-processing operator is applied to parameter *Y*. For instance, specifications such as:

average(*L*,*A*) iff *A* is the average value of *L*,
where *A* is an integer and *L* is a non-empty integer-list.

are beyond the scope of the given schemata, and must be solved via a different approach. The *average*/2 problem could be solved by a top-down decomposition:

```
average(L,A) ⇔
   sum(L,S) ∧ length(L,N) ∧ div(S,N,A)
```

plus optimization by loop-merging the algorithms for *sum*/2 and *length*/2. Another solution would be to extend the divide-and-conquer schemata accordingly, as in [55].

The remainder of this discussion is mostly about the third or fourth version.

2.3 Justifications

A series of decisions have been made during the schema synthesis above. Let me justify them now.

First, *NonMinimal*(*X*) can be replaced by ¬*Minimal*(*X*), but only if this preserves their mutual exclusion over the domain of the induction parameter (see the constraints in the next sub-section).

The distinction between *NonMinimal* and *Decompose* may seem artificial at first sight. Indeed, in many of the logic algorithms of Section 2.1, their instances can be unified, such as in *LA*(*compress-int-L*) (LA 2). But the mission of *NonMinimal* only is to *detect* a non-minimal form, whereas the mission of *Decompose* is to *decompose* a form that is *known* to be non-minimal. This is also reflected in the different parameters to the corresponding predicate-variables. Sample logic algorithms that clearly illustrate these differences are *LA*(*compress-ext-L*) (LA 5), *LA*(*efface-int-L*) (LA 7), and *LA*(*sort-ext-L*) (LA 12).

Note that the elements of ***HX*** (respectively ***HY***) may be of the same type as *X* (respectively *Y*). Indeed, there is no reason why, say, *HL* should be an integer when *L* is a list of integers. This is illustrated by *LA*(*compress-ext-L*) (LA 5), where *HL* is a list.

One may also wonder about the differences between *NonMinimal* and the $Discriminate_k$. The mission of *NonMinimal* is to detect a non-minimal form, *without* considering the value of the induction parameter, whereas the mission of the $Discriminate_k$ is to detect subforms (that lead to subcases), which usually only goes *with* considering the value of the induction parameter. These two functionalities can of course be merged into a single predicate-variable, as shown in Schema 5. This reflects a slightly different methodology, because the notion of *non-primitive form* (which *only* has recursive cases) replaces the notion of non-minimal form (which *may* have non-recursive cases). Note that, even when $c = 1$, $NonPrimitive_k(X,\boldsymbol{Y})$ can here definitely *not* be replaced by $\neg Primitive(X,\boldsymbol{Y})$, because this would prevent the coverage of non-deterministic algorithms. Now, I believe that my original distinction is important. Moreover it allows a da-

```
R(X,Y) ⇔
            Primitive(X,Y)      ∧ SolvePrim(X,Y)
   ∨ ∨v<k≤c  NonPrimitivek(X,Y) ∧ Decompose(X,HX,TX)
                                ∧ R(TX,TY)
                                ∧ Processk(HX,HY)
                                ∧ Composek(HY,TY,Y)
```

Logic Algorithm Schema 5: Divide-and-conquer (version 4')

tabase approach to instantiating *NonMinimal* (because there are not so many possible instances), which is impossible for the $Discriminate_k$ or $NonPrimitive_k$ (because the instances are problem-dependent).

The last two justifications should also clarify the distinction between *Decompose* and the $Discriminate_k$.

Instances of *Solve* may be defined by fairly complex formulas, including divisions into subcases and the corresponding discriminating mechanisms, just as in non-minimal cases. But since this is relatively exceptional, I preferred to keep the schema simple, and always treat such a formula as the instance of a single predicate-variable.

Such is however not the case with the $SolveNonMin_k$, where *Decompose* and $Discriminate_k$ are explicitly present: this is for reasons of symmetry with the recursive non-minimal cases. This even implies that instances of the $SolveNonMin_k$ may use the variables ***HX*** and ***TX*** introduced by *Decompose*, rather than start from scratch with a non-decomposed *X*.

Instances of *Solve* and the $SolveNonMin_k$ are fundamentally different in nature from instances of the $Process_k \wedge Compose_k$ conjunctions. In the former, *Y* can be anything, even totally unrelated to *X*, ***HX***, ***TX***. In the latter, *Y must* be in terms of ***TY*** at least, because otherwise there would be no point in computing ***TY***. So if it turns out impossible to compose *Y* in terms of ***TY***, then the corresponding recursive non-minimal case should be “downgraded” into a non-recursive one before trying to solve *Y* in terms of *X*, ***HX***, ***TX***.

Finally, what is the point in isolating the $Process_k$ from the $Compose_k$? They could indeed be merged into $ProcessCompose_k(\boldsymbol{HX},\boldsymbol{TY},Y)$, which often eliminates the intermediate construction of ***HY***, and human synthesizers tend to do so anyway. But the instantiations of the $Process_k$ are not always trivial or irrelevant, as illustrated by *LA*(*compress-int-C*) (LA 4) and *LA*(*plateau-int-N*) (LA 10), so I believe the distinction is important, especially that it further stresses the symmetry between *X* and *Y*.

In general, an algorithm synthesis strategy (see Section 3.1) might lead to the simultaneous instantiation of *several* predicate-variables, in a way that the identification of the individual instances is not obvious. The covering schema then is a more compact rewriting of the original schema, and similarly for its constraints (see Section 2.4). The schemata and constraints given here have a maximum of information, but they may be tailored to specific strategies by rewriting.

2.4 Constraints on Schema Instances

The schemata above can be instantiated in many ways. However, some constraints need to be verified by schema instances in order to be “valid” divide-and-conquer logic algorithms and to be correct w.r.t. their specifications. Such constraints result from correctness proofs of the schemata. I here list the constraints for version 3 of the schema.

Given the specification $\langle R, dom(\mathcal{R}), \mathcal{R}[X,Y] \rangle$ of the top-level problem, the synthesis of a divide-and-conquer algorithm defining *R* amounts to *first* reducing this specification to a set of $4 + 3\cdot c - v$ ($= 4 + 2\cdot c + w$) new specifications of sub-problems:

$$\langle Minimal, dom(\mathcal{Minimal}), \mathcal{Minimal}[X] \rangle$$

$$\langle NonMinimal, dom(\mathcal{NonMinimal}), \mathcal{NonMinimal}[X] \rangle$$

$$\langle Solve, dom(\mathcal{Solve}), \mathcal{Solve}[X,Y] \rangle$$

$$\langle Decompose, dom(\mathcal{Decompose}), \mathcal{Decompose}[X,\boldsymbol{HX},\boldsymbol{TX}] \rangle$$

$$\langle Discriminate_k, dom(\mathcal{Discriminate}_k), \mathcal{Discriminate}_k[\boldsymbol{HX},\boldsymbol{TX},Y] \rangle \quad (1 \le k \le c)$$

$$\langle SolveNonMin_k, dom(\mathcal{SolveNonMin}_k), \mathcal{SolveNonMin}_k[\boldsymbol{HX},\boldsymbol{TX},Y] \rangle \quad (1 \le k \le v)$$

$$\langle Process_k, dom(\mathcal{Process}_k), \mathcal{Process}_k[\boldsymbol{HX},\boldsymbol{HY}] \rangle \quad (v < k \le c)$$

$$\langle Compose_k, dom(\mathcal{Compose}_k), \mathcal{Compose}_k[\boldsymbol{HY},\boldsymbol{TY},Y] \rangle \quad (v < k \le c)$$

and *then* synthesizing (not necessarily using the divide-and-conquer methodology!) algorithms for these specifications, before *finally* assembling these algorithms according to a divide-and-conquer schema. It is important to realize that instances of the predicate-variables are not (necessarily) directly derived, but are rather the results of auxiliary syntheses from derived specifications. The specification reduction phase features constraints on the derived domain descriptions (Section 2.4.1), constraints on the derived relation descriptions (Section 2.4.2), and constraints on the instances of the schema-variables (Section 2.4.3).

2.4.1 Constraints on the Derived Domains

By merely looking at the schema, we can list the constraints on the domains of the derived specifications:

$$dom(\mathcal{Minimal}) = dom(\mathcal{NonMinimal}) = dom(\mathcal{R}_X) \quad (1)$$

$$dom(\mathcal{Solve}) = dom(\mathcal{R}) \quad (2)$$

$$dom(\mathcal{Decompose}) = dom(\mathcal{R}_X) \times dom(\mathcal{Process}_{\boldsymbol{HX}}) \times dom(\mathcal{R}_X)^t \quad (3)$$

$$dom(\mathcal{Discriminate}_k) = dom(\mathcal{Process}_{\boldsymbol{HX}}) \times dom(\mathcal{R}_X)^t \times dom(\mathcal{R}_Y) \quad (1 \le k \le c) \quad (4)$$

$$dom(\mathcal{SolveNonMin}_k) = dom(\mathcal{Process}_{\boldsymbol{HX}}) \times dom(\mathcal{R}_X)^t \times dom(\mathcal{R}_Y) \quad (1 \le k \le v) \quad (5)$$

$$dom(\mathcal{Process}_k) = dom(\mathcal{Process}_{\boldsymbol{HX}}) \times dom(\mathcal{Process}_{k\ \boldsymbol{HY}}) \quad (v < k \le c) \quad (6)$$

$$dom(\mathcal{Compose}_k) = dom(\mathcal{Process}_{k\ \boldsymbol{HY}}) \times dom(\mathcal{R}_Y)^t \times dom(\mathcal{R}_Y) \quad (v < k \le c) \quad (7)$$

Note that the domain of $\boldsymbol{HX}$ in $\mathcal{Process}_k$ is independent of k, as $\boldsymbol{HX}$ stems from the decomposition of X. The constraints explain why I restricted, in Section 1.3, the domain of a relation to being equal to the Cartesian product of the domains of its parameters.

2.4.2 Constraints on the Derived Relations

The constraints on the relations of the derived specifications are obtained from our knowledge of the divide-and-conquer methodology. They are listed and justified hereafter. Unquantified variables are assumed to be universally quantified. Quantifications are assumed to be over the domain of the quantified variable in the involved relations.

The instance of X, that is the selected induction parameter, must be of an inductive type. This means that

$$type(X,\mathcal{T}) \ \wedge \ \mathcal{T} \in \{integer,\ list,\ tree,\ \ldots,\ intList,\ intTree,\ \ldots\} \quad (8)$$

must be valid. Indeed, the decomposition of X into tails that are each smaller than X according to some wfr would otherwise be impossible, and the divide-and-conquer methodology not applicable.

The minimal and non-minimal forms must be mutually exclusive over the domain of the induction parameter, because otherwise they wouldn't reflect some wfr, say "<", over this domain. This means that

$$\mathcal{Minimal}[X] \veebar \ \mathcal{NonMinimal}[X] \quad (9)$$

$$\text{or, equivalently: } \mathcal{Minimal}[X] \Leftrightarrow \neg\mathcal{NonMinimal}[X]$$

must be valid, where $\veebar$ denotes the exclusive-or connective. Of course, as I shall show in Section 2.5.1 below, a rewriting of the synthesized algorithm may blur this mutual exclusion, which is thus only mandatory during the synthesis of canonical representations of divide-and-conquer logic algorithms.

Only non-minimal forms can be decomposed, and the heads and tails obtained by decomposition must be unique.This means that the formula

$$\mathcal{NonMinimal}[X] \ \Rightarrow \ \exists!\boldsymbol{HX}\,\exists!\boldsymbol{TX} \ \ \mathcal{Decompose}[X,\boldsymbol{HX},\boldsymbol{TX}] \quad (10)$$

must be valid. The pre-condition part is a reasonable constraint, as it facilitates synthesis, especially in conjunction with constraint (9), because no domain-checking literals must then be added for decomposition. The full determinism of decomposition is not necessary for correctness reasons, but avoids redundant logical consequences of the synthesized algorithm.

The decomposition of X must yield tails TX_i that are each smaller than X according to the well-founded relation "<" underlying (9). This means that the formula

$$\mathcal{Decompose}[X,\boldsymbol{HX},TX_1,\ldots,TX_t] \ \Rightarrow \ \exists\text{``<''} \ \ \forall i \in \{1,\ldots,t\} \ \ TX_i \text{ ``<'' } X \quad (11)$$

must be valid. It ensures "termination" of the algorithm in the all-ground mode.

Note that a similar constraint cannot be imposed on the tails of Y, because Y could be an auxiliary parameter and hence undecomposable ($Y = TY$), or Y could be the result parameter of a filtering problem (for example, in *LA*(*delOddElems-int-L*) (LA 6), we have $R = TR$ when $odd(HL)$), or for other reasons.

When the induction parameter X is non-minimal, then at least one of the c subcases must apply after decomposition. This means that the formula

$$\mathcal{D}ecompose[X,\boldsymbol{HX},\boldsymbol{TX}] \;\Rightarrow\; \exists k \in \{1,\ldots,c\}\;\; \mathcal{D}iscriminate_k[\boldsymbol{HX},\boldsymbol{TX},Y] \tag{12}$$

must be valid. It ensures non-determinism of the algorithm, when needed.

Every subcase must be solved correctly. This means that the formulae

$$\mathcal{M}inimal[X] \;\wedge\; \mathcal{S}olve[X,Y] \;\Rightarrow\; \mathcal{R}[X,Y] \tag{13}$$

$$\begin{aligned}&\mathcal{D}ecompose[X,\boldsymbol{HX},\boldsymbol{TX}] \;\wedge\; \mathcal{D}iscriminate_k[\boldsymbol{HX},\boldsymbol{TX},Y]\\ &\wedge\; \mathcal{S}olve\mathcal{N}on\mathcal{M}in_k[\boldsymbol{HX},\boldsymbol{TX},Y] \;\Rightarrow\; \mathcal{R}[X,Y] \quad (1 \le k \le v)\end{aligned} \tag{14}$$

$$\begin{aligned}&\mathcal{D}ecompose[X,\boldsymbol{HX},\boldsymbol{TX}] \;\wedge\; \mathcal{D}iscriminate_k[\boldsymbol{HX},\boldsymbol{TX},Y] \;\wedge\; \mathcal{R}[\boldsymbol{TX},\boldsymbol{TY}]\\ &\wedge\; \mathcal{P}rocess_k[\boldsymbol{HX},\boldsymbol{HY}] \;\wedge\; \mathcal{C}ompose_k[\boldsymbol{HY},\boldsymbol{TY},Y] \;\Rightarrow\; \mathcal{R}[X,Y] \quad (v < k \le c)\end{aligned} \tag{15}$$

must be valid. They ensure partial correctness of the algorithm in the all-ground mode.

Every element of the specified relation $\mathcal{R}$ must be solved by some subcase. This means that the formula

$$\begin{aligned}&\mathcal{R}[X,Y] \;\Rightarrow\; \mathcal{S}olve[X,Y] \;\vee\; \exists(\,\mathcal{D}ecompose[X,\boldsymbol{HX},\boldsymbol{TX}] \;\wedge\\ &(\,\exists k \in \{1,\ldots,v\}\; \mathcal{S}olve\mathcal{N}on\mathcal{M}in_k[\boldsymbol{HX},\boldsymbol{TX},Y]\,) \;\vee\\ &(\,\exists k \in \{v{+}1,\ldots,c\}\; \mathcal{R}[\boldsymbol{TX},\boldsymbol{TY}] \;\wedge\; \mathcal{P}rocess_k[\boldsymbol{HX},\boldsymbol{HY}] \;\wedge\; \mathcal{C}ompose_k[\boldsymbol{HY},\boldsymbol{TY},Y]\,)\,)\end{aligned} \tag{16}$$

must be valid. It ensures completeness of the algorithm in the all-ground mode.

If none of the variables $\boldsymbol{TY}$ is "used" in the instance of some $Compose_k$, then non-minimal subcase k actually is non-recursive: its recursive calls may be dropped, and the instance of $Process_k(\boldsymbol{HX},\boldsymbol{HY}) \wedge Compose_k(\boldsymbol{HY},\boldsymbol{TY},Y)$ may be seen as an instance of $SolveNonMin_k(\boldsymbol{HX},\boldsymbol{TX},Y)$, with $\boldsymbol{HY}$ as internal variables. The schema-variables v and w must then be updated to $v + 1$ and $w - 1$, respectively. This suggests a technique of non-recursive non-minimal case detection: start with $v = 0$ and adjust later.

Similarly, but in general, constraints on the dataflow may be added so as to ensure that the parameters in the schema are actually "used" in computations. For instance, the $\boldsymbol{TY}$ *must* be used in the composition of Y, but the $\boldsymbol{HY}$ are *optional* in that composition. This can be done by parameter dependency graphs based on call-mode information (state of the parameters at call-time), as in the SIERES system of Wirth and O'Rorke [75], or, more powerfully, by construction-mode information (relationship between the parameters of a literal), as in the SYNAPSE system of Flener [24].

2.4.3 Constraints on the Schema-Variables

The constraints on the instances of the schema-variables (for all versions of the schema) are as follows:

$$n \ge 1 \tag{17}$$

$$v \ge 0 \tag{18}$$

$$w \ge 1 \tag{19}$$

$$h \ge 0 \tag{20}$$

$$h' \ge 0, \text{ respectively: } h'(j) \ge 0 \quad (1 \le j \le n-1) \tag{21}$$

$$t \ge 1 \tag{22}$$

Constraint (17) states that 0-ary relations cannot be solved by the divide-and-conquer methodology. Constraints (18) and (19) say that each non-minimal case must have at least one recursive subcase. Constraints (20) to (22) state that every non-minimal form must be decomposable into at least zero heads and one tail.

2.5 Discussion

Let me now discuss various issues around the divide-and-conquer methodology and its schemata. First, in Section 2.5.1, I explain the differences between minimal cases and non-recursive non-minimal cases. Then, in Section 2.5.2, I define auxiliary parameters and comment on how to detect them and how to improve the schemata accordingly. Finally, in Section 2.5.3, I develop the notion of logic algorithm inversion.

2.5.1 Minimal Cases and Non-Recursive Non-Minimal Cases

As the logic algorithms of this paper exhibit, not all non-minimal cases are recursive. Indeed, prefix-traversal logic algorithms have non-recursive non-minimal cases. But some structural cases (for example, the second case of Logic Algorithm 1) seem hard to classify. There are several potential reasons to this.

A first reason for non-recursive non-minimal cases is recursion elimination by partial evaluation. The result is a non-minimal case that looks like a minimal case. This is hard to detect, since the cases still exhibit mutually exclusive structural forms. There is no limit to creating non-recursive non-minimal cases by partial evaluation.

Example 27: *LA*(*compress*) (LA 1) is actually a rewriting of *LA*(*compress-int-L*) (LA 2). Indeed, we have that $TL = [\,]$ iff $TC = [\,]$, so the rewriting is correct. But this rewriting is also misleading as the fact that there really is only one minimal form and only one non-minimal form is not apparent at all.

Another reason is that non-recursive non-minimal cases of prefix-scan logic algorithms can often be merged with their minimal cases. The resulting algorithm looks as if it had no minimal case. It exhibits cases whose structural forms are not mutually exclusive, and is thus easy to detect as being a rewriting of the "canonical" version.

Example 28: *LA*(*efface-int-L*) (LA 7) could be rewritten as follows:

```
efface(E,L,R) ⇔
     L=[_|_]     ∧ L=[HL|TL]
                 ∧ HL=E
                 ∧ E=HL ∧ R=TL
   ∨ L=[_,_|_]   ∧ L=[HL|TL]
                 ∧ HL≠E
                 ∧ efface(TE,TL,TR)
                 ∧ HE=_ ∧ HR=HL
                 ∧ E=TE ∧ E=HE ∧ R=[HR|TR]
```

The form [_|_] is not minimal, as it overlaps with the other form. Its case rather results from a merger of the minimal case and the non-recursive non-minimal case. ♦

It is important to understand that such logic algorithms with non-recursive non-minimal cases are the result of re-writing canonical representations of logic algorithms, rather than unpleasant aberrations.

2.5.2 Auxiliary Parameters

In Section 2.1.2 (*e.g.* in Example 16), I informally introduced the notion of *auxiliary parameter*. Intuitively, an *auxiliary parameter* is a parameter that has nothing to do with the "inductive nature" of the relation. Note that a parameter is auxiliary for a relation, and hence for *all* possible logic algorithms for that relation. Logic algorithm synthesis by induction on an auxiliary parameter is obviously a bad idea. But, so far, I have completely ignored auxiliary parameters. This is justifiable by the observation that the identification of auxiliary parameters is not necessary at all for correct algorithm synthesis. Indeed, as the logic algorithms of Section 2.1.2 show, it *is* possible to write logic algorithms that don't distinguish between auxiliary parameters and "ordinary" parameters. However, the (de)composition of an auxiliary parameter from (into) its heads and tails may look cumbersome because an auxiliary parameter *Y* and its tail *TY* are eventually found to be identical: $Y = TY$. But it is precisely this composition pattern that allows the subsequent detection of auxiliary parameters, and their elimination from the decomposition machinery, so as transform the logic algorithm into a more "graceful" and "natural" version.

Example 29: *LA*(*efface-int-L*) (LA 7) could be rewritten as follows:

```
efface(E,L,R) ⇔
     L=[_]       ∧ L=[HL] ∧ E=HL ∧ R=[]
   ∨ L=[_,_|_]   ∧ L=[HL|TL]
                 ∧ HL=E
                 ∧ E=HL ∧ R=TL
```

```
R(X,Y,Z) ⇔
            Minimal(X)     ∧ Solve(X,Y,Z)
     ∨ ∨1≤k≤c NonMinimal(X)  ∧ Decompose(X,HX,TX)
                           ∧ Discriminatek(HX,TX,Z)
                           ∧ (   SolveNonMink(HX,TX,Y,Z)
                                |
                                 R(TX,TY,Z)
                              ∧ Processk(HX,HY,Z)
                              ∧ Composek(HY,TY,Y,Z)              )
```

Logic Algorithm Schema 6: Divide-and-conquer, with auxiliary parameters (version 4")

```
     ∨  L=[_,_|_] ∧ L=[HL|TL]
                  ∧ HL≠E
                  ∧ efface(E,TL,TR)
                  ∧ HR=HL
                  ∧ R=[HR|TR]                                    ♦
```

I call this *late detection*, because it only allows the transformation of the synthesized logic algorithm, rather than a simplification of its actual synthesis process. The appearance of $Y = TY$ in all recursive cases of a logic algorithm is a formal definition of the notion of auxiliary parameter.

We should however not forget that just a casual glance at a specification will not always tell whether a parameter is an “ordinary” or an auxiliary one. Things are even more difficult with automated algorithm synthesis, and the surest way is indeed to ignore the potential existence of auxiliary parameters until a transformation phase. But suppose now that knowledge about which parameters are auxiliary parameters *is* available *earlier* during the algorithm synthesis process (for instance because the specifier declares them as such, or because type heuristics detect them as such). It would then certainly be helpful to pre-compile the needed subsequent transformations into a schema with an explicit consideration of auxiliary parameters and modified constraints. The benefit would be smaller search spaces. The modified version 4 of the divide-and-conquer schema is Schema 6, where ***Z*** is the vector of auxiliary parameters. Note that ***Y*** has disappeared from the discriminants: this knowledge must otherwise be encoded as a synthesis heuristic.

Now, how could knowledge about which parameters are auxiliary parameters be available *earlier* during the algorithm synthesis? There are basically two solutions:

- *declaration*: the specifier could declare them as such at specification time;
- *early detection*: a parameter that is not of an inductive type must be auxiliary; this heuristic is sound, but not complete. Other (possibly interactive) heuristics could be elaborated.

These solutions allow faster synthesis and more “natural” synthesized algorithms.

2.5.3 Logic Algorithm Inversion

An interesting exercise is to compare logic algorithms synthesized by induction on different parameters, or using different decomposition operators.

For a predicate *r/n* having *X* and *Y* as parameters of an inductive type, it is no surprise that *LA*(*r-int-X*) and *LA*(*r-ext-Y*) are similar, and that *LA*(*r-int-Y*) and *LA*(*r-ext-X*) are similar [20].

Example 30: *LA*(*compress-int-L*) (LA 2) and *LA*(*compress-ext-C*) (LA 3) are quite similar, and so are *LA*(*compress-int-C*) (LA 4) and *LA*(*compress-ext-L*) (LA 5), and so would be *LA*(*sort-ext-L*) (LA 12) and *LA*(*sort-int-S*) (which is not in this paper).

Sometimes, *LA*(*r-int-X*) is similar to *LA*(*r-int-Y*), in addition to *LA*(*r-ext-Y*) and *LA*(*r-ext-X*), namely when there is a one-to-one relationship between *X* and *Y*.

Example 31: *LA*(*plateau-int-N*) (LA 7) would be quite similar to *LA*(*plateau-int-L*) (which is not in this paper), because *N* is the length of *L*. However, *LA*(*compress-int-L*) (LA 2) is not comparable at all to *LA*(*compress-int-C*) (LA 4).

Such pairs of algorithms are called *inversions* of each other. This is often possible, in a relational framework, due to a symmetrical role of parameters *X* and *Y*, and of their (de)composition operators: 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 wfr, which is not necessarily the case for all composition opera-

tors), 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 algorithms. However, constraints (10) and (11) sometimes prevent such an inversion. Indeed, the decomposition mode of *Compose*(***HY***,***TY***,*Y*) might be non-deterministic. For instance, *efface*(*HY*,*Y*,*TY*), *insert*(*HY*,*TY*,*Y*), *append*(TY_1,[$HY \mid TY_2$],*Y*), and *merge*(TY_1,TY_2,*Y*) may not be used as instances of *Decompose*, but are suitable as instances of *Compose*. Or the decomposition mode of *Compose*(***HY***,***TY***,*Y*) might not reflect a well-founded relation. For instance, $Y = TY_i$ may not be used as an instance of *Decompose*, but is suitable as an instance of *Compose*. Such an instance may appear because Y is an auxiliary parameter, or because Y would be part of a compound induction parameter for the inversion (as in Logic Algorithm 9), or because Y is the result parameter of a filtering problem, or for other reasons.

Note that the inversion of a logic algorithm is not necessarily covered by the same schema.

Example 32: *LA*(*compress-int-L*) (LA 2) is covered by version 2, whereas its inversion, *LA*(*compress-ext-C*) (LA 3), is covered by version 1.

2.6 Comparison With Other Synthesis Methodologies

Some issues about the divide-and-conquer methodology need to be discussed in order to show its generality (Section 2.6.1), and to clearly distinguish it from some other methodologies, such as top-down decomposition (Section 2.6.2), the “Deville methodology” (Section 2.6.3), and Global Search (Section 2.6.4).

2.6.1 Precisions about the Divide-and-Conquer Methodology

The divide-and-conquer methodology is often believed to be restricted to algorithms that involve some “sophisticated” synthesis decisions. A famous example is the Quick-Sort algorithm, where the decomposition operator partitions the given list into two sublists of elements that are greater (respectively smaller) than a given pivot. However, such decisions only affect the complexity of the resulting algorithm, and are thus not strictly necessary for the synthesis of correct algorithms. Hence, a decomposition operator that simply (intrinsically) reduces a list into its head and tail is also acceptable. In the sorting problem, this leads to the Insertion-Sort algorithm. In other words, the divide-and-conquer methodology is *not* restricted to the use of extrinsic or logarithmic decomposition, but also encompasses intrinsic decomposition.

Even better, as implied by the phenomenon of logic algorithm inversion (see Section 2.5.3), “sophisticated” synthesis decisions as to the discovery of “good” extrinsic or logarithmic decompositions of a candidate induction parameter can often be discovered automatically, and easily!, by performing an intrinsic decomposition of some other parameter, precisely because the resulting algorithms are inversions of each other! In a functional framework, one would select a “simple” (intrinsic) composition operator for that other parameter and reason backwards, using the constraints, in order to discover such a “good” extrinsic or logarithmic decomposition operator for the input parameter (as shown in [62]). So there seems to be little need to worry about the support of problem-dependent extrinsic or logarithmic decomposition, because the mere support of problem-independent intrinsic decomposition often allows their automatic discovery in case synthesis iterates over all candidate induction parameters. Of course, if both the decomposition and composition operators are not “simple,” then their discovery is more difficult.

Also, I wrote that step (1) of the divide-and-conquer methodology consists of “dividing a problem into sub-problems, unless it can be trivially solved.” I have here taken the option that the “unless it can be trivially solved” clause is applicable iff a minimal form of the domain of the induction parameter is reached. An alternative interpretation would be that the clause may be applicable in even other cases. A good illustration of this point of view is Sedgewick’s enhancement of Hoare’s original Quick-Sort algorithm: it switches to Insertion-Sort once the unsorted list has less than, say, 15 elements.

An apparent disadvantage of the divide-and-conquer methodology is that it seems to lead to logic algorithms, and hence logic programs, that are not tail-recursive, because of the very placement of recursion in the schema. This argument can however be disputed because (automatable) transformation techniques exist for obtaining tail-recursive versions of a program, and this in many cases [20].

2.6.2 The Top-Down Decomposition Methodology

The divide-and-conquer methodology should not be confused with the top-down decomposition methodology, despite their almost synonymic names and their overall problem-reduction approaches. Indeed, the

former is a very precisely defined methodology that, if applicable, always yields recursive algorithms, whereas the latter is a rather vaguely defined methodology that always applies, but without necessarily yielding recursive algorithms.

2.6.3 The "Deville Methodology"

What is the relationship between the divide-and-conquer methodology and the methodology of logic algorithm construction by structural induction, as described by Deville [20]? Let's summarize that methodology first. It is based on structural induction. The principle of well-founded induction suggests synthesizing a logic algorithm by induction over the structure of some parameter. The value of that induction parameter is reduced to something smaller according to some well-founded relation, and a partial result is recursively computed. If no reduction is possible, then the problem is solved directly. There are four steps to this synthesis methodology:

Step 1: Selection of an induction parameter;

Step 2: Selection of a well-founded relation;

Step 3: Selection of the structural forms of the induction parameter;

Step 4: Construction of the structural cases.

A tool, called *Logist*, is being developed to assist a human synthesizer in following these steps [21]. *Logist* is a component of the *Folon* environment [35], which also supports the surrounding stages of specification acquisition and logic program derivation.

Fundamentally, there is no difference between the divide-and-conquer methodology and the Deville one: both are based on the principle of well-founded induction. However, in practice, there are some differences. The divide-and-conquer methodology, especially its formalization as a schema with constraints, is much more prescriptive and much more precisely defined than Deville's methodology. For example, the first-order variables of a divide-and-conquer schema have well-defined scopes. A good illustration of this phenomenon is that the logic algorithms of [20] are not covered by any of my divide-and-conquer schemata. My schema instances usually look a little bit contrived compared to their more natural-looking hand-synthesized counterparts in [20]. But it is precisely this extreme formalization that allows (partial) mechanization of the synthesis process (and many opportunities for subsequent logic algorithm transformation and optimization). The point thus merely is that both methodologies were formulated with different objectives in mind: manual or semi-automated synthesis, respectively any kind of automation in synthesis.

2.6.4 The Global Search Methodology

Global search is an enumerative search that generalizes many known search strategies, such as binary search, backtracking, branch-and-bound, heuristic search, constraint satisfaction, and so on. The class of global search algorithms is described by Smith [63] [64]. In his words: "The basic idea of global search is to […] manipulate sets of candidate solutions [represented intensionally by means of *descriptors*]. The principal operations are to *extract* candidate solutions from a set and to *split* a set into subsets. Derived operations include various *filters* which are used to eliminate sets containing no feasible or optimal solutions. Global search algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts candidates, splits sets, and eliminates sets via filters until no sets remain to be split."

I here cast Smith's functional programming results into the logic programming framework, even though a constraint logic programming reformulation might be more appropriate. I will write $LA(r\text{-}gs)$ to show that $LA(r)$ was synthesized via the global search methodology.

Example 33: Here is a global search logic algorithm for the *member*/4 problem:

```
member(A,N,K,I) ⇔
     A[1]≤K≤A[N] ∧ member_gs(A,K,I,1,N)
```

```
P(X) ⇔
     F(X,d0) ∧ P_gs(X,d0)
P_gs(X,D) ⇔
     Extract(X,D) ∧ 𝒫[X]
   ∨ Split(X,D,E) ∧ F(X,E) ∧ P_gs(X,E)
```

Logic Algorithm Schema 7: Global search (with filter)

```
member_gs(A,K,I,V,W) ⇔
     V=I=W ∧ A[I]=K
   ∨ V<W ∧ W' is (V+W) div 2
           ∧ K≤A[W'] ∧ member_gs(A,K,I,V,W')
   ∨ V<W ∧ V' is 1 + (V+W) div 2
           ∧ A[V']≤K ∧ member_gs(A,K,I,V',W)
```

Logic Algorithm 15: *LA*(*member*/4-*gs*)

This is the well-known binary search algorithm. ♦

In general now, given a specification $\langle P, dom(\mathcal{P}), \mathcal{P}[X] \rangle$ and a filter $F(X,D)$ that eliminates descriptors D that do not contain any solution X, a global search algorithm defining P is covered by Schema 7, where the involved sub-problem is specified as follows:

$P_gs(X,D)$ iff $\mathcal{P}[X] \wedge Satisfies(X,D)$
where $X \in dom(\mathcal{P}) \wedge D \in \hat{\mathcal{D}} \wedge F(X,D)$

The set of *all* solutions is either obtained implicitly upon backtracking, or explicitly (as in Smith's original formulation) by wrapping up the call to *P* in a set constructor, such as *setof*/3 of Prolog.

Filters are only meaningful if *P* is called in a search-mode. For instance, if *A, N, K* are given (index search-mode) for *member*/4, then the filter can be used to eliminate sub-arrays. But if only *A* and *N* are given (element enumeration-mode), then the filter cannot be used, as all elements ought to be enumerated. Technically speaking, "searching something" is thus not a problem, but only a possible mode of a problem.

Global search is distinct from divide-and-conquer, for various reasons [63]. Most importantly, global search is done by iterated disjunctive decomposition (a solution lies in one subset *or* in another), whereas divide-and-conquer algorithms work by recursive conjunctive decomposition (a solution is composed from the solutions to *all* sub-problems). Also, *any* problem can be solved (not necessarily optimally) by global search, but such is not possible with divide-and-conquer (due to constraint (19), we can't just let the minimal case(s) solve the entire problem).

Problems that *do* have divide-and-conquer algorithms (and hence global search ones) are such that there is a homomorphism between algebras on the domains of their parameters [63] [65].

Example 34: In *length*(*L,N*), there is a quite obvious homomorphism between *L* and *N*. A divide-and-conquer algorithm would exploit this, but a global search algorithm would degrade into a generate-and-test algorithm, and hence be non-optimal: computing the length of a list is not optimally done by generating possible lengths and testing whether they are correct or not, because the testing sub-problem is then the same as the overall problem.

Problems that do *not* have divide-and-conquer algorithms (but global search ones) are such that there is no such homomorphism. A global search algorithm would exploit this, as it has less assumptions on the domains of its parameters.

Example 35: In *nQueens*(*N,S*)—which holds iff list *S* is a permutation of list [1,2,…,*N*] such that the ith element of *S* represents the line of the queen placed in the ith column of an $N \times N$ chessboard, such that no queen attacks another one—there is no such homomorphism.

However, my divide-and-conquer schemata and corresponding methodologies (from version 3 on) feature an important and useful deviation from the "classical" divide-and-conquer methodology: the admission of *non-recursive* non-minimal cases (which I showed in Section 2.5.1 to be technically different from minimal cases) allows prefix-traversal algorithms, as well as suffix-traversal algorithms, or, in general, *incomplete-traversal algorithms*, to be covered by my divide-and-conquer schemata. This means that search "problems" can (to a certain extent) be solved by following such a hybrid divide-and-conquer methodology.

Example 36: Consider *member*(*A,N,K,I*) again. It has hybrid divide-and-conquer algorithms, such as sequential search (by intrinsic decomposition), which is a slight specialization of *LA*(*member-int-L*) (LA 8), as well as global search algorithms, such as binary search (see Logic Algorithm 15). The latter has a better complexity in the index search-mode (given *A, N, K*), but they have the same complexity in the element enumeration-mode (given *A, N*), and both are "bad" in the confirmation-mode (given *A, N, K, I*). The conjunctive decomposition of divide-and-conquer algorithms prevents algorithms by extrinsic/logarithmic decomposition of *A*, because an element does not necessarily belong to *all* sub-arrays of *A* obtained by such a decomposition. In other words, Logic Algorithm 15 can*not* be recast as a hybrid divide-and-conquer algorithm, because it would not have recursive non-minimal cases.

Example 37: Now consider *efface*(*E,L,R*). It has hybrid divide-and-conquer algorithms, such as Logic Algorithm 7 (which is by intrinsic decomposition), as well as global search algorithms, such as the following (naive) Logic Algorithm 16:

```
efface(E,L,R) ⇔ {Pre-condition: member(E,L)}
     true ∧ efface(E,[],L,R)

efface(E,P,S,R) ⇔ {Pre-condition: ¬member(E,P)}
     S=[E|T] ∧ append(P,T,R)
   ∨ (S=[F|S'] ∧ F≠E ∧ append(P,[F],P')) ∧ true ∧ efface(E,P',S',R)
```

Logic Algorithm 16: *LA*(*efface-gs*)

The conjunctive decomposition of divide-and-conquer algorithms prevents algorithms by extrinsic/logarithmic decomposition of *L*, because *E* must not be deleted from *all* sub-lists of *L* obtained by such a decomposition. But Logic Algorithm 16 *can* be recast as a hybrid divide-and-conquer algorithm (it is actually already covered by Schema 5), because its instance of *Split* is deterministic (given *P, S, E*). ♦

In general now, every global search algorithm (covered by Schema 7) with a *true* filter *F* and a deterministic instance of *Split* (given ***X*** and *D*) may be recast as a hybrid divide-and-conquer algorithm covered by Schema 5 (and hence by Schema 4, after some rewriting), namely:

```
P(X) ⇔
     P_gs(X,d0)

P_gs(X,D) ⇔
            Extract(X,D)   ∧ 𝒫[X]
     ∨      true           ∧ Split(X,D,E)
                           ∧ P_gs(X,E)
                           ∧ true
                           ∧ true
```

Conversely, every hybrid divide-and-conquer algorithm (covered by Schema 5, and hence by Schema 4, after some rewriting) with decomposition yielding one tail ($t = 1$), with one non-minimal case ($w = 1$), and with *true* processing and composition operators, may be recast as a global search algorithm (covered by Schema 7), namely:

```
R(X,Y) ⇔
     R_gs(X,Y)

R_gs(X,Y) ⇔
     Primitive(X,Y) ∧ SolvePrim(X,Y)
   ∨ ( NonPrimitive1(X,Y) ∧ Decompose(X,HX,TX1) ) ∧ true ∧ R_gs(TX1,Y)
```

For instance, if *E* had been declared an auxiliary parameter, then *LA*(*member-int-L*) (LA 8) would also be a global search algorithm.

To summarize, the class of hybrid divide-and-conquer algorithms has been deliberately designed to overlap with the class of global search algorithms. This hybrid feature of my divide-and-conquer schemata can of course be switched off by simply tightening constraint (18) to $v = 0$.

3 Schema Analysis, or Schema-Guided Algorithm Synthesis

In this section, I first outline a vision of schema-guided logic algorithm synthesis (Section 3.1), and formalize the criteria under which a known predicate may be re-used (Section 3.2). Then I report on particular synthesis strategies that are guided by a divide-and-conquer schema (Section 3.3) or a global search schema (Section 3.4).

3.1 A Vision of Schema-Guided Algorithm Synthesis

An interesting idea is to devise *stepwise* synthesis strategies where each step synthesizes instance(s) of some place-holder(s) of a given algorithm schema. The schema thus actively *guides* the synthesis. 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.

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. Possible modes of reasoning are deductive inference, inductive inference, abductive inference, analogical inference, and so on. Such reasoning would be based on the specifications and the algorithm synthesized so far. Several methods of such a *tool-box* might be applicable at each step, thus yielding opportunities for specifier interaction, or for the application of synthesis heuristics.

I thus here advocate a disciplined approach to algorithm 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 [62]):

(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 synthesis strategies 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 specifier-guided selection of place-holders and methods. My grand view of algorithm synthesis systems 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 algorithms.

Note that this discussion is independent of the used specification formalism, and hence of the specifications' properties (such as their correctness and completeness w.r.t. the intentions).

Backtracking within (the decisions taken at) synthesis steps yields entire families of algorithms for a given specification. Such backtracking may be user-provoked, because s/he wants another algorithm of that family, or synthesizer-provoked, because it encounters an error in an earlier decision or because its methods lack the power to pursue an earlier decision. Thus, if no algorithm is synthesized at all, either the synthesizer is not powerful enough, or the selected schema is inadequate, or the specification is not self-consistent.

Schema-guided synthesis is actually an answer to the combinatorial explosion of search-spaces of both deductive synthesis (transformational synthesis) and proofs-as-programs synthesis (constructive synthesis) [22] [24]. Indeed, the usage of a schema may be seen as the application of a macroscopic transformation rule that embodies very-high-level synthesis decisions. And the usage of a schema may also be seen as a proof tactic that embodies very-high-level algorithm knowledge, rather than just proof knowledge.

3.2 Re-Using Known Predicates

Every top-down synthesis sooner or later reaches sub-problems for which algorithms can be directly obtained by re-using known predicates, rather than by breaking these sub-problems down into sub-sub-problems whose algorithms are then assembled into algorithms for the sub-problems.

Definition 8: A *primitive predicate* (or *primitive*) is not formally specified in terms of other predicates: it is used in the relation description of its own specification. Specifications of primitives are thus of the form:

$R(X)$ iff $R(X)$,
where $X \in dom(\mathcal{R})$.

A *known predicate* is either a non-primitive predicate, whose specification has been made available to the synthesizer, or a primitive predicate. ♦

Every synthesizer must at least have a way of re-using known predicates. This section formalizes the criteria under which a known predicate *P/m* may be re-used for satisfying a logic specification of a predicate *R/n*. There are at least two approaches to do this. In both approaches, we first need to make sure that the domain of *R/n* is a "subset" of the domain of *P/m* (this establishes applicability of *P/m*, because domains are here pre-conditions). Then, in the first approach, we just make sure that all correct solutions to *R/n* actually are solutions to *P/m* (this establishes completeness of using *P/m*). Additional solutions to *P/m* must somehow be filtered out later in order to establish partial correctness of using *P/m*. Formally:

Definition 9: The logic specification $\langle R/n, dom(\mathcal{R}), \mathcal{R}[Y] \rangle$ *reduces to* the logic specification $\langle P/m, dom(\mathcal{P}), \mathcal{P}[X] \rangle$ *under substitution* θ iff the formula:

$$(\mathcal{D}_{\mathcal{R}}[Y] \Rightarrow \mathcal{D}_{\mathcal{P}}[X]\theta) \wedge (\mathcal{R}[Y] \Rightarrow \mathcal{P}[X]\theta) \tag{23}$$

is valid, where $dom(\mathcal{R}) = \{Y \mid \mathcal{D}_{\mathcal{R}}[Y]\}$ and $dom(\mathcal{P}) = \{X \mid \mathcal{D}_{\mathcal{P}}[X]\}$. This definition also applies when the specification of *R/n* is an implication specification. ♦

Note that the first condition cannot be rewritten as $dom(\mathcal{R}) \subseteq dom(\mathcal{P})$, because the arities *m* and *n* are not necessarily the same. This approach is taken in Smith's KIDS synthesizer [63] [64] (see Section 3.4).

Example 38: The logic specification

$insert(E,L,R) \Leftrightarrow permutation([E|L],R) \wedge ordered(L) \wedge ordered(R)$,
where $integer(E) \wedge intList(L) \wedge intList(R)$.

reduces to the logic specification

$efface(F,M,S) \Leftrightarrow \exists A\, \exists B\; append(A,[F|B],M) \wedge \neg member(F,A) \wedge append(A,B,S)$,
where $list(M) \wedge list(S)$.

under substitution $\{F/E, M/R, S/L\}$. Another illustration is given in Example 48. ♦

In the second approach, we immediately look for a condition that filters incorrect solutions to *R/n* out of the solutions to *P/m* (this establishes partial correctness of using *P/m*). If the found condition is the weakest such condition, then completeness of using *P/m* may be established. In general, we are interested in the weakest "possible" such condition (under computational and complexity constraints, say), so as to avoid finding *false* as an obvious condition. Formally:

Definition 10: The logic specification $\langle R/n, dom(\mathcal{R}), \mathcal{R}[Y] \rangle$ *reduces to* the logic specification $\langle P/m, dom(\mathcal{P}), \mathcal{P}[X] \rangle$ *under substitution* θ *and condition* $\mathcal{C}[Y]$ iff the formula:

$$(\mathcal{D}_{\mathcal{R}}[Y] \Rightarrow \mathcal{D}_{\mathcal{P}}[X]\theta) \wedge (\mathcal{P}[X]\theta \wedge \mathcal{C}[Y] \Rightarrow \mathcal{R}[Y]) \tag{24}$$

is valid, where $dom(\mathcal{R}) = \{Y \mid \mathcal{D}_{\mathcal{R}}[Y]\}$ and $dom(\mathcal{P}) = \{X \mid \mathcal{D}_{\mathcal{P}}[X]\}$. This definition also applies when the specification of *R/n* is an implication specification. A logic specification *trivially reduces to* another one iff it reduces to it under some substitution and the *true* condition. ♦

This approach is taken in Smith's CYPRESS synthesizer [62] (see Example 42). The derivation of weakest possible conditions is shown in [24] [27] [60]. This is akin to abduction [40].

Example 39: The logic specification of *insert*/3 reduces to the logic specification of *efface*/3 (see the previous example) under substitution $\{F/E, M/R, S/L\}$ and condition $ordered(L) \wedge ordered(R)$. The condition is the weakest and actually establishes an equivalence. Other illustrations appear in Example 45.

The following theorem states which correctness criteria are achieved through re-use. I here use correctness criteria that are parameterized on a specification semantics *S* and an algorithm semantics *A*, as in [22].

Theorem 1: Let $P(X) \Leftrightarrow \mathcal{B}_{\mathcal{P}}[X]$ be a logic algorithm that is totally correct (w.r.t. *S* and *A*) w.r.t. its logic specification $\langle P/m, dom(\mathcal{P}), \mathcal{P}[X] \rangle$. Then:

(1) If $\langle R/n, dom(\mathcal{R}), \mathcal{R}[Y] \rangle$ reduces to $\langle P/m, dom(\mathcal{P}), \mathcal{P}[X] \rangle$ under substitution θ and condition $\mathcal{C}[Y]$ such that $\mathcal{P}[X]\theta \wedge \mathcal{C}[Y] \Leftrightarrow \mathcal{R}[Y]$, then the logic algorithm $Q(Y) \Leftrightarrow \mathcal{B}_{\mathcal{P}}[X]\theta \wedge \mathcal{C}[Y]$ is totally correct (w.r.t. *S* and *A*) w.r.t. $\langle R/n, dom(\mathcal{R}), \mathcal{R}[Y] \rangle$.

(2) If $\langle R/n, dom(\mathcal{R}), \mathcal{R}[\boldsymbol{Y}] \rangle$ reduces to $\langle P/m, dom(\mathcal{P}), \mathcal{P}[\boldsymbol{X}] \rangle$ under substitution θ and condition $\mathcal{C}[\boldsymbol{Y}]$ such that $\mathcal{P}[\boldsymbol{X}]\theta \wedge \mathcal{C}[\boldsymbol{Y}] \Rightarrow \mathcal{R}[\boldsymbol{Y}]$, then the logic algorithm $Q(\boldsymbol{Y}) \Leftrightarrow \mathcal{B}_{\mathcal{P}}[\boldsymbol{X}]\theta \wedge \mathcal{C}[\boldsymbol{Y}]$ is partially correct (w.r.t. *S* and *A*) w.r.t. $\langle R/n, dom(\mathcal{R}), \mathcal{R}[\boldsymbol{Y}] \rangle$.

Proof 1: Obvious (analogous to the proof of Theorem 5.1 in [62]).

Similar results can be established for the other specification reduction criterion and for implication specifications.

Example 40: Continuing from the previous example, the logic algorithm

```
insert(E,L,R) ⇔
   efface(E,R,L) ∧ ordered(L) ∧ ordered(R)
```

is totally correct (w.r.t. *S* and *A*) w.r.t. its logic specification iff the used logic algorithm *LA*(*efface*) is totally correct (w.r.t. *S* and *A*) w.r.t its own logic specification. ♦

The decisions of *when* to try to re-use and *which* predicates to try to re-use are beyond the scope of this section.

3.3 Divide-and-Conquer Synthesis Strategies

Looking at the sheer variety of possible divide-and-conquer logic algorithms, some of them in Section 2.1, especially the automatic synthesis of logic algorithms looks like quite a formidable task. Indeed, the following difficulties need to be tackled.

What induction parameter to select? How to discover compound induction parameters? How to decompose the induction parameter? For a given problem, there are usually several potential induction parameters, and for a given induction parameter, there are usually many potential decomposition operators. Ideally, a synthesizer should be able to synthesize a whole family of possible algorithms for a given problem. For instance, given the *sort*/2 problem, at least the three logic algorithms of Example 20 should be synthesized.

What are the structural forms? How many structural forms are there? that is: *What well-founded relation will be used in the correctness proof?* While many logic algorithms have one minimal and one non-minimal form, this does not always hold, as illustrated by *LA*(*sort-log-L*) (LA 13) and *LA*(*split*) (LA 14). The type of the induction parameter is not sufficient to infer its structural forms: these actually depend on the domain of the induction parameter, and are thus problem-specific.

Into how many subcases is each structural case divided? How to discriminate between these subcases? Many of the logic algorithms listed in Section 2.1 fork their non-minimal case into subcases.

How to detect that recursion is useless in some non-minimal subcases? Sometimes, the desired result is obtainable before reducing the induction parameter to a minimal form, and no recursion is then needed: this happens for instance in *LA*(*efface-int-L*) (LA 7). The existence or not of useless recursion is dependent on the selected induction parameter.

How to invent or re-use appropriate predicates? How to specify and implement invented predicates? The combination of partial values ***TY*** into the overall value of *Y* often is a full-scale problem by itself, as illustrated by *LA*(*compress-int-C*) (LA 4) and the logic algorithms defining *sort*/2 (see Example 20). The same holds for extrinsic and logarithmic decomposition operators.

How to discover which parameters are auxiliary parameters? Problems such as *efface*/3, *member*/2, and *plateau*/3 have auxiliary parameters: unless this is declared somewhere, considerable synthesis effort may go into detecting this. The logic algorithms listed in Section 2.1 for these problems actually are versions for which neither declaration nor detection was done.

How to achieve a synthesis that yields logic algorithms that are totally correct w.r.t. their specifications and/or intentions? This is a crucial problem.

And so on. The list of challenges is impressive. The answers depend of course a lot on the chosen specification language.

Let's have a look now at several strategies for divide-and-conquer schemata, and then synthesize an algorithm following one of them.

Example 41: Given Schema 4, a possible synthesis strategy would be the following fixed sequence of five steps (vaguely reminiscent of the ones in Section 2.1):

Step 1: Synthesis of *Decompose*;

Step 2: Synthesis of *Minimal* and *NonMinimal*;
Step 3: Syntactic introduction of the recursive atoms;
Step 4: Synthesis of *Solve* and the $SolveNonMin_k$;
Step 5: Synthesis of the $Process_k$, $Compose_k$, and $Discriminate_k$.

This sequence of predicate-variable instantiations is quite "natural". Steps 4 and 5 can be done in parallel. Many alternative strategies exist, however. For instance, one might first synthesize an adequate composition operator (for some parameter *Y*), and then reason backwards in order to infer the corresponding decomposition operator (for parameter *X*). As a result, Step 1 would be interchanged with Step 5. A similar analysis is made by Smith [62] for Schema 5. However, due to the multi-directionality (or reversibility) of the logic algorithms defining the (de)composition operators, I can here claim that the given sequence and its first outlined alternative are isomorphic: synthesizing a composition operator *Compose* for parameter *Y* actually amounts to selecting *Y* as the induction parameter and using *Compose*, in its reversed directionality, as a *de*composition operator for *Y*. In other words, these two strategies are the same. This can also be observed from the dataflow diagram in Figure 2. Note that the recursive atoms need never be "discovered" (synthesized), because they are mandatory and are thus merely syntactically introduced. ♦

Example 42: Smith's CYPRESS synthesizer [62] features three strategies, the first two having been mentioned in the previous example. The third one is as follows (for Schema 5 actually):
Step 1: Synthesis of *Decompose* and the $Compose_k$;
Step 2: Synthesis of *Primitive* and *NonPrimitive*;
Step 3: Syntactic introduction of the recursive atoms;
Step 4: Synthesis of *SolvePrim*;
Step 5: Synthesis of the $Process_k$.

Steps 4 and 5 can be done in parallel. Synthesis starts from a logic specification and deductively sets up logic specifications for sub-problems, and recursively so on, until a re-use method can apply known predicates. ♦

Example 43: Deville's four-step synthesis methodology [20] (see Section 2.6.3) encodes the following strategy (without actually mentioning a schema):
Step 1: Selection of an induction parameter;
Step 2: Synthesis of *Decompose*;
Step 3: Synthesis of *Minimal* and *NonMinimal*;
Step 4: Syntactic introduction of the recursive atoms,
synthesis of *Solve*, the $SolveNonMin_k$, $Process_k$, $Compose_k$, and $Discriminate_k$.

The methodology is meant for synthesis by informal methods, from informal specifications, hence the accumulation in the fourth step. ♦

Example 44: Flener's SYNAPSE synthesizer [24–27] (SYNthesis of Algorithms from PropertieS and Examples) encodes the following strategy (for Schema 4):
Step 1: Syntactic creation of a first approximation;
Step 2: Synthesis of *Minimal* and *NonMinimal*;
Step 3: Synthesis of *Decompose*;
Step 4: Syntactic introduction of the recursive atoms;
Step 5: Synthesis of *Solve* and the $SolveNonMin_k$;
Step 6: Synthesis of the $Process_k$ and $Compose_k$;
Step 7: Synthesis of the $Discriminate_k$.

Steps 5 and 6+7 can be done in parallel. Synthesis starts from assumed-to-be-incomplete formal specifications (by examples and properties) and relies on a tool-box of knowledge-based, inductive, deductive, and abductive methods. ♦

Note that these strategies could be enhanced with synthesis *preferences* and/or *hints* (from extended specifications or through dialogue): a certain induction parameter or decomposition strategy could be preferred to a non-deterministic selection; an auxiliary parameter could be hinted at (because this is likely to speed up the synthesis); once the induction parameter selected, a prefix-traversal could be hinted at; and so on.

Example 45: Let's synthesize a logic algorithm defining *sort*/2, starting from the following logic specification (assuming definitions of the used predicates are known):

sort(*L*,*S*) $\Leftrightarrow$ *permutation*(*L*,*S*) $\wedge$ *ordered*(*S*),
where *intList*(*L*) $\wedge$ *intList*(*S*).

The relation description of this specification already is a legal body for logic algorithms, so synthesis is not strictly necessary. But let's synthesize another, recursive logic algorithm.

Step 0: Select a schema and a strategy. Suppose we select version 4 of the divide-and-conquer schema (Schema 4). We can thus instantiate schema-variable *n* to 2 which satisfies constraint (17), and predicate-variable *R* to *sort*. Suppose we select the first strategy outlined in Example 41. There will thus be five more synthesis steps.

Step 1: Synthesis of *Decompose*. First, we need to select an induction parameter, in accord with constraint (8). Both parameters, *L* and *S*, are suitable candidates, so suppose we select *L*. This instantiates *X* to *L*, and ***Y*** to *S*. A decomposition operator for *L* need not really be synthesized if the type of *L*, namely *intList*, is known. Suppose such is the case, and that we have a knowledge-base with decomposition operators for all decomposition strategies. Suppose then that the intrinsic decomposition strategy is selected, and that *intrDecList*(*A*,*HA*,*TA*) is selected, which is specified as follows:

intrDecList(*A*,*HA*,*TA*) $\Leftrightarrow$ $A = [HA|TA]$,
where *list*(*A*) $\wedge$ *list*(*TA*).

This is a suitable operator, because *intList* is a sub-type of *list*. By abuse of language (see Section 2.2.1), $L = [HL|TL]$ is the instantiation of *Decompose*(*L*,***HX***,***TX***), and ***HX*** is instantiated to *HL*, and ***TX*** to *TL*. This instantiates schema-variables *h* and *t* to 1, which satisfies constraints (20) and (22). Moreover, *intrDecList* encodes the wfr *is-the-tail-of*, hence satisfying constraint (11). Finally, by domain inference and constraint (3), the domains of *HL* and *TL* are *integer* and *intList*, respectively.

Step 2: Synthesis of *Minimal* and *NonMinimal*. Constraints (10) and (1) set up the following logic specification for *NonMinimal*:

NonMinimal(*L*) $\Rightarrow$ $\exists! HL\ \exists! TL\ \ L = [HL|TL]$,
where *intList*(*L*).

This specification trivially reduces to the following supposedly known specification:

pseudoList(*T*) $\Leftrightarrow$ $T = [_|_]$,
where *true*.

hence instantiating *NonMinimal*(*L*) to $L = [_|_]$. Constraints (9) and (1) then set up the following logic specification for *Minimal*:

Minimal(*L*) $\Leftrightarrow$ $\neg L = [_|_]$,
where *intList*(*L*).

This specification trivially reduces to the following supposedly known specification:

emptyList(*T*) $\Leftrightarrow$ $T = [\,]$,
where *true*.

hence instantiating *Minimal*(*L*) to $L = [\,]$.

Step 3: Syntactic introduction of the recursive atoms. A single recursive atom *sort*(*TL*,*TS*) is introduced, hence instantiating ***TY*** to *TS*.

Step 4: Synthesis of *Solve* and the $SolveNonMin_k$. Constraints (13) and (2) set up the following "specification" for *Solve*:

$L = [\,]$ $\wedge$ *Solve*(*L*,*S*) $\Rightarrow$ *permutation*(*L*,*S*) $\wedge$ *ordered*(*S*),
where *intList*(*L*) $\wedge$ *intList*(*S*).

This "specification" can easily be rewritten into a real logic specification, which trivially reduces to the logic specification of *emptyList*/1, hence instantiating *Solve*(*L*,*S*) to $S = [\,]$.

An initial hypothesis is that there are no non-recursive non-minimal cases: $v = 0$, which is the easiest way to satisfy constraint (18). Hence there are no instances to be synthesized for the $SolveNonMin_k$, so constraints (5) and (14) need not be verified. In general (but not in this case), this hypothesis may have to be revised later.

Step 5: Synthesis of the $Process_k$, $Compose_k$, and $Discriminate_k$. Another initial hypothesis is that there is one recursive non-minimal case: $w = 1$, which is the easiest way to satisfy constraint (19). This instantiates c $(= v + w)$ to 1.

Constraints (3), (4), (6), (7), and (15) set up a joint specification for $Process_1$, $Compose_1$, and $Discriminate_1$. Let's call the involved predicate $ProcCompDisc_1$. It has *HL, TL, TS, S* as parameters. Note that ***HY*** can be abstracted away: constraint (21) need thus not be verified. The set up "specification" is:

$$ProcCompDisc_1(HL,TL,TS,S) \wedge permutation(TL,TS) \wedge ordered(TS)$$
$$\Rightarrow permutation([HL|TL],S) \wedge ordered(S),$$
$$\text{where } integer(HL) \wedge intList(TL) \wedge intList(TS) \wedge intList(S).$$

This "specification" trivially reduces to the logic specification of *insert*/3 (see Example 38), hence instantiating $ProcCompDisc_1(HL,TL,TS,S)$ to *insert*(*HL,TS,S*). Constraint (12) need not be verified as no discriminant has been explicitly synthesized. The synthesized logic algorithm is equivalent to *LA*(*sort-int-L*) (LA 11). It is partially correct by virtue of constraints (13) and (15).The verification of constraint (16) is left as an exercise to the reader: it establishes the completeness of the synthesized logic algorithm. Upon backtracking, other logic algorithms can be synthesized. ♦

3.4 A Global Search Synthesis Strategy

In the global search methodology [63] [64], a logic specification is seen as an axiomatic theory, and called a *specification theory*.

Example 46: The *member*/4 problem could be formally specified as follows:

$R = member$

$dom(\mathcal{R}) = \{ \langle A,N,K,I \rangle \mid array(A,1,N,integer) \wedge ordered(A) \wedge integer(N) \wedge integer(K) \wedge integer(I) \}$

$\mathcal{R}[A,N,K,I] \Leftrightarrow A[I] = K \wedge 1 \leq I \leq N$

where $array(A,I,J,T)$ iff A is an array of elements of type T, indexed from I to J. ♦

Definition 11: A global search algorithm is an axiomatic theory, or *algorithm theory*, which represents the algorithm as an 8-tuple:

$$\langle P, dom(\mathcal{P}), \mathcal{P}[X], \hat{\mathcal{D}}, \hat{d}_0(X), Satisfies(X,D), Split(X,D,E), Extract(X,D) \rangle$$

where the first three components (called the *specification-part*) are the same as in specification theories, $\hat{\mathcal{D}}$ is the domain of meaningful descriptors, $\hat{d}_0(X)$ is the descriptor of the initial set of all candidate solutions for X, $Satisfies(X,D)$ is a wff deciding whether candidate solution X satisfies descriptor D, $Split(X,D,E)$ is a wff deciding whether descriptor E represents a subset of the solutions represented by descriptor D with respect to values X, and $Extract(X,D)$ is a wff deciding whether candidate solution X is directly extractable from descriptor D. The last five components of an algorithm theory are called the *algorithm-part*. The constraints (called *axioms*) on the possible instances of the eight components of algorithm theories are as follows. First, *Split* must induce a well-founded ordering over the descriptors:

$$Split(X,D,E) \Rightarrow \exists \text{“}<\text{”} \; E \text{ “}<\text{” } D \tag{25}$$

$$\text{the witness “}<\text{” of (25) is a well-founded relation over } \hat{\mathcal{D}} \tag{26}$$

Next, all solutions must satisfy the initial descriptor:

$$\mathcal{P}[X] \Rightarrow Satisfies(X,\hat{d}_0(X)) \tag{27}$$

Finally, a candidate solution satisfies a descriptor iff it can be extracted after finitely many splits of that descriptor:

$$Satisfies(X,D) \Leftrightarrow \exists E \; Split^*(X,D,E) \wedge Extract(X,E) \tag{28}$$

I here omit the (obvious) definition of $Split^*$. ♦

Example 47: The algorithm theory *gs_binary_split_over_integer_subrange* "encodes the binary split paradigm. It enumerates all elements of a subrange [*L*…*U*] of integers. [Descriptors] correspond to subsubranges and are split roughly in half." Formally:

$P = gs_binary_split_over_integer_subrange$

$dom(\mathcal{P}) = \{ \langle L,U,J \rangle \mid integer(L) \wedge integer(U) \wedge integer(J) \}$

$\mathcal{P}[L,U,J] \Leftrightarrow L \leq J \leq U$

$\hat{\mathcal{D}} = \{ \langle V,W \rangle \mid integer(V) \wedge integer(W) \wedge L \leq V \leq W \leq U \}$
$\hat{d}_0(L,U,J) = \langle L,U \rangle$
$Satisfies(L,U,J,\langle V,W \rangle) \Leftrightarrow V \leq J \leq W$
$Split(L,U,J,\langle V,W \rangle,\langle V',W' \rangle) \Leftrightarrow V < W \wedge [V' = V \wedge W' \text{ is } (V + W) \text{ div } 2 \vee V' \text{ is } 1 + (V + W) \text{ div } 2 \wedge W' = W]$
$Extract(L,U,J,\langle V,W \rangle) \Leftrightarrow V = J = W$

Other such algorithm theories, whose constraints are guaranteed to be satisfied, can be found in [63]. ♦

Definition 12: If a specification theory reduces to the specification-part of an algorithm theory under substitution θ, then the θ-*extension* of the specification theory into a new, concrete algorithm theory is obtained by adding to it the θ-instances of the five components of the algorithm-part of the given, abstract algorithm theory.

Example 48: The specification theory of Example 46 reduces to the specification-part of the abstract algorithm theory *gs_binary_split_over_integer_subrange* (see Example 47) under the substitution θ = {*L*/1, *J*/*I*, *U*/*N*}. Indeed, formula (23) of Definition 9, namely:

$$[array(A,1,N,integer) \wedge ordered(A) \wedge integer(N) \wedge integer(K) \wedge integer(I) \Rightarrow integer(L)\theta \wedge integer(U)\theta \wedge integer(J)\theta] \wedge [A[I] = K \wedge 1 \leq I \leq N \Rightarrow (L \leq J \leq U)\theta]$$

is valid for θ. The obtained concrete algorithm theory is thus as follows:

$R = member$
$dom(\mathcal{R}) = \{ \langle A,N,K,I \rangle \mid array(A,1,N,integer) \wedge ordered(A) \wedge integer(N) \wedge integer(K) \wedge integer(I) \}$
$\mathcal{R}[A,N,K,I] \Leftrightarrow A[I] = K \wedge 1 \leq I \leq N$
$\hat{\mathcal{D}} = \{ \langle V,W \rangle \mid integer(V) \wedge integer(W) \wedge 1 \leq V \leq W \leq N \}$
$\hat{d}_0(1,N,I) = \langle 1,N \rangle$
$satisfies(1,N,I,\langle V,W \rangle) \Leftrightarrow V \leq I \leq W$
$split(1,N,I,\langle V,W \rangle,\langle V',W' \rangle) \Leftrightarrow V < W \wedge [V' = V \wedge W' = (V + W) \text{ div } 2 \vee V' = 1 + (V + W) \text{ div } 2 \wedge W' = W]$
$extract(1,N,I,\langle V,W \rangle) \Leftrightarrow V = I = W$

Note that the algorithm-part does not involve all the parameters of $\mathcal{R}$. This is because abstract algorithm theories may have less parameters than specifications that are reduced to them. ♦

Filters are crucial to the efficiency of global search algorithms, as they are used to eliminate descriptors that do not contain solutions. Using *ideal filters* (which eliminate all useless search by actually deciding whether some descriptor *D* contains any solutions **Y** or not) would be too computationally expensive, so approximations are needed. A *necessary filter F* satisfies the following implication:

$$\forall D \in \hat{\mathcal{D}} \ \forall \mathbf{Y} \in dom(\mathcal{R}) \ [\mathcal{R}[\mathbf{Y}] \wedge Satisfies(\mathbf{Y},D) \Rightarrow F(\mathbf{Y},D)] \quad (29)$$

The contrapositive indicates that descriptors that don't pass the filter cannot contain any solutions. Finding "good" necessary filters is a problem-specific task (because filters are not components of global search theories), and can be done by a generalized theorem prover [60].

Example 49: The derivation of a necessary filter from formula (29) for the concrete algorithm theory obtained in Example 48, namely (after some rewriting):

$$\forall V,W,A,N,K,I \ [integer(V) \wedge integer(W) \wedge 1 \leq V \leq W \leq N \wedge array(A,1,N,integer) \wedge ordered(A) \wedge integer(N) \wedge integer(K) \wedge integer(I) \wedge A[I] = K \wedge 1 \leq I \leq N \wedge V \leq I \leq W \Rightarrow f(A,N,K,I,\langle V,W \rangle)]$$

could yield:

$$f(A,N,K,I,\langle V,W \rangle) \Leftrightarrow A[V] \leq K \leq A[W]$$

assuming that sufficient background knowledge for *array*/4 and *ordered*/1 is available. Indeed, by the contrapositive of the implication above, any ordered sub-array of *A* described by the indices $\langle V,W \rangle$ cannot contain an index *I* such that $A[I] = K$ unless *K* is in that sub-array. ♦

Now, given a filter $F(\mathbf{X},D)$ and an algorithm theory $\langle P, dom(\mathcal{P}), \mathcal{P}[\mathbf{X}], \hat{\mathcal{D}}, \hat{d}_0(\mathbf{X}), Satisfies(\mathbf{X},D), Split(\mathbf{X},D,E), Extract(\mathbf{X},D) \rangle$, the corresponding global search algorithm schema is obtained through Schema 7 (see Section 2.6.4). Note that the predicate-variable *Satisfies* of the algorithm theory is only used

in the specification of the sub-problem *P_gs*, but not in the algorithm for *P_gs*. Synthesis thus "merely" consists of theorem proving tasks. The following synthesis strategy instantiates all place-holders and satisfies all constraints of the schema at the same time:

(1) Reduce the given specification theory to the specification-part of some pre-defined abstract algorithm theory, using formula (23). This reveals a substitution, say θ.
(2) Obtain the corresponding θ-extension, that is, a concrete algorithm theory.
(3) Derive a necessary filter *F* via formula (29) for the obtained algorithm theory.
(4) Assemble the filter *F* and the components of the obtained algorithm theory into a concrete algorithm, according to Schema 7.

Strictly speaking, there is thus no synthesis *per se*, as a generic algorithm is simply instantiated in one of a finite set of predefined ways. Algorithms synthesized in this fashion are complete (by a variant of Theorem 1) and partially correct (by the elimination of incorrect solutions through a "call" to $\mathcal{P}[\boldsymbol{X}]$ "after" *Extract*($\boldsymbol{X}$,$\boldsymbol{D}$)) w.r.t. their logic specifications, but usually not of an optimal complexity, due to the usually high level of genericity of the predefined abstract algorithm theories. So there are usually a lot of optimization opportunities (via simplification, partial evaluation, finite differencing, and so on) [63].

Example 50: For the *member*/4 problem formally specified in Example 46, the first two synthesis steps are illustrated in Example 48. The third synthesis step is shown in Example 49. The final step yields the following logic algorithm:

```
member(A,N,K,I) ⇔
     A[1]≤K≤A[N] ∧ member_gs(A,N,K,I,⟨1,N⟩)

member_gs(A,N,K,I,⟨V,W⟩) ⇔
     V=I=W ∧ A[I]=K ∧ 1≤I≤N
   ∨ split(1,N,I,⟨V,W⟩,⟨V',W'⟩)
             ∧ A[V']≤K≤A[W'] ∧ member_gs(A,N,K,I,⟨V',W'⟩)

split(1,N,I,⟨V,W⟩,⟨V',W'⟩) ⇔
   V<W ∧
   [ V'=V ∧ W' is (V+W) div 2
   ∨ V' is 1 + (V+W) div 2 ∧ W'=W ]
```

This is the well-known binary search algorithm, but there are many simplification and optimization opportunities. An improved version is Logic Algorithm 15. ♦

All these ideas, and more, are implemented in the KIDS (*Kestrel Interactive Development System*) synthesizer of Smith [63] [64].

4 Related Work

Algorithm/program schemata are an old, and ever popular, idea of computer science. They have been proposed for a huge variety of applications, such as proving program properties (Section 4.1), programming tutors (Section 4.2), manual synthesis (Section 4.3), (semi-)automatic synthesis (Section 4.4), and transformation (Section 4.5).

4.1 Proving Program Properties

Some properties of programs can be proven independently of their actual computations. Hence the idea of abstracting away these computational details, and proving such properties for the resulting schemata. This makes proofs of properties at the instance level easier, as it suffices to show that the program is covered by a schema that is known to have the desired property. Sample properties are termination, divergence, equivalence, isomorphism, and so on. An early survey of this research was made by Manna [51]. Note that his schemata are first-order schemata, and that their instances are defined via (Herbrand) interpretations. This semantics-based approach is of course also perfectly acceptable. It is sufficient for the study of schemata, but not as a tool for algorithm synthesis. Indeed, unlike our second-order approach, it doesn't permit the concept of instantiation of a schema, and is thus less "constructive" for synthesis purposes.

4.2 Programming Tutors

In the field of logic programming tutors for beginners, Gegg-Harrison [30] proposes a hierarchy of fourteen logic program schemata. They are second-order logic expressions (as each is the most-specific generalization of a class of programs), and embody the otherwise rare feature of arbitrary arities of predicates. Most are specialized versions of my divide-and-conquer schemata, in the sense that they have less predicate-variables and that they are data-structure specific and already partly instantiated (induction parameter of type *list*, fixed forms, fixed decomposition operators, fixed number of cases, fixed discriminants, etc.).

4.3 Manual Synthesis

In the area of manual or computer-aided algorithm synthesis by experts, Deville and Burnay [20] [21] suggest an ancestor version of my divide-and-conquer schemata. It roughly corresponds to a highly instantiated version 1 (no discriminants, most schema-variables are instantiated to 1).

A similar study is made by O'Keefe [55], who phrases specifications of problems in an algebraic way. The functions of such specifications can be directly plugged into given divide-and-conquer logic program schemata. Several schemata may be applicable according to the properties (associativity, commutativity, existence of left identities, and so on) of the specification's functions.

Yokomori [76] also studies the similarity between logic programs, and indicates how programming problems can be solved by analogy, using his logic program *forms*.

Barker-Plummer [1] discusses a system based on *clichés* that assists experienced programmers in the construction of Prolog programs. His clichés are data-structure- specific second-order sentences. Here, as in the previous two approaches, synthesis is done by successively instantiating a cliché's placeholders. This is in a way related to Kwok and Sergot's *implicit definitions* of logic programs [44].

A different approach is taken by Brna *et al.* [8], Lakhotia [45], Sterling and Kirschenbaum [66], Vasconcelos [72], and many others (see the related work sections of these papers), who synthesize algorithms by repeatedly applying programming techniques to skeletons, which yields *extensions*. Separate extensions of the same skeleton may be *composed* [67] into a single algorithm. *Skeletons* are partly pre-instantiated algorithm schemata defining just a control flow for a specific data-type (such as recursing down or up a data-structure; other examples are parsers and meta-interpreters), and *techniques* are standard programming practices (such as adding extra parameters or computations) that do not alter the control flow of a skeleton. Extensions may themselves be seen as skeletons, to which further techniques may be applied, hence yielding a *stepwise enhancement* methodology. So synthesis is done by explicit manipulation of an entire skeleton, rather than of its individual place-holders. Also, the schemata presented in this paper blur the distinction between skeletons and techniques, which may be inadequate in certain contexts.

Marakakis and Gallagher [52] synthesize logic programs by top-down design, where each refinement step introduces an instance of a data-structure-independent program schema (an abstraction of control flow) or of an abstract datatype operation. They have five schemata, two of which are particular cases of version 1 of my divide-and-conquer schemata, the other three being conjunctive decomposition, disjunctive decomposition, and backtracking.

None of the approaches in the previous two subsections considers constraints on schemata. This also holds for the next subsections, unless otherwise indicated.

4.4 (Semi-)Automatic Synthesis

The field of (semi-)automatic algorithm/program synthesis has naturally seen a lot of interest in schemata. The promise of schema-guidance is a disciplined synthesis that exploits useful knowledge about algorithm synthesis methodologies. I here only discuss approaches to synthesis of logic algorithms/programs from logic specifications with known-to-be-incomplete information about the intentions (Section 4.4.1) or from assumed-to-be-complete information (Section 4.4.2).

4.4.1 Synthesis from Incomplete Specifications

The area of synthesis from incomplete specifications (such as examples) features two sub-areas that take different approaches to using schemata, namely trace-based synthesis and model-based synthesis [24] [22].

Trace-based synthesis. Trace-based synthesis (an early survey is in [61]) from incomplete specifications proceeds in two steps:

(1) *trace generation*: trace(s) are generated from the given example(s);

(2) *trace generalization*: the trace(s) are generalized into a recursive program.

The idea of such a two-step decomposition seems to stem from Siklóssy and Sykes [59]. The first step usually yields trace(s) according to some (divide-and-conquer) schema, such as in [3] [4] [5] [7] [34] [42] [58] [59] [68]. There are two algorithmic approaches to the second step, depending on the number of examples (and hence the number of traces): either a single "long" trace is folded onto itself using a function merging mechanism [3] [4] [5], or a recurrence relation is detected between several "short" traces [68] [42]. The other approaches to the second step are guided by heuristics [7] [34] [58] [59]. The *Basic Synthesis Theorem* of Summers' THESYS system [68] constituted a major breakthrough, as it provided a firm theoretical foundation to synthesis from examples: basically, it encodes a (rather primitive) divide-and-conquer schema and relates it to the generated traces.

Initially restricted to the functional programming paradigm, the trace-based approach seems to have been abandoned in the late 1970s, and was dormant for a decade before it was revived by several researchers in the logic programming community.

Flener's SYNAPSE system [24–27] (also see Example 44) synthesizes logic algorithms from specifications by examples and properties, the latter being disambiguating generalizations of examples. The synthesis is guided by (a hardwired) version 4 of the divide-and-conquer schema of this paper. It features a mix of inductive, abductive, and deductive inference, and follows the tool-box approach outlined in Section 3.1.

Le Blanc [50] formalizes and generalizes Kodratoff *et al.*'s follow-up research (the *BMWk* algorithm [42], with an implicitly underlying simple divide-and-conquer schema) on Summers' ideas [68], but in a term rewriting framework. This is very close to, though a subset of, Flener's approach cited above, but there are no properties, no background knowledge, no constructive induction, and, as of now, rather severe restrictions on the class of synthesizable algorithms (only structural manipulation algorithms). Idestam-Almquist's work on recursive anti-unification [36] is also close to the *BMWk* algorithm.

Ling *et al.*'s CILP system [46] is also related to the above-cited works. Their schemata are, as of now, quite restrictive divide-and-conquer ones. Background knowledge (but not properties) are used in a different way (based on inverting implication) compared to SYNAPSE, but they solve the predicate invention (or: constructive induction) problem in the same way as SYNAPSE, namely by recursively invoking the entire synthesizer on a projected set of new examples.

Hamfelt and Fischer-Nilsson [33] propose METAINDUCE, a meta-programming approach to logic program synthesis from examples: a simple data-structure-specific second-order divide-and-conquer schema is instantiated through a call to a meta-interpreter that proves that the conjunction of examples follows from an instance of that schema. Their system is very close to a subset of SYNAPSE, because there are no properties and the schema is, as of now, rather primitive: in fact, the system can be seen as a very elegant re-implementation of the core of SYNAPSE.

Hagiya [32] also re-formulates Summers' recurrence relation detection mechanism in a logic framework, using higher-order unification in a type theory with a recursion operator. The method is even extended to synthesizing deductive proofs-by-induction from concrete sample proofs.

The work of Jorge and Brazdil [38] [39] is another approach to trace-based synthesis of logic programs. Their specifications by examples are augmented with partial execution traces, called algorithm *sketches*. This is also related to the old techniques of synthesis-from-traces (*e.g.*, Biermann's *Trainable Turing Machine* [2]), which actually underlie the second step of trace-based synthesis from incomplete specifications.

As van Lamsweerde [71] observes, this kind of synthesis from examples actually is a precursor to EBL (*Explanation-Based Learning*), a branch of Machine Learning. Indeed, the goal of EBL in general, and EBG (*Explanation-Based Generalization*) in particular, is the elaboration of a concept description from a very small number of examples, in the presence of much background knowledge. EBL/EBG proceeds by first seeking an explanation of why the example describes an instance of the intended concept (using background knowledge), and then generalizing that explanation. Rules of deductive, abductive, and analogical inference are typically used in EBL/EBG, in contrast to the rules of inductive inference used in empirical learning: EBL/EBG reflect thus analytic learning, as opposed to empirical learning (which is from many examples, with little background knowledge, and only by induction).

Model-based synthesis. Model-based synthesis from incomplete specifications (also known as *Inductive Logic Programming* (ILP) [54]) proceeds by refining a logic program until its least Herbrand model coincides with its intended interpretation. Surprisingly, this strand of research has long ignored the interest of schemata. Shapiro's *Model Inference System* (MIS) [57] was one of the early systems in this approach, but it was almost only a decade later that schemata were found to be interesting for organizing search spaces for such systems. Schemata are a form of syntactic bias (which is any means of restricting hypotheses spaces in learning).

The XOANON system of Tinkham [70] is based on the insight that synthesis need not start from the empty program, but could actually start from the most-specific schema that is believed to be applicable. The search space is thus extended to a second-order space, at the bottom of which are logic programs, and at the top of which are logic program schemata. If synthesis starts from a "good" schema, then the improvement in synthesis speed is shown to be exponential.

A similar approach is taken for the MISST system (MIS with Skeletons and Techniques) of Sterling and Kirschenbaum [66], who develop a new clause generation operator for MIS that is based upon the view of logic programs as skeletons to which programming techniques have been applied (see Section 4.3).

The RDT (*Rule Discovery Tool*) component of Kietz and Wrobel's MOBAL system [41], as well as its predecessor system BLIP, also propose controlling the clause search space by *syntactic rule models* (or schemata) and even by task-oriented *predicate topologies* (which is similar to the structured background knowledge in SYNAPSE). Feng and Muggleton [23] investigate a similar formalism. Contrary to the previously mentioned two approaches, there is not necessarily a focus here on learning recursive concept descriptions (only). As a consequence, the used schemata tend to be quite application-specific (and hence arises the question as to their acquisition) or to cover the entire clause space (and hence to be useless).

This is symptomatic for a lot of ILP research, which caters a lot more to classification (learning of non-recursive concept descriptions) than to synthesis (learning of recursive descriptions). If an ILP-style system is adapted to learn recursive descriptions (only), this often happens through the use of schemata and results in a trace-based synthesizer.

Wirth and O'Rorke [75] stop just short of defining schemata for their SIERES system: they only constrain the dataflow of hypothesis clauses by means of argument dependency graphs (using mode information), but not the control flow nor the parameters.

De Raedt and Bruynooghe's CLINT learner has been augmented with the CIA component [17], which 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 concepts (by trying to group learned conjunctions of literals according to bodies of schemata and asking the user to name, if possible, that concept). This system is however argued to be inadequate for algorithm synthesis, and it only features single-clause schemata anyway.

Tausend [69] proposes a graph-based unifying representation for syntactic biases, including schemata and other, more abstract or more procedural, restrictions that are not surveyed here.

4.4.2 Synthesis from Complete Specifications

In the sub-area of synthesis from complete specifications, the use of schemata seems even less established.

Schemata are one of the pillars of Dershowitz' environment, which supports the evolution of programs. There, (imperative) programs (with assertions in annotations) are constructed and debugged either from scratch, or, better, by analogy-guided modification of another program, or, ideally, by synthesis through instantiation of some schema that abstracts previously constructed programs.

The *Programmer's Apprentice* project [56] was based on the notion of *clichés*, which are application-specific schemata.

The CYPRESS system of Smith [62] (also see Example 42) interactively synthesizes totally correct divide-and-conquer algorithms from logic specifications, and is even able to cope with specifications whose domains are not precisely enough constrained. The underlying schema is as follows:

```
F(X) =
     if Primitive(X)
            then SolvePrim(X)
            else Compose o (G × F) o Decompose(X)
     fi
```

The differences with my Schema 5 (a rewriting of version 4) are that:

- this schema is set in the functional programming paradigm, and hence less powerful, as it distinguishes between input and output parameters and does not allow for partial or multi-valued functions (relations); this is also reflected in the implicit $\neg$*Primitive*(X) instead of a *NonPrimitive*(X);
- there is only one non-minimal (or rather: non-primitive) case ($c = 1$);
- *Decompose* must here yield one head ($h = 1$) and one tail ($t = 1$);
- the processing-operator G may only be F or the identity function *id*;
- the notion of auxiliary parameter is not discussed at all;

To be fair, Smith [62] mentions actually having a more general schema, but I have not had access to it (yet) and don't know whether CYPRESS supports it.

The KIDS (*Kestrel Interactive Development System*) synthesizer of Smith [63] [64] (also see Section 2.6.4 and Section 3.4) interactively synthesizes and optimizes global search, local search, and divide-and-conquer algorithms from logic specifications. Much effort has been put into transformation techniques for optimizing the synthesized algorithms. KIDS is believed to be very close to the break-even point where its usage is more economical than manual algorithm synthesis by an expert.

The previous two systems are noteworthy because they *do* involve constraints on instances. They synthesize functional programs. As of now, I am not aware of any full-fledged schema-guided synthesis systems for logic programs. Example 45 outlines how this could be done (in an approach similar to CYPRESS).

However, the system of Lau and Prestwich [48] can be seen as performing schema-guided synthesis, because the specifier must also produce a *folding problem*, which defines the head and the required recursive calls of the step case clauses of the program to be synthesized. This is akin to giving a partial and data-structure-specific divide-and-conquer schema (where R and *Decompose* are instantiated. and the other predicate-variables are omitted). By means of a non-deterministic choice from a typed database of folding problems, this specifier hint could be eliminated, like in SYNAPSE. In any case, this approach allows the synthesis of entire families of algorithms from a single specification [49].

Also, the *Periwinkle* system of Kraan *et al*. [43] is definitely schema-*based*, but their logic program schemata are not finegrained enough to more effectively *guide* the synthesis. They take a novel approach: a logic program is synthesized as a by-product of the planning of a verification proof of the specification. This proof is performed, at the object level, in a sorted, first-order logic with equality. The proof is however first planned, at the meta-level, while initially having the actual body of the extracted program represented by a second-order variable (this is called *middle-out reasoning*). As the planning proceeds by applying tactics to a conjecture (this is tantamount to applying transformation rules), the program becomes gradually instantiated. This requires an extension of the used *Clam* proof planner. Replacing the second-order variable that represents the entire program by a more sophisticated second-order expression (or schema) should result in schema-guidance, but this hasn't been done yet.

4.5 Transformation

The borderline between logic program synthesis and transformation is quite vague (and a matter of definitions, on which there is no consensus anyway), and similarly for the difference between synthesis schemata and transformation schemata. I here distinguish synthesis and transformation as outlined in Section 1.2.

Deville and Burnay [20] [21] discuss transformation schemata that encode the techniques of structural and computational (descending or ascending) generalization of the initial problem, given an algorithm for that problem. The necessary eureka-discovery techniques are discussed as well.

Fuchs and Fromherz [29] introduce the interesting idea of pre-compiling standard sequences of transformations at the program level into a single abstract transformation at the schema level. A concrete transformation of a program P is then performed by three steps: classification (abstraction) of P into a covering schema S; selection of an applicable transformation schema having S as input and some schema S' as output; instantiation (specialization) of S' into a program P' according to the substitution revealed at the first step.

Only the second step need be interactive, which provides for high-level user decisions, but of course also for heuristic choice.

The work of Waldau [73] is a first step towards proving the validity of transformation schemata.

5 Conclusions and Future Work

In this paper, I have given a brief semi-formal introduction to algorithm schemata. Focusing mostly on the divide-and-conquer methodology, I have incrementally synthesized four increasingly powerful data-structure-independent schemata and their constraints from a series of divide-and-conquer logic algorithms. Finally, I have proposed a vision of stepwise, schema-guided algorithm synthesis mechanisms, where each variable of a schema is instantiated using the best-suited method from a tool-box of such methods. Different synthesis strategies exist for each schema, which allows a lot of flexibility.

I have shown that algorithm schemata are used in a widespread variety of areas, including (semi-)automated algorithm synthesis. This is quite natural, as algorithm schemata are a powerful way of embodying our knowledge about algorithms. Surprisingly, though, schemata are not as broadly used across all existing synthesis approaches as I believe they ought to be. For instance, in the branch of induction-based synthesis, schemata had almost disappeared until recently. One of the reasons could be that Inductive Logic Programming (ILP) in general seems to be more focused on classification (of concepts) than on synthesis (of recursive programs), and it is little wonder that concept schemata are hardly existent, and thus often considered with skepticism. Similarly, in the branch of deduction-based synthesis of *logic* programs, schemata have curiously attracted little attention. I think that deduction-based synthesis paradigms, such as proofs-as-programs synthesis or deductive/transformational synthesis, would gain a lot from schema-guided approaches, as their search spaces can then be significantly pruned. I don't think it is a coincidence that the by far most powerful synthesizer, namely Smith's KIDS [63] [64], is schema-guided.

It must also be observed that most of the research is going into investigating divide-and-conquer schemata. This is not a bad thing in itself, as the covered class of algorithms is very interesting and large. But the published schemata tend to be extremely simplified ones, namely often at best the equivalents of my version 1. The notion of (explicit) constraints on schema instances is mostly absent, hence a loss of an important way of encoding and using programming knowledge. In order to be more useful, I think that more sophisticated and realistic divide-and-conquer schemata, as well as schemata encoding other methodologies, should also be supported and their constraints identified and verified.

Finally, a powerful and standard syntactic representation of schemata and constraints is sorely needed. The work of Tausend [69] goes into this direction, though for a slightly more general purpose (namely a unified representation of syntactic bias in learning). However, her graph-based representation is not very adequate as a means of communication, and thus loses out on the advantages of second-order expressions, which are syntactically very close to first-order algorithms/programs. In view of having a theory of schemata and a formalization of the accompanying notions of instantiation and classification, a language for schemata needs to be developed and its semantics defined. I hope that this paper gives some ideas towards achieving these goals.

Acknowledgments

Many thanks to Yves Deville (Université Catholique de Louvain), for enlightening discussions during the early stages of this research.

References

[1] Dave Barker-Plummer. Cliché programming in Prolog. In M. Bruynooghe (ed), *Proc. of META'90*, pp. 247–256.

[2] Alan W. Biermann. On the inference of Turing machines from sample computations. *Artificial Intelligence* 3(3):181–198, Fall 1972.

[3] Alan W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics* 8(8):585–600, 1978.

[4] Alan W. Biermann. Dealing with search. In [6], pp. 375–392.

[5] Alan W. Biermann and Douglas R. Smith. A production rule mechanism for generating LISP code. *IEEE Transactions on Systems, Man, and Cybernetics* 9(5):260–276, May 1979.

[6] Alan W. Biermann, Gérard Guiho, and Yves Kodratoff (eds). *Automatic Program Construction Techniques*. Macmillan, 1984.

[7] Ted J. Biggerstaff. Design directed synthesis of LISP programs. In [6], pp. 393–420.

[8] P. Brna, Alan Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, D. Robertson, B. Smith, and M. van Someren. Prolog programming techniques. *Instructional Science* 20(2):111–133, 1991.

[9] Manfred Broy. Program construction by transformations: A family tree of sorting programs. In A.W. Biermann and G. Guiho (eds), *Computer Program Synthesis Methodologies*, pp. 1–49. Reidel, 1983.

[10] Alan Bundy. Tutorial notes: Reasoning about logic programs. In G. Comyn, N.E. Fuchs, and M.J. Ratcliffe (eds), *Logic Programming in Action*, pp. 252–277. LNAI 636, Springer-Verlag, 1992.

[11] Alan Bundy, Alan Smaill, and Geraint Wiggins. The synthesis of logic programs from inductive proofs. In J.W. Lloyd (ed), *Proc. of the ESPRIT Symposium on Computational Logic*, pp. 135–149. Springer-Verlag, 1990.

[12] Keith L. Clark. Negation-as-failure. In H. Gallaire and J. Minker (eds), *Logic and Databases*, pp. 293–322. Plenum Press, 1978.

[13] Keith L. Clark and John Darlington. Algorithm classification through synthesis. *The Computer Journal* 23(1):61–65, 1980.

[14] Tim Clement and Kung-Kiu Lau (eds), *Proc. of LOPSTR'*91. Springer-Verlag, 1992.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[16] Pierre De Boeck and Baudouin Le Charlier. Static type analysis of Prolog procedures for ensuring correctness. In P. Deransart and J. Maluszynski (eds), Proc. *of PLILP'*90. LNCS 456:222–237, Springer-Verlag, 1990.

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

[18] Nachum Dershowitz. *The Evolution of Programs*. Purchaser, 1983.

[19] Yves Deville. *A Methodology for Logic Program Construction*. Ph.D. Thesis, Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), 1987.

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

[21] Yves Deville and Jean Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'*89, pp. 409–425. The MIT Press, 1989.

[22] Yves Deville and Kung-Kiu Lau. Logic program synthesis: A survey. *Journal of Logic Programming* 19–20:321–350, May/July 1994.

[23] Cao Feng and Stephen Muggleton. Towards inductive generalization in higher order logic. *Proc. of IWML'*92. Morgan Kaufmann, 1992.

[24] Pierre Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1994.

[25] Pierre Flener and Yves Deville. Towards stepwise, schema-guided synthesis of logic programs. In [14], pp. 46–64.

[26] Pierre Flener and Yves Deville. Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In [37], pp. 67–96.

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

[28] Ria Follet. Combining program synthesis with program analysis. In [6], pp. 91–107.

[29] Norbert E. Fuchs and Markus P.J. Fromherz. Schema-based transformations of logic programs. In [14], pp. 111–125.

[30] Timothy S. Gegg-Harrison. *Exploiting Program Schemata in a Prolog Tutoring System*. Ph.D. Thesis, Duke University, Durham (NC, USA), 1993.

[31] Cordell Green and David R. Barstow. On program synthesis knowledge. *Artificial Intelligence* 10(3):241–270, November 1978.

[32] Masami Hagiya. Programming by example and proving by example using higher-order unification. In M.E. Stickel (ed), *Proc. of CADE'*90. LNCS 449:588–602, Springer-Verlag, 1990.

[33] 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.

[34] Steven Hardy. Synthesis of LISP functions from examples. In *Proc. of IJCAI'*75, pp. 240–245.

[35] Jean Henrard and Baudouin Le Charlier. Folon: An environment for declarative construction of logic programs. In M. Bruynooghe and M. Wirsing (eds), *Proc. of PLILP'*92. LNCS 631:217–231, Springer-Verlag, 1992.

[36] Peter Idestam-Almquist. Recursive anti-unification. *Proc. of ILP*'93 (Technical Report IJS-DP-6707, Jozef Stefan Institute, Ljubljana, Slovenia), pp. 241–253.

[37] Jean-Marie Jacquet (ed). *Constructing Logic Programs*. John Wiley, 1993.

[38] Alípio M. Jorge and Pavel Brazdil. Exploiting algorithm sketches in ILP. In Proc. *of ILP*'93 (Technical Report IJS-DP-6707, Jozef Stefan Institute, Ljubljana, Slovenia), pp. 193–203.

[39] Alípio M. Jorge and Pavel Brazdil. Learning by refining algorithm sketches. In *Proc. of ECAI*'94. 1994.

[40] Andonakis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of Logic and Computation* 2(6):719–770, 1993.

[41] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In [53], pp. 335–359.

[42] Yves Kodratoff and Jean-Pierre Jouannaud. Synthesizing LISP programs working on the list level of embedding. In [6], pp. 325–374.

[43] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In D.S. Warren (ed), *Proc. of ICLP'*93, pp. 441–455. The MIT Press, 1993.

[44] Chor Sang Kwok and Marek Sergot. Implicit definitions of logic programs. In R.A. Kowalski and K.A. Bowen (eds), *Proc. of ICLP'*88, pp. 374–385. The MIT Press, 1988.

[45] Arun Lakhotia. Incorporating "programming techniques" into Prolog programs. In E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'*89, pp. 426–440. The MIT Press, 1989.

[46] 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.

[47] Kung-Kiu Lau. A note on synthesis and classification of sorting algorithms. *Acta Informatica* 27:73–80, 1989.

[48] Kung-Kiu Lau and Steven D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi (eds), *Proc. of ICLP'*90, pp. 667–684. The MIT Press, 1990.

[49] Kung-Kiu Lau and Steven D. Prestwich. Synthesis of a family of recursive sorting procedures. In V. Saraswat and K. Ueda (eds), *Proc. of SLP'*91, pp. 641–658. The MIT Press, 1991.

[50] Guillaume Le Blanc. BMWk revisited: Generalization and formalization of an algorithm for detecting recursive relations in term sequences. In F. Bergadano and L. De Raedt (eds), *Proc. of ECML*'94. LNAI 784:183–197, Springer-Verlag, 1994.

[51] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[52] Emmanouil Marakakis and John P. Gallagher. Schema-based top-down design of logic programs using abstract data types. In L. Fribourg and F. Turini (eds), *Joint Proc. of META'*94 *and LOPSTR'*94. LNCS xxx:ppp–qqq, Springer-Verlag, 1994. Forthcoming.

[53] Stephen Muggleton (ed). *Inductive Logic Programming*. Volume APIC-38, Academic Press, 1992.

[54] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19–20:629–679, May/July 1994.

[55] Richard A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.

[56] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer* 21(11):10–25, November 1988.

[57] Ehud Y. Shapiro. *Algorithmic Program Debugging*. Ph.D. Thesis, Yale University, New Haven (CT, USA), 1982. Published under the same title by the MIT Press, 1983.

[58] David E. Shaw, William R. Swartout, and Cordell Green. Inferring LISP programs from examples. In *Proc. of IJCAI'75*, pp. 260–267.
[59] L. Siklóssy and D.A. Sykes. Automatic program synthesis from example problems. In *Proc. of IJCAI'75*, pp. 268–273.
[60] Douglas R. Smith. Derived preconditions and their use in program synthesis. In D.W. Loveland (ed), *Proc. of CADE'82*, pp. 172–193.
[61] Douglas R. Smith. The synthesis of LISP programs from examples: A survey. In [6], pp. 307–324.
[62] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.
[63] Douglas R. Smith. *The structure and design of global search algorithms*. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto (CA, USA), 1988.
[64] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering* 16(9):1024–1043, September 1990.
[65] Douglas R. Smith. Constructing specification morphisms. In *Journal of Symbolic Computation* 15(5–6):571–606, May/June 1993.
[66] Leon S. Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In [37], pp. 127–140.
[67] Leon S. Sterling and Arun Lakhotia. Composing Prolog meta-interpreters. In R.A. Kowalski and K.A. Bowen (eds), *Proc. of ICLP'88*, pp. 386–403. The MIT Press, 1988.
[68] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM* 24(1):161–175, January 1977.
[69] Birgit Tausend. A unifying representation for language restrictions. *Proc. of ILP*'93 (Technical Report IJS-DP-6707, Jozef Stefan Institute, Ljubljana, Slovenia), pp. 205–220.
[70] Nancy L. Tinkham. *Induction of Schemata for Program Synthesis*. Ph.D. Thesis, Duke University, Durham (NC, USA), 1990.
[71] Axel van Lamsweerde. Learning machine learning. In A. Thayse (ed), *From Natural Language Processing to Logic for Expert Systems*, pp. 263–356. John Wiley, 1991.
[72] Wamberto W. Vasconcelos. Designing Prolog programming techniques. In Y. Deville (ed), *Proc. of LOPSTR*'93, pp. 85–99. Springer-Verlag, 1994.
[73] Mattias Waldau. Formal validation of transformation schemata. In [14], pp. 97–110.
[74] Geraint Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. Apt (ed), *Proc. of JICSLP'92*, pp. 351–365, The MIT Press, 1992.
[75] Rüdiger Wirth and Paul O'Rorke. Constraints for predicate invention. In [53], pp. 299–318.
[76] T. Yokomori. Logic Program Forms. *New Generation Computing* 4:305–320, 1986.