J. Symbolic Computation (1993) **15**, 775–805

Logic Program Synthesis from Incomplete Specifications

PIERRE FLENER AND YVES DEVILLE

Unité d'Informatique, Université Catholique de Louvain
Place Sainte Barbe 2, B – 1348 Louvain-la-Neuve, Belgium

We develop a framework for stepwise synthesis of logic programs from incomplete specifications. After the definition of logic formalisms for specifications and programs, logic program correctness and comparison criteria are proposed. Then we define criteria for upward and downward progression, in order to state strategies for incremental and non-incremental stepwise synthesis. It is shown how these strategies can be applied in practice. Finally, we instantiate the framework on a particular synthesis mechanism that we have developed. Our synthesis system, called SYNAPSE, is non-incremental, both deductive and inductive, and guided by a divide-and-conquer schema.We describe the objectives and methods of the crucial steps, and illustrate them on a sample problem.

1. Introduction

Program synthesis research aims at maximally automating the passage from specifications to programs (see the survey by Biermann, 1992). We define possible formalisms for the starting points (specifications: see Section 1.1) and results (programs: see Section 1.2) of synthesis, and state existing approaches and related work, before pinning down the objectives (Section 1.3) of this paper. Some familiarity with logic programming is assumed.

1.1. SPECIFICATIONS

There are many specification formalisms (natural language, first-order logic, pre/post conditions, algebraic specifications, examples, …). In this paper, we focus on specifications (of logic programs) that are written in first-order logic.

Specifications by examples (an extreme case of logic statements), e.g.:

firstPlateau([*a,a,a,b,b,c,c,c,c*], [*a,a,a*], [*b,b,c,c,c,c*]*)*

lead to synthesis based on inductive inference. Biermann's systems (1979, 1984), and THESYS (Summers, 1977) synthesize LISP functions from positive examples, using matching techniques. Shapiro's MIS (1982) and its derivatives (e.g. (Drabent *et al*., 1988), (Tinkham, 1990)) synthesize Prolog procedures from positive and negative examples, using machine learning techniques. Such specifications have the advantages of naturalness (examples are easy to elaborate, and to understand) and conciseness (examples can implicitly describe manipulations of parameters). Their disadvantages are limited expressive power and ambiguity (examples can't completely specify a problem).

Axiomatizations of a problem in (some subset of) first-order logic, e.g.:

$$\mathit{firstPlateau(List,Plateau,Suffix)} \Leftrightarrow$$
$$\mathit{append(Plateau,Suffix,List)} \wedge \mathit{allEqual(Plateau)} \wedge \mathit{break(Plateau,Suffix)}$$

lead to synthesis based on deductive inference. The systems of, e.g., (Bundy *et al.*, 1990), (Fribourg, 1990), (Wiggins, 1992) perform proofs-as-programs synthesis: programs are extracted from constructive proofs of the satisfiability of specifications. The systems of, e.g., (Clark, 1981), (Hansson, 1980), (Hogger, 1981), (Lau and Prestwich, 1990), perform transformational synthesis: programs are derived from specifications by applications of transformation rules. Such specifications have the advantages of expressiveness (axioms benefit from the full expressive power of logic) and non-ambiguity (axioms can completely specify a problem). Their disadvantages are artificiality (axioms can be difficult to elaborate, and to understand) and length (axioms require a complex formalization process).

It turns out that examples and axioms have complementary strengths and weaknesses. The idea is then to combine both approaches, taking advantage of the pros, while trying to alleviate the cons of each existing approach. This may be achieved by relaxing axioms into *properties*, a potentially incomplete source of information.

Let $\mathfrak{R}$ be the relation one has in mind when elaborating a specification of a procedure for a predicate *r/n*. We call $\mathfrak{R}$ the *intended relation*, in contrast to the relation actually specified, called the *specified relation*, which is what logically follows from the specification. This distinction is very important in general, but crucial with incomplete specifications, where one deliberately admits a gap between the two. We assume the specifier knows $\mathfrak{R}$, even if s/he has no formal definition of it.

DEFINITION 1.1. A *specification by examples and properties* of a procedure for a predicate *r/n* consists of:

- a set $\mathcal{E}(r)$ of examples of *r/n*, partitioned into:
  - a set $\mathcal{E}^{+}(r)$ of positive examples of *r/n* (i.e. ground atoms whose *n*-tuples are supposed to belong to $\mathfrak{R}$);
  - a set $\mathcal{E}^{-}(r)$ of negative examples of *r/n* (i.e. ground atoms whose *n*-tuples are supposed not to belong to $\mathfrak{R}$);
- a set $\mathcal{P}(r)$ of properties (first-order logic statements) of *r/n*. ♦

EXAMPLE 1.1. Non-formally speaking, let the *sum(L,S)* relation hold iff integer *S* is the sum of the elements of the integer list *L*. Let $\mathcal{Sum}$ denote this intended relation. A sample specification by positive examples and properties is:

```
    ℰ(sum) = ℰ⁺(sum) = { sum([],0)
                          sum([1],1)
                          sum([3,2],5)
                          sum([2,6,4],12) }
```

```
𝒫(sum) = {  sum([X],X)
            sum([X,Y],S) ⇐ add(X,Y,S) }
```

where *add(I,J,S)* holds iff integer *S* is the sum of the integers *I* and *J*. ♦

The only syntactic choice so far is the deliberate decision that properties are any first-order logic statements that are not examples.

Very few systems, e.g. (Drabent *et al*., 1988) (De Raedt and Bruynooghe, 1992), start from both examples and properties. Although our considerations also hold for the extreme cases where one of these sets is empty, or where properties are axioms in the above sense, this paper achieves its full relevance only if both sets are non-empty, and if properties are an incomplete source of information.

1.2. LOGIC ALGORITHMS

Since we aim at the synthesis of logic programs with negation, synthesized programs are completed programs. We express programs in a logic formalism close to the one of Deville (1990), called logic algorithms. For syntactic convenience, we here restrict theoretical considerations to binary relations.

DEFINITION 1.2. A *logic algorithm* of a predicate *r*, denoted *LA(r)*, is a formula of the form: *r(X,Y)* ⇔ *Def* [*X,Y*], where *Def* (called the *body*) is a formula that only involves the logical *and* (∧) and *or* (∨) connectives. The left-hand atom is called the *head* of *LA(r)*. The variables *X* and *Y* are called the *universal variables* of *LA(r)*, and all other variables are called the *existential variables* of *LA(r)*. [1]

EXAMPLE 1.2. A sample logic algorithm for *sum*/2 is:

```
LA₂(sum): sum(L,S) ⇔     L=[]       ∧ S=0
                       ∨ L=[HL|TL]  ∧ sum(TL,TS)
                                    ∧ add(HL,TS,S)        ♦
```

Executable Prolog programs can easily be derived from such logic algorithms (Deville, 1990).

EXAMPLE 1.3. The logic program derived from *LA_2(sum)* is the following:

```
sum(L,S) ← L=[],S=0
sum(L,S) ← L=[HL|TL],sum(TL,TS),add(HL,TS,S)
```

or, with the unifications moved into the heads:

```
sum([],0) ←
sum([HL|TL],S) ← sum(TL,TS),add(HL,TS,S)                ♦
```

1. (Predicate) variable names start with an uppercase; functors and predicates start with a lowercase. *F*[*X,Y*] denotes a formula *F* whose free variables are *X* and *Y*; *F*[*a,b*] denotes *F*[*X,Y*] where the free occurrences of *X* and *Y* have been replaced by the terms *a* and *b*, respectively. The variables *X* and *Y* are assumed to be universally quantified over *LA(r)*; other free variables in *Def* are assumed to be existentially quantified over *Def*.

1.3. OBJECTIVES OF THIS PAPER

The objectives of this paper are (*i*) the elaboration of a generic framework for logic algorithm synthesis from specifications by examples and properties, and (*ii*) the description, within this framework, of a particular synthesis mechanism that has been developed and is being implemented. This paper is then organized as follows. In Section 2, we define correctness criteria for specifications and logic algorithms. In Section 3, we propose comparison criteria for logic algorithms. This provides an adequate framework for the formulation, in Section 4, of stepwise synthesis strategies. In Section 5, we instantiate this framework and discuss the design choices used for our synthesis mechanism. In Section 6 to Section 8, we present the objectives and methods of the major steps of our synthesis mechanism, and illustrate them on a sample problem. Finally, in Section 9, we draw some conclusions on the results presented here. This paper extends results presented by Flener and Deville (1992, 1993). A full development of this paper can be found in (Flener, 1993).

2. Correctness of Logic Algorithms

It is important to measure a logic algorithm against its intended relation. Since we are concerned with the declarative semantics of logic algorithms, we define model-theoretic criteria, rather than proof-theoretic ones.

Let $LA(r)$ be $r(X,Y) \Leftrightarrow Def[X,Y]$, and $\mathfrak{R}$ be the intended relation. We here assume that *Def* contains only primitive predicates and possibly *r*. This amounts to assuming that all the predicates involved in the design of $LA(r)$ have been—or will be—correctly implemented, and can thus be seen as primitives for the design of $LA(r)$. This restriction can be overcome by simultaneously considering $LA(r)$ and its non-primitive predicates (Deville, 1990).

The idea behind correctness is to state that the intended relation $\mathfrak{R}$ is equivalent to the relation defined by $LA(r)$:

$$\mathfrak{R} = LA^+(r) \quad \text{with} \quad LA^+(r) = \{<a,b> \mid LA(r) \models r(a,b)\}$$

$$\overline{\mathfrak{R}} = LA^-(r) \quad \text{with} \quad LA^-(r) = \{<a,b> \mid LA(r) \models \neg r(a,b)\}$$

where $\overline{\mathfrak{R}}$ is the complement of $\mathfrak{R}$, and where the considered interpretations are Herbrand interpretations in which the primitive predicates are interpreted according to their specifications. Correctness thus states an equivalence, in the models of $LA(r)$, between the intended relation $\mathfrak{R}$ and the interpretation of predicate *r*. The second criterion, which in general is not a consequence of the first one, is necessary to handle logic algorithms with negation (Deville, 1990).

When a logic algorithm is designed by structural induction (see (Deville, 1990) for a precise methodology for this) on some parameter, then predicate *r* can be interpreted in any Herbrand model of $LA(r)$:

THEOREM 2.1. *If LA(r) is designed by structural induction, then the interpretation of r is the same in all the Herbrand models of LA(r).*

PROOF. *Base case*. In a design by structural induction of *Def*, there exist disjuncts in *Def* that are without recursion. Since all predicates other than *r* have a fixed interpretation in all the Herbrand models, so will the instance of *r*, satisfying the non-recursive disjuncts. *Induction*. Since *LA(r)* is designed by structural induction, in any recursive disjunct the recursive atoms involve parameters that are smaller, according to some well-founded relation, than those in the head. More precisely, for every ground instance of the logic algorithm such that the non-recursive literals in the disjunct are *true*, the recursive literals have smaller parameters than the head of the considered ground instance of the logic algorithm. Hence, by the induction hypothesis, the recursive instance of *r* also has a fixed interpretation in all the Herbrand models. Since the other non-recursive literals have a fixed interpretation, so will *r* for the non-recursive disjuncts. ❑

In the sequel, we only consider recursive logic algorithms where some well-founded relation can be defined between the recursive literals and the head. We thus have to enforce that a synthesis mechanism doesn't synthesize non-terminating recursion (for ground queries).

In this framework, the set $LA^{-}(r)$ is thus the complement of $LA^{+}(r)$. Total correctness reduces to $\mathfrak{R} = LA^{+}(r)$. Partial correctness is achieved when $\mathfrak{R} \supseteq LA^{+}(r)$ (i.e. when the atoms "computed" by *LA(r)* are correct), and completeness is achieved when $\mathfrak{R} \subseteq LA^{+}(r)$ (i.e. when all the correct atoms are "computed" by *LA(r)*).

For convenience, correctness definitions will be formalized wrt a single Herbrand interpretation $\mathfrak{I}$, called the *intended interpretation*, such that the next two conditions hold:

- *r(a,b)* is *true* in $\mathfrak{I}$ iff $\mathfrak{R}(a,b)$ holds,
- $\mathfrak{I}$ is a model of all primitive predicates.

Note that $\mathfrak{I}$ captures $\mathfrak{R}$ since the interpretation of *r* in $\mathfrak{I}$ is $\mathfrak{R}$. So $\mathfrak{R}$ does not have to be explicitly considered in the correctness criteria.

EXAMPLE 2.1. Here are three other logic algorithms for *sum*/2:

LA_1(*sum*): `sum(L,S)` ⇔ `L=[]` ∧ `S=0`

LA_3(*sum*): `sum(L,S)` ⇔ `L=[]` ∧ `S=0`
∨ `length(L,N)` ∧ `N>0` ∧ `sub(S,TS,HL)`

LA_4(*sum*): `sum(L,S)` ⇔ `L=[]` ∧ `S=0`
∨ `L=[HL|TL]`

where *length(L,N)* holds iff integer *N* is the number of elements of list *L*, and *sub(I,J,D)* holds iff *add(J,D,I)* holds. ♦

Three layers of correctness criteria are now defined.

2.1. LOGIC ALGORITHM vs. INTENDED RELATION

Total correctness can now be re-expressed as follows:

DEFINITION 2.1. *LA(r)* is *totally correct* wrt $\mathfrak{R}$ iff $r(X,Y) \Leftrightarrow Def[X,Y]$ is *true* in $\mathfrak{I}$.

One can show that Definition 2.1 is equivalent to the criterion $\mathfrak{R} = LA^{+}(r)$. For partial correctness and completeness, slightly stronger criteria are used:

DEFINITION 2.2. *LA(r)* is *partially correct* wrt $\mathfrak{R}$ iff $r(X,Y) \Leftarrow Def[X,Y]$ is *true* in $\mathfrak{I}$.

DEFINITION 2.3. *LA(r)* is *complete* wrt $\mathfrak{R}$ iff $r(X,Y) \Rightarrow Def[X,Y]$ is *true* in $\mathfrak{I}$.

One can show that Definition 2.2 (respectively Definition 2.3) implies the criterion $\mathfrak{R} \supseteq LA^+(r)$ (respectively $\mathfrak{R} \subseteq LA^+(r)$), but not the converse. This prevents logic algorithms that are "badly" partially correct (respectively "badly" complete), that is logic algorithms that cannot be easily "extended" to totally correct algorithms.

EXAMPLE 2.2. $LA_2(sum)$ is totally correct wrt $\mathcal{Sum}$. $LA_1(sum)$ is only partially correct wrt $\mathcal{Sum}$. $LA_3(sum)$ and $LA_4(sum)$ are only complete wrt $\mathcal{Sum}$.

2.2. LOGIC ALGORITHM vs. SPECIFIED RELATION

Next come criteria for measuring a logic algorithm against its specification by examples and properties. Given a set of examples $\mathcal{E}(r) = \mathcal{E}^+(r) \cup \mathcal{E}^-(r)$, a logic algorithm $LA(r)$ is complete wrt $\mathcal{E}(r)$ iff the examples are covered by the relation defined by $LA(r)$ ($\mathcal{E}^+(r) \subseteq LA^+(r)$ and $\mathcal{E}^-(r) \subseteq LA^-(r)$). And $LA(r)$ is partially correct wrt $\mathcal{E}(r)$ iff the positive examples cover the defined relation ($\mathcal{E}^+(r) \supseteq LA^+(r)$; note that it is meaningless to include here the partial correctness of the negative examples).

Similar criteria can be expressed for a set of properties $\mathcal{P}(r)$. In the above criteria, the sets $\mathcal{E}^+(r)$ and $\mathcal{E}^-(r)$ then have to be replaced by the following two sets:

$$\mathcal{P}^+(r) = \{<a,b> \mid \mathcal{P}(r) \models r(a,b)\}$$
$$\mathcal{P}^-(r) = \{<a,b> \mid \mathcal{P}(r) \models \neg r(a,b)\}$$

The following formalization of all these criteria is defined in terms of the intended interpretation $\mathfrak{I}$. Although slightly different from the above criteria, the following definitions are more adapted to a framework of logic algorithm synthesis (see (Flener, 1993) and (Deville and Flener, 1993) for a precise account on this subject):

DEFINITION 2.4. $LA(r)$ is *complete* wrt $\mathcal{E}(r)$ iff the following conditions hold:
- $r(a,b) \in \mathcal{E}^+(r) \Rightarrow Def[a,b]$ is *true* in $\mathfrak{I}$;
- $r(a,b) \in \mathcal{E}^-(r) \Rightarrow Def[a,b]$ is *false* in $\mathfrak{I}$.

DEFINITION 2.5. $LA(r)$ is *partially correct* wrt $\mathcal{E}(r)$ iff the following condition holds:
- $r(a,b) \in \mathcal{E}^+(r) \Leftarrow Def[a,b]$ is *true* in $\mathfrak{I}$.

DEFINITION 2.6. $LA(r)$ is *complete* wrt $\mathcal{P}(r)$ iff the following conditions hold:
- $\mathcal{P}(r) \models r(a,b) \Rightarrow Def[a,b]$ is *true* in $\mathfrak{I}$;
- $\mathcal{P}(r) \models \neg r(a,b) \Rightarrow Def[a,b]$ is *false* in $\mathfrak{I}$.

DEFINITION 2.7. $LA(r)$ is *partially correct* wrt $\mathcal{P}(r)$ iff the following condition holds:
- $\mathcal{P}(r) \models r(a,b) \Leftarrow Def[a,b]$ is *true* in $\mathfrak{I}$.

DEFINITION 2.8. $LA(r)$ is *totally correct* wrt $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$) iff $LA(r)$ is complete and partially correct wrt $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$).

EXAMPLE 2.3. $LA_2(sum)$, $LA_3(sum)$, and $LA_4(sum)$ are complete wrt $\mathcal{EP}(sum)$.

2.3. SPECIFIED RELATION vs. INTENDED RELATION

Finally, there is consistency of a specification by examples and properties wrt the intended relation. For instance, consistency of the examples $\mathcal{E}(r)$ wrt the intended relation $\mathfrak{R}$ means that the positive examples are in $\mathfrak{R}$, and that the negative examples are in its complement $\overline{\mathfrak{R}}$.

DEFINITION 2.9. $\mathcal{E}(r)$ is *consistent* with $\mathfrak{R}$ iff the following conditions hold:

- $r(a,b) \in \mathcal{E}^+(r) \Rightarrow r(a,b)$ is *true* in $\mathfrak{I}$ (i.e. $\mathcal{E}^+(r) \subseteq \mathfrak{R}$);
- $r(a,b) \in \mathcal{E}^-(r) \Rightarrow r(a,b)$ is *false* in $\mathfrak{I}$ (i.e. $\mathcal{E}^-(r) \subseteq \overline{\mathfrak{R}}$).

DEFINITION 2.10. $\mathcal{P}(r)$ is *consistent* with $\mathfrak{R}$ iff the following condition holds:

- $p \in \mathcal{P}(r) \Rightarrow p$ is *true* in $\mathfrak{I}$ (i.e. $\mathcal{P}^+(r) \subseteq \mathfrak{R}$ and $\mathcal{P}^-(r) \subseteq \overline{\mathfrak{R}}$).

EXAMPLE 2.4. $\mathcal{E}(sum)$ and $\mathcal{P}(sum)$ are consistent with $\mathcal{S}um$.

The specified relation of a consistent specification is a subset of the intended relation. Moreover, if $LA(r)$ is partially or totally correct wrt $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$), and $\mathcal{E}(r)$ (respectively $\mathcal{P}(r)$) is consistent with $\mathfrak{R}$, then $LA(r)$ is partially correct wrt $\mathfrak{R}$.

If there is no formal definition of the intended relation $\mathfrak{R}$, some correctness criteria cannot be applied in a formal way. But they can be used to state features and heuristics of a synthesis mechanism.

3. Comparison of Logic Algorithms

Let $\mathcal{L}_r$ be the set of all possible logic algorithms of r, where the bodies only involve some fixed set of primitive predicates as well as the binary r predicate, and where X and Y are the distinct variables used in the heads.

It is important to compare logic algorithms for the same intended relation. Indeed, this is useful in stepwise synthesis to establish strategies of progression towards a correct algorithm. Let:

- $LA_1(r)$: $r(X,Y) \Leftrightarrow Def_1[X,Y]$
- $LA_2(r)$: $r(X,Y) \Leftrightarrow Def_2[X,Y]$

be two logic algorithms in $\mathcal{L}_r$. We define a criterion for comparing logic algorithms in terms of generality (Section 3.1). Since verifying this criterion is only semi-decidable, we then introduce a sound approximation thereof (Section 3.2).

3.1. SEMANTIC GENERALIZATION

Intuitively, $LA_1(r)$ is less general than $LA_2(r)$ iff Def_1 is "less often" true than Def_2. More formally:

DEFINITION 3.1. $LA_1(r)$ is *less general* than $LA_2(r)$ (denoted $LA_1(r) \leq LA_2(r)$) iff $\forall X \forall Y\, Def_1 \Rightarrow Def_2$ is *true* in $\mathfrak{I}$.

The fact of being *more general* ($\geq$) is defined dually. Two logic algorithms, each more general than the other, are *equivalent* ($\cong$). We use $<$ for $\leq$ and $\ncong$.

EXAMPLE 3.1. We have $LA_1(sum) < LA_2(sum) < LA_3(sum) < LA_4(sum)$.

The set $\mathcal{L}_r$ modulo $\cong$ (denoted $\mathcal{L}_r^{\cong}$) is partially ordered under $\leq$. It includes as least element $\perp_r$ (defined as $r(X,Y) \Leftrightarrow$ *false*, and called *bottom*) and as greatest element $\top_r$ (defined as $r(X,Y) \Leftrightarrow$ *true*, and called *top*). In order to have an upper bound to any ascending sequence of logic algorithms, let's extend $\mathcal{L}_r$ to $\mathcal{M}_r$ by allowing an infinite number of literals in the body of a logic algorithm. Let $\mathcal{U}$ be the considered Herbrand universe. It is clear that $(\mathcal{M}_r^{\cong},\leq)$ is isomorphic to $(\mathcal{P}(\mathcal{U}^2), \subseteq)$, where $\mathcal{P}(\mathcal{S})$ denotes the set of subsets of set $\mathcal{S}$. Hence $(\mathcal{M}_r^{\cong}, \leq)$ is a complete lattice, whose lub operator is the logical *or* ($\vee$), and whose glb operator is the logical *and* ($\wedge$) connective over the bodies of logic algorithms.

Comparing logic algorithms in terms of generality can be a difficult task, and is only semi-decidable anyway. We thus define a particular case of this generality relation, but in terms of purely syntactic criteria.

3.2. SYNTACTIC GENERALIZATION

We represent formulas by multisets, so as to exclude ordering problems. A similar development, though for second-order expressions, but without negation, has been made by Tinkham (1990).

DEFINITION 3.2. Let F be a conjunction of literals (respectively a disjunction of conjunctions of literals). Then $\mu(F)$ is $\emptyset$ if F is the predicate *true* (respectively *false*), and the multiset of the literals of F (respectively the multiset of the conjunctions of literals of F), otherwise.

Let's first define syntactic generalization over conjunctions of literals, and then over logic algorithms:

DEFINITION 3.3. A conjunction C_1 is *syntactically less general* than a conjunction C_2 (denoted $C_1 \ll C_2$) with a substitution θ iff $\mu(C_2\theta) \subseteq \mu(C_1)$.

EXAMPLE 3.2. $p(a,X) \wedge q(Y) \ll p(V,W)$, namely with the substitution $\{V/a, W/X\}$.

DEFINITION 3.4. $LA_1(r)$ is *syntactically less general* than $LA_2(r)$ (denoted $LA_1(r) \ll LA_2(r)$) iff there is a total function ϕ from $\mu(Def_1)$ to $\mu(Def_2)$, such that, for every disjunct D in $\mu(Def_1)$, there is a substitution θ that only binds existential variables of $LA_2(r)$, such that $D \ll \phi(D)$ with substitution θ.

EXAMPLE 3.3. We have $LA_1(sum) \ll LA_2(sum) \ll LA_4(sum)$. However, $LA_2(sum)$ and $LA_3(sum)$ are incomparable under $\ll$, as they involve different predicates.

The fact of being *syntactically more general* ($\gg$) is defined dually. Two logic algorithms, each syntactically more general than the other, are *syntactically equivalent* ($\approx$). Note that syntactical equivalence is more general than alphabetic variance, because of the irrelevance of the ordering of disjuncts within logic algorithms, and of literals within disjuncts.

The set $\mathcal{L}_r^{\approx}$ is partially ordered under $\ll$. The following proposition is a direct consequence of the definitions:

PROPOSITION 3.1. *The relations* $\ll$, $\gg$, *and* $\approx$ *are sub-relations of* $\leq$, $\geq$, *and* $\cong$, *respectively.*

We now define an atomic refinement operator, after making two preliminary observations. A *most general literal* in a disjunct D of $LA(r)$ is of the form $p(Z_1,\ldots,Z_n)$ or $\neg p(Z_1,\ldots,Z_n)$, where p is an n-ary predicate and $Z_1,\ldots,Z_n$ are existential variables occurring exactly once in D. And a *most general term* in a disjunct D of $LA(r)$ is of the form $f(Z_1,\ldots,Z_n)$, where f is an n-ary functor and $Z_1,\ldots,Z_n$ are existential variables occurring exactly once in D.

DEFINITION 3.5. Let γ be a refinement operator such that $LA_2(r) \in \gamma(LA_1(r))$ iff exactly one of the following holds:

- $LA_2(r)$ is derived from $LA_1(r)$ by adding a disjunct to $LA_1(r)$; [2]
- $LA_2(r)$ is derived from $LA_1(r)$ by replacing a disjunct D_1 by D_2, such that:
  - D_2 is D_1 without a most-general literal in D_1; [3]
  - D_2 is D_1 where one or more occurrences of a variable V are replaced by a new existential variable W;
  - D_2 is D_1 where one or more occurrences of a most general term in D_1 are replaced by a new existential variable W.

The ability to add a disjunct of course often overrides the need to modify a disjunct, as it suffices to add the modified disjunct in the first place, for instance when creating a logic algorithm from $\perp_r$. However, this is not always possible, for instance when modifying an existing logic algorithm into another one.

EXAMPLE 3.4. $LA_2(sum) \in \gamma(LA_1(sum))$, and $LA_4(sum) \in \gamma(\gamma(\gamma(LA_2(sum))))$.

Let us now relate the refinement operator γ to the generality relation «:

THEOREM 3.2. *The following three assertions hold:*

(1) γ *is a syntactic generalization operator*: $\forall LA'(r) \in \gamma(LA(r)) \quad LA(r) \ll LA'(r)$;

(2) γ *can generate any syntactic generalization:*
$LA_1(r) \ll LA_2(r) \quad \Leftrightarrow \quad \exists n \ \exists LA_2'(r) \in \gamma^n(LA_1(r)) \ \ LA_2'(r) \approx LA_2(r)$;

(3) γ *can generate all logic algorithms of* $\mathcal{L}_r$ *from* $\perp_r$: $\gamma^*(\perp_r) = \mathcal{L}_r$.

PROOF. Analogous to the proof in (Tinkham, 1990). ❑

An inverse operator σ of γ can also be defined, such that σ is a syntactic specialization operator that can generate all logic algorithms of $\mathcal{L}_r$ from $\top_r$.

4. Stepwise Synthesis Strategies

It is useful to decompose a synthesis process into a series of steps, each designing an intermediate logic algorithm. Indeed, this (*i*) allows different techniques to be deployed at each step (thus enforcing a neat separation of concerns), and (*ii*) yields monitoring points where correctness and comparison criteria can be applied (hence measuring the effectiveness and progression of synthesis).

Stepwise synthesis can be *incremental* (when examples and properties are presented one-by-one, each presentation yielding a run through all synthesis steps), or *non-incremental* (when examples and properties are presented all-at-once, yielding a single run through all synthesis steps).

An interesting approach to stepwise synthesis is to progress towards the desired algorithm while preserving correctness criteria.

Let's give a criterion for upward (partial-correctness preserving) progression:

DEFINITION 4.1. (See Figure 1a.) If the following two conditions hold:

- $LA_2(r) \geq LA_1(r)$,
- $LA_2(r)$ is partially correct wrt $\mathfrak{R}$,

2. By convention, adding a disjunct D to $\perp_r$ amounts to replacing *false* by D.
3. By convention, deleting the unique literal of a singleton disjunct D amounts to replacing D by *true* if there is no *true* disjunct yet in $LA_1(r)$, and to discarding D, otherwise.

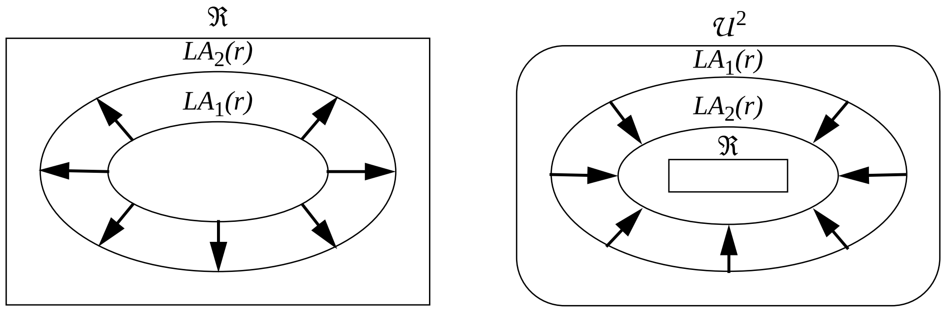

Figure 1. (a) Upward and (b) downward progression

then $LA_2(r)$ is a *better partially correct approximation* of $\mathfrak{R}$ than $LA_1(r)$. ♦

Dually, the criterion for downward (completeness preserving) progression is:

DEFINITION 4.2. (See Figure 1b.) If the following two conditions hold:

- $LA_2(r) \leq LA_1(r)$,
- $LA_2(r)$ is complete wrt $\mathfrak{R}$,

then $LA_2(r)$ is a *better complete approximation* of $\mathfrak{R}$ than $LA_1(r)$. ♦

EXAMPLE 4.1. $LA_3(sum)$ is a better complete approximation of $\mathcal{Sum}$ than $LA_4(sum)$.

We now briefly sketch an incremental synthesis strategy (Section 4.1), and then develop a non-incremental synthesis strategy (Section 4.2).

4.1. AN INCREMENTAL SYNTHESIS STRATEGY

In the case of incremental synthesis, let's view the steps of one synthesis increment as a macro-step performing a transformation Θ. Synthesis is then the design of a series of logic algorithms $LA_0(r), LA_1(r), \ldots, LA_i(r), \ldots$, from a series of specifications $\mathcal{S}_1(r), \ldots, \mathcal{S}_i(r), \ldots$, with $\mathcal{S}_i(r) \subseteq \mathcal{S}_{i+1}(r)$ (where $\mathcal{S}_i(r)$ is a set of examples and properties), such that the following two conditions hold:

- $LA_0(r) = \perp_r$,
- $LA_i(r) = \Theta(LA_{i-1}(r), \mathcal{S}_i(r))$, for $i>0$.

This covers *iterative synthesis*, where only the last presented example or property is actually used by Θ. If Θ is monotonic and continuous (wrt the $\leq$ order on logic algorithms), then $\Theta^{\omega}(\perp_r)$ is its least fixpoint. So if Θ preserves partial correctness wrt $\mathfrak{R}$, then the fixpoint is also partially correct wrt $\mathfrak{R}$. Note that completeness wrt $\mathfrak{R}$ is not necessarily achieved, and that the resulting logic algorithm can involve infinitely many literals. This is related to *identification-in-the-limit*, as first presented in Gold's seminal paper on inductive inference (1967). This incremental strategy is monotonic and consistent. Other approaches are, e.g., non-monotonic synthesis (Jantke, 1991) and inconsistent synthesis (Lange and Wiehagen, 1991).

4.2. A NON-INCREMENTAL SYNTHESIS STRATEGY

A first idea of a non-incremental stepwise synthesis strategy (with a fixed, finite number *f* of steps) is to achieve upward progression:

At **Step 1**, “create” $LA_1(r)$ such that:

• $LA_1(r)$ is partially correct wrt $\mathfrak{R}$.

At **Step *i*** ($2 \le i \le f$), transform $LA_{i-1}(r)$ into $LA_i(r)$ such that:

• $LA_i(r)$ is a better partially correct approximation of $\mathfrak{R}$ than $LA_{i-1}(r)$.

But this strategy doesn’t take care of the completeness aspect. Moreover, since synthesis is here example-based, constants extracted or derived from the examples might appear in intermediate logic algorithms instead of variables, and thus destroy all hopes for completeness. We define a generalization operator that allows the transformation of the series of intermediate logic algorithms into a series of intermediate logic algorithms that reflects downward progression.

We assume that, given $\mathcal{E}(r)$, the literals of a synthesized logic algorithm are partitioned into two classes by the synthesis mechanism: the *synthesized literals*, and the *trailing atoms*. The latter are of the form $V=t$ or $V \in t$, where V is a variable occurring in the synthesized literals, and t is a term or a set of terms textually extracted or derived from the examples in $\mathcal{E}(r)$.

DEFINITION 4.3. Let Γ be a total function in $\mathcal{L}_r$, such that $\Gamma(LA(r))$ is $LA(r)$ whose trailing atoms have been deleted.

EXAMPLE 4.2. Let $LA_5(sum)$ be:

```
sum(L,S) ⇔
        L=[]       ∧   L=[] ∧ S=0
     ∨ L=[HL|TL]  ∧ sum(TL,TS)
                   ∧   HL=1 ∧ S=1 ∧ TL=[] ∧ TS=0
                     ∨ HL=3 ∧ S=5 ∧ TL=[2] ∧ TS=2
                     ∨ HL=2 ∧ S=12 ∧ TL=[6,4] ∧ TS=10
```

where the bold atoms are synthesized atoms, and the other atoms are trailing atoms introduced from $\mathcal{E}(sum)$, as in Example 1.1. Thus, $\Gamma(LA_5(sum))$ is:

```
sum(L,S) ⇔
        L=[]
     ∨ L=[HL|TL]  ∧ sum(TL,TS)
```
♦

It is obvious that Γ can be expressed as a sequence of applications of γ: thus Γ is a generalization function: $LA(r) \ll \Gamma(LA(r))$, hence $LA(r) \le \Gamma(LA(r))$.

The strategy above can now be refined as follows:

At **Step 1**, “create” $LA_1(r)$ such that:

• $LA_1(r)$ is partially correct wrt $\mathfrak{R}$,

• $\Gamma(LA_1(r))$ is complete wrt $\mathfrak{R}$.

At **Step *i*** ($2 \le i \le f$), transform $LA_{i-1}(r)$ into $LA_i(r)$ such that:

• $LA_i(r)$ is a better partially correct approximation of $\mathfrak{R}$ than $LA_{i-1}(r)$,

• $\Gamma(LA_i(r))$ is a better complete approximation of $\mathfrak{R}$ than $\Gamma(LA_{i-1}(r))$.

At **Step *f***, “obtain” $LA_f(r)$ such that:

• $LA_f(r) \cong \Gamma(LA_f(r))$.

Thus, $LA_f(r)$ is totally correct wrt $\Re$. Hence, convergence of the synthesis process is achieved.

We now state and prove a generic theorem showing how steps 2 to f–1 of the above generic strategy can be refined in order to obtain a practical framework:

THEOREM 4.1. GENERIC SYNTHESIS THEOREM.
Let $LA(r)$ be $r(X,Y) \Leftrightarrow \vee_{1 \leq j \leq m} A_j$, and $LA'(r)$ be $r(X,Y) \Leftrightarrow \vee_{1 \leq j \leq m} A_j \wedge B_j$, where A_j and B_j are any formulas. The following two assertions hold:

(1) *If $LA(r)$ is partially correct wrt $\Re$ and $A_j \Rightarrow B_j \quad (1 \leq j \leq m)$ then $LA'(r)$ is a better partially correct approximation of $\Re$ than $LA(r)$.*

(2) *If $LA(r)$ is complete wrt $\Re$ and $\Re(X,Y) \wedge A_j \Rightarrow B_j \quad (1 \leq j \leq m)$ then $LA'(r))$ is a better complete approximation of $\Re$ than $LA(r)$.*

PROOF. Let's prove these assertions one by one:

(1) Obviously, we have $\vee_{1 \leq j \leq m} A_j \wedge B_j \Rightarrow \vee_{1 \leq j \leq m} A_j$. Moreover, the second hypothesis implies $\vee_{1 \leq j \leq m} A_j \Rightarrow \vee_{1 \leq j \leq m} A_j \wedge B_j$. Thus $LA'(r) \cong LA(r)$, i.e., in particular $LA'(r) \geq LA(r)$. Using the first hypothesis, we obtain that $LA'(r)$ is partially correct wrt $\Re$. Thus: $LA'(r)$ is a better partially correct approximation of $\Re$ than $LA(r)$.

(2) By Definition 2.3, the first hypothesis reads $r(X,Y) \Rightarrow \vee_{1 \leq j \leq m} A_j$. By the definition of $\Im$, the second hypothesis reads $(1 \leq i \leq m)\ r(X,Y) \wedge A_i \Rightarrow B_i$, or, equivalently $r(X,Y) \Rightarrow \neg A_i \vee B_i$. Combined with the first hypothesis, we get $r(X,Y) \Rightarrow (\vee_{1 \leq j \leq m} A_j) \wedge (\wedge_{1 \leq i \leq m} \neg A_i \vee B_i)$. The right-hand side can be rearranged as $\vee_{1 \leq j \leq m} A_j \wedge B_j \wedge C_j$, where C_j is a formula involving $\neg A_i$ and B_i $(1 \leq i \leq m,\ i \neq j)$. Hence $r(X,Y) \Rightarrow \vee_{1 \leq j \leq m} A_j \wedge B_j$, i.e. $LA'(r)$ is complete wrt $\Re$. By construction, we have $LA'(r) \ll LA(r)$, i.e., by Theorem 3.1, $LA'(r)) \leq LA(r)$. Thus $LA'(r)$ is a better complete approximation of $\Re$ than $LA(r)$. ❑

The second hypothesis of assertion (1) ensures that the introduced literals are redundant with the already existing ones. In other words, as the proof shows, we then actually have $LA'(r) \cong LA(r)$. But strict progression is achieved by the generalizations. The second hypothesis of assertion (2) ensures that the introduced literals are “redundant” with the intended relation $\Re$.

In practice, assertion (1) is applied to the logic algorithms $LA_i(r)$, whereas assertion (2) is applied to the logic algorithms $\Gamma(LA_i(r))$, where i>1. The first hypotheses of both assertions need not be proved if they are established by Step 1 and then preserved by application of Theorem 4.1 to all previous steps. Proving the second condition of assertion (2) can't be done in a formal way for lack of a formal definition of $\Re$. However, this can be used to guide a synthesis mechanism, for instance by means of interaction with the specifier, hence increasing the confidence in the synthesis.

5. A Particular Synthesis Mechanism

We now instantiate the general framework above to the particular synthesis mechanism we have developed. We first justify some design choices made for this mechanism (Section 5.1), and then discuss algorithm schemata as a means to guide synthesis (Section 5.2), before outlining the mechanism itself (Section 5.3).

```
ℰ(firstPlateau) = {  firstPlateau([a],[a],[])                (E1)
                     firstPlateau([b,b],[b,b],[])            (E2)
                     firstPlateau([c,d],[c],[d])             (E3)
                     firstPlateau([e,f,g],[e],[f,g])         (E4)
                     firstPlateau([h,i,i],[h],[i,i])         (E5)
                     firstPlateau([j,j,k],[j,j],[k])         (E6)
                     firstPlateau([m,m,m],[m,m,m],[]) }      (E7)
𝒫(firstPlateau) = {  firstPlateau([X],[X],[])                (P1)
                     firstPlateau([X,Y],[X,Y],[]) ⇐ X=Y      (P2)
                     firstPlateau([X,Y],[X], [Y]) ⇐ X≠Y }    (P3)
```

Figure 2. Sample versions of *ℰ(firstPlateau)* and *𝒫(firstPlateau)*

```
firstPlateau(L,P,S) ⇔
    L=[HL]           ∧ P=L ∧ S=[]
  ∨ L=[HL1,HL2|TL]   ∧ HL1≠HL2 ∧ P=[HL1] ∧ S=[HL2|TL]
  ∨ L=[HL1,HL2|TL]   ∧ HL1=HL2
                     ∧ firstPlateau([HL2|TL],TP,TS)
                     ∧ P=[HL1|TP] ∧ S=TS
```

Figure 3. A sample version of *LA(firstPlateau)*

5.1. DESIGN CHOICES

Specification language. Specifications here consist of a non-empty set of positive examples, and a possibly empty set of properties that are Horn clauses. Negative examples are not used. Although the mechanism could handle recursive clauses, properties are limited to be non-recursive, so as to focus on synthesis from incomplete specifications. With recursive properties, a transformational or constructive approach would be more appropriate than our approach.

EXAMPLE 5.1. Non-formally speaking, let the *firstPlateau(L,P,S)* relation hold iff list *P* is the first plateau (maximal sequence of identical elements) at the beginning of non-empty list *L*, and list *S* is the corresponding suffix of *L*. A sample specification by examples and properties of this relation is given in Figure 2. Note that properties P_1 to P_3 generalize examples E_1 to E_3, respectively.

EXAMPLE 5.2. A sample version of *LA(firstPlateau)* is given in Figure 3.

Degree of automation. With incomplete specifications it is more realistic to strive for an interactive synthesis mechanism, so as to explicitly disambiguate some situations, rather than to have a default iteration over all possible decisions.

5.2. ALGORITHM SCHEMATA

Algorithms can be classified according to their design strategies, such as divide-and-conquer, generate-and-test, global search, and so on. It is thus interesting to guide the design process by an algorithm schema (a template algorithm with a fixed control flow) that captures the essence of such a strategy. This has been done by Summers (1977), Smith (1985, 1988), Tinkham (1990) in the context of automated program synthesis, and

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

where **R(TX,TY)** denotes a conjunction of recursive atoms, and where "**|**" denotes the *exclusive-or* connective of the schema-language.

Figure 4. A divide-and-conquer logic algorithm schema

by Deville and Burnay (1989) in the context of assisted program construction. Other applications of schemata are programming tutors (Gegg-Harrison, 1989), and program correctness or equivalence checking (see the survey by Manna, 1974).

Our synthesis mechanism is guided by a divide-and-conquer logic algorithm schema (a particular form of design by structural induction).

Loosely speaking, a *divide-and-conquer algorithm* for a 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, if *X* is non-minimal, decompose *X* into a vector ***HX*** of heads of *X* and a vector ***TX*** of tails of *X*, the latter 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 ***HX*** are processed into a series ***HY*** of heads of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***. It may happen that sub-cases emerge with different processing and composition operators: discriminate between them according to the values of ***HX***, ***TX***, and *Y*. It may also happen that the non-minimal case is partitioned into a recursive and a non-recursive case, each of which is partitioned into sub-cases. 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***.

Logic algorithm schemata can be expressed as second-order logic algorithms. For instance, logic algorithms designed by a divide-and-conquer strategy, and having a single minimal case and a single non-minimal case, fit the schema of Figure 4. In the sequel, for simplicity of the presentation, all logic algorithms follow the layout of this schema.

5.3. THE SYNTHESIS MECHANISM

Instantiating some predicate variable(s) of the above divide-and-conquer schema, and introducing trailing atom(s), is a synthesis step. A synthesis mechanism can then be expressed as the following sequence of steps:

- 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 $\textit{SolveNonMin}_k$;
- Step 6: Synthesis of the $\textit{Process}_k$ and the $\textit{Compose}_k$;
- Step 7: Synthesis of the $\textit{Discriminate}_k$;
- Step 8: Syntactic generalization.

A detailed description of all these steps, and the motivation for this particular order of the steps, are beyond the scope of this paper, and can be found in (Flener, 1993). Here we quickly overview the relatively straightforward Steps 1 to 5, and illustrate them on the *firstPlateau*/3 relation, before presenting the truly creative Steps 6 to 8 in greater detail in Section 6 to Section 8, respectively. But first two more design choices:

Types of inference. With specifications by examples (which traditionally give rise to inductive synthesis) and properties (which traditionally give rise to deductive synthesis), it is natural to use both inductive and deductive inference in the synthesis mechanism, whichever is best suited for each step.

Strategy criteria. Our synthesis mechanism is stepwise and non-incremental: examples are presented to it in an all-at-once fashion. It conforms to the strategy described in Section 4.2.

Step 1 yields $LA_1(r)$ by mere syntactic transformation of the example set $\mathcal{E}(r)$ into a logic algorithm. For instance, the definition part of $LA_1(\textit{firstPlateau})$ is $(L=[a] \wedge P=[a] \wedge S=[\,]) \vee (L=[b,b] \wedge P=[b,b] \wedge S=[\,]) \vee \ldots$ Thus, $LA_1(r)$ is totally correct wrt $\mathcal{E}(r)$. Under the hypothesis that $\mathcal{E}(r)$ is consistent with $\mathfrak{R}$, $LA_1(r)$ is even partially correct wrt $\mathfrak{R}$. Moreover, $\Gamma(LA_1(r))$ is $\top_r$, and hence complete wrt $\mathfrak{R}$. Step 1 thus conforms to the strategy of Section 4.2.

Steps 2 and 3 rely on databases of type-specific predicates. For instance, we here assume that Step 2 selects L as induction parameter, and introduces one minimal form $(L=[\_])$, and one non-minimal form $(L=[\_,\_|\_])$, and that Step 3 decomposes the non-minimal form into its head HL and tail TL.

Step 4 syntactically introduces recursive atoms, as dictated by the selected decomposition operator. It performs deductive inference from the specification for computing the witnesses of the trailing atoms. For instance, in example E_6, given the tail $[j,k]$ of L, property P_3 allows the inference of $\textit{firstPlateau}([j,k],[j],[k])$, i.e. $TP=[j] \wedge TS=[k]$. Note that recursion is detected to be useless for some examples.

Step 5 uses particular cases of the methods of Steps 6 and 7 in order to solve the minimal case and the non-recursive, non-minimal case. Note that the latter can have different sub-cases.

Figure 5 shows $LA_5(\textit{firstPlateau})$, where each disjunct has as annotation the set of examples it "covers". The atoms in boldface represent $\Gamma(LA_5(\textit{firstPlateau}))$.

Step 6 (see Section 6) performs inductive inference, and is based on the most-specific-generalization concept. Step 7 (see Section 7) performs deductive inference, and takes a proofs-as-programs approach.

Steps 2, 3, 5, and 6 are non-deterministic: different logic algorithms can be synthesized. Steps 2 to 7 fit the hypotheses of Theorem 4.1. It can be shown that, for $i \in \{2,\ldots,7\}$, $LA_i(r)$ {«,≱,≅} $LA_{i-1}(r)$, and that $\Gamma(LA_i(r))$ {«,≱,<} $\Gamma(LA_{i-1}(r))$.

```
firstPlateau(L,P,S) ⇔
    L=[_]       ∧ P=L ∧ S=[] ∧ L=[_]
                ∧     L=[a] ∧ P=[a] ∧ S=[]                {E1}
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
                ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                    ∧ HL=c ∧ TL=[d]                       {E3}
                  ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                    ∧ HL=e ∧ TL=[f,g]                     {E4}
                  ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                    ∧ HL=h ∧ TL=[i,i]                     {E5}
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ firstPlateau(TL,TP,TS)
                ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                    ∧ HL=b ∧ TL=[b]
                    ∧ TP=[b] ∧ TS=[]                      {E2}
                  ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                    ∧ HL=j ∧ TL=[j,k]
                    ∧ TP=[j] ∧ TS=[k]                     {E6}
                  ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                    ∧ HL=m ∧ TL=[m,m]
                    ∧ TP=[m,m] ∧ TS=[]                    {E7}
```

Figure 5. Assumed version of *LA_5(firstPlateau)*

Step 8 (see Section 8) yields $LA_8(r)$ by application of the Γ operator to $LA_7(r)$. Indeed, all the predicate variables of the divide-and-conquer schema have been instantiated, so synthesis may stop there. But this transformation doesn't entirely satisfy the requirements for Step *f* in the strategy of Section 4.2, because $LA_8(r)$ can only be guaranteed to be complete, but not necessarily totally correct, wrt $\mathfrak{R}$. Such potential over-generalization is inherent to synthesis from incomplete specifications. We have that $LA_8(r)$ {»,≉,>} $LA_7(r)$, and that $\Gamma(LA_8(r)) = \Gamma(LA_7(r))$.

6. Synthesis of the *Process$_k$* and *Compose$_k$* (Step 6)

The *Process$_k$*(***HX***,***HY***) procedure transforms, in the k^{th} sub-case of the recursive case, the heads ***HX*** of the induction parameter *X* into heads ***HY*** of the other parameter *Y*. The *Compose$_k$*(***HY***,***TY***,*Y*) procedure computes, in the k^{th} sub-case of the recursive case, parameter *Y* from its heads ***HY*** (obtained by processing ***HX***) and tails ***TY*** (obtained by recursion on ***TX***). We formally present the objective and methods of Step 6 (Section 6.1), and illustrate them on the *firstPlateau*/3 relation (Section 6.2). Other examples can be found in (Flener and Deville, 1993).

$$
\begin{array}{llll}
\mathtt{r(X,Y)} \Leftrightarrow & & & \\
 & \mathtt{minimal(X)} & \wedge & \mathtt{solve(X,Y)} \\
 & & \wedge & \quad \vee_{j \in I} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
\vee \vee_{1 \le k \le v} & \mathtt{nonMinimal(X)} & \wedge & \mathtt{decompose(X,\mathbf{HX},\mathbf{TX})} \\
 & & \wedge & \mathtt{solveNonMin_k(\mathbf{HX},\mathbf{TX},Y)} \\
 & & \wedge & \quad \vee_{j \in \mathcal{K}_k} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{HX=hx_j} \wedge \mathbf{TX=tx_j} \\
\vee & \mathtt{nonMinimal(X)} & \wedge & \mathtt{decompose(X,\mathbf{HX},\mathbf{TX})} \\
 & & \wedge & \mathbf{r(TX,TY)} \\
 & & \wedge & \quad \vee_{j \in \mathcal{L}} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{HX=hx_j} \wedge \mathbf{TX=tx_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{TY} \in \mathbf{ty_j}
\end{array}
$$

where I is the set of indices of the minimal examples, $\mathcal{K}$ is the set of indices of the non-recursive examples, and $\mathcal{L}$ is the set of indices of the recursive examples, such that I, $\mathcal{K}$, $\mathcal{L}$ form a partition of $\{1,\ldots,m\}$. The $\mathcal{K}_k$ form a partition of $\mathcal{K}$. The $\boldsymbol{ty}_j$ are vectors of sets ty_{jh} of witnesses such that $r(tx_{jh},ty_{jhi})$ holds for some element ty_{jhi} of ty_{jh} $(j \in \mathcal{L})$.

Figure 6. $LA_5(r)$

$$
\begin{array}{llll}
\mathtt{r(X,Y)} \Leftrightarrow & & & \\
 & \mathtt{minimal(X)} & \wedge & \mathtt{solve(X,Y)} \\
 & & \wedge & \quad \vee_{j \in I} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
\vee \vee_{1 \le k \le v} & \mathtt{nonMinimal(X)} & \wedge & \mathtt{decompose(X,\mathbf{HX},\mathbf{TX})} \\
 & & \wedge & \mathtt{solveNonMin_k(\mathbf{HX},\mathbf{TX},Y)} \\
 & & \wedge & \quad \vee_{j \in \mathcal{K}_k} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{HX=hx_j} \wedge \mathbf{TX=tx_j} \\
\vee \vee_{c-w<k \le c} & \mathtt{nonMinimal(X)} & \wedge & \mathtt{decompose(X,\mathbf{HX},\mathbf{TX})} \\
 & & \wedge & \mathbf{r(TX,TY)} \\
 & & \wedge & \mathtt{procComp_k(\mathbf{HX},\mathbf{TY},Y)} \\
 & & \wedge & \quad \vee_{j \in \mathcal{L}_k} \quad \mathtt{X=x_j} \wedge \mathtt{Y=y_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{HX=hx_j} \wedge \mathbf{TX=tx_j} \\
 & & & \qquad\qquad \wedge\ \mathbf{TY} \in \mathbf{ty'_j}
\end{array}
$$

where the elements ty'_{jh} of the $\boldsymbol{ty'}_j$ are subsets of the sets of witnesses ty_{jh} generated at Step 4 (Introduction of the recursive atoms). The $\mathcal{L}_k$ form a partition of $\mathcal{L}$.

Figure 7. $LA_6(r)$

6.1. FORMALIZATION: OBJECTIVE AND METHODS

Given $LA_5(r)$ as shown in Figure 6, the aim at Step 6 is to transform $LA_5(r)$ into $LA_6(r)$ that fits the schema of Figure 7. A total of m disjuncts (m being the number of examples) can be identified by expanding the bodies of $LA_5(r)$ and $LA_6(r)$ into disjunctive normal form. The trailing atoms of each disjunct are offset by additional tabulations.

We decide to merge each $Process_k(\boldsymbol{HX},\boldsymbol{HY})$ with its $Compose_k(\boldsymbol{HY},\boldsymbol{TY},Y)$ into a $ProcComp_k(\boldsymbol{HX},\boldsymbol{TY},Y)$, so that their instances are synthesized at the same time.

We have identified two methods to synthesize instances of the $ProcComp_k$:

- computation of most-specific-generalizations (msg): the MSG Method applies if each $ProcComp_k$ is implemented as a conjunction of equality atoms;
- synthesis from an inferred specification by examples and properties: the Synthesis Method applies if some $ProcComp_k$ needs a full-fledged recursive algorithm, i.e. is implemented as a disjunction of conjunctions of any literals.

We now discuss these methods in turn.

6.1.1. THE MSG METHOD

The MSG Method partitions the recursive disjuncts of $LA_5(r)$ incrementally and non-deterministically. The concept of most-specific-generalization (msg) was introduced simultaneously, but independently, by Plotkin (1970) and Reynolds (1970).

DEFINITION 6.1. Term s is *less general* than term t (denoted $s \leq t$) iff there is a substitution σ such that $s = t\sigma$.

The relation $\leq/2$ forms a complete lattice on the term set $\mathcal{U}$ (modulo variable renaming) to which a least element has been added (Lassez *et al.*, 1987). The glb operator computes the greatest common instance of two terms (by a unification algorithm, yielding their mgu); the lub operator computes the msg of two terms (by an anti-unification algorithm). The msg of two terms thus always exists, and is unique up to variable renaming.

EXAMPLE 6.1. The msg of terms $f(a,b,X,Y,X)$ and $f(a,c,d,Z,d)$ is $f(a,L,M,N,M)$.

Intuitively, the MSG Method collects into a subset the recursive disjuncts of $LA_5(r)$ in which Y is constructed in a uniform way, by unification only, from $\boldsymbol{HX}$ and $\boldsymbol{TY}$.

Note that, by the definition of examples, every y_j is ground, and that, by construction (Step 3 only uses decomposition predicates that are deterministic given a ground value of the induction parameter), all the $\boldsymbol{hx}_j$ and $\boldsymbol{tx}_j$ are ground. However, if r is non-deterministic given a ground value of the induction parameter, then there may be several values for the $\boldsymbol{TY}$. Hence, Step 4 yields, in all generality, sets of terms as $\boldsymbol{ty}_j$.

DEFINITION 6.2. A *determinate* disjunct has a unique, ground value in its trailing atoms for the $\boldsymbol{TY}$. Otherwise, it is an *indeterminate* disjunct.

Here, we only present the situation where all the recursive disjuncts are determinate, and indicate how to extend it to the indeterminate situation. We now define a criterion for verifying whether several disjuncts construct Y in the same way from $\boldsymbol{HX}$ and $\boldsymbol{TY}$.

DEFINITION 6.3. The *msg* of a set of determinate disjuncts is the msg of the $<\boldsymbol{hx}_j,\boldsymbol{ty}_j,y_j>$ value-tuples extracted from these disjuncts.

DEFINITION 6.4. A set of determinate disjuncts $\mathcal{D}$ is *compatible* iff the msg $<\boldsymbol{hx},\boldsymbol{ty},y>$ of $\mathcal{D}$ is such that $leaves(\boldsymbol{ty}) \subseteq leaves(y)$, where $leaves(t)$ denotes the set of constants and variables occurring in term t.

A compatible set $\mathcal{D}$ of disjuncts is of great interest, because each of its disjuncts constructs Y from $\boldsymbol{HX}$ and $\boldsymbol{TY}$ in the same way as its msg does. If $<\boldsymbol{hx},\boldsymbol{ty},y>$ is the msg of $\mathcal{D}$, then all the values of Y in the different disjuncts of $\mathcal{D}$ can be computed by the same conjunction:

$$\boldsymbol{HX}=\boldsymbol{hx} \wedge \boldsymbol{TY}=\boldsymbol{ty} \wedge Y=y.$$

But we are of course not interested in inferring instances of the $ProcComp_k$ that cover "too many", if not all, examples beyond the given ones. Compatibility is thus meant to restrict the covered examples. Nor are we interested in inferring instances of the $ProcComp_k$ that cover "too little", if not only one, example(s). The idea is thus to partition the set of recursive disjuncts of $LA_5(r)$ into a (*i*) minimal number of (*ii*) compatible subsets.

ALGORITHM. Initially, there are no subsets. At any moment, the recursive disjuncts can be classified according to whether or not they belong to some subset. Progression is achieved by selecting a disjunct D_j that doesn't belong to any subset. If there is some subset $\mathcal{D}$ such that $\mathcal{D} \cup \{D_j\}$ is compatible, then $\mathcal{D}$ becomes $\mathcal{D} \cup \{D_j\}$. Otherwise a new singleton subset $\{D_j\}$ is created.

In the indeterminate situation, either the non-empty sets of values for the $\boldsymbol{TY}$ in some disjunct are ground, and choice-points will appear in the algorithm for rendering that disjunct determinate, or some variables appear in these sets of values for the $\boldsymbol{TY}$, and choice-points will appear in the search for grounding substitutions.

Let $procComp_k$ be instantiations of $ProcComp_k$. The msg $<\boldsymbol{hx}_k,\boldsymbol{ty}_k,y_k>$ of a subset is rewritten as follows:

$$procComp_k(\boldsymbol{HX},\boldsymbol{TY},Y) \Leftrightarrow \boldsymbol{HX}=\boldsymbol{hx}_k \wedge \boldsymbol{TY}=\boldsymbol{ty}_k \wedge Y=y_k$$

so that it can be inserted into the corresponding disjuncts of $LA_5(r)$.

One can show that the MSG Method yields instances of the $ProcComp_k$ that are as general as possible (Flener, 1993). The MSG Method obviously is non-deterministic (choice of unclassified disjunct, choice of ground values of the $\boldsymbol{TY}$, respectively choice of grounding substitutions). Its time complexity is $\Omega(e^m)$. This exponential complexity is not a drawback, as the number of examples m usually is quite small.

6.1.2. THE SYNTHESIS METHOD

The Synthesis Method assumes that there is exactly one subset (w=1), and that $ProcComp_1(\boldsymbol{HX},\boldsymbol{TY},Y)$ is implemented as a disjunction of conjunctions of any literals, i.e. needs to be synthesized from scratch, just like any other logic algorithm. Let *procComp* be the chosen instantiation of $ProcComp_1$. A specification by examples and properties for $procComp(\boldsymbol{HX},\boldsymbol{TY},Y)$ has to be inferred from $LA_5(r)$.

The inference of an example set is easy: extract the $<\boldsymbol{hx}_j,\boldsymbol{ty}_j,y_j>$ tuples from the recursive disjuncts of $LA_5(r)$. If there are several possible values for $\boldsymbol{ty}_j$, then extract the value that was successfully used by the MSG Method: compatibility with other disjuncts is indeed a strong argument that this is a "good" choice.

The inference of a property set is based on the observation that the properties of r are "inherited" by *procComp*. For every property:

$$r(x_j,y_j) \Leftarrow B_j \qquad (P_j)$$

find variants of examples or body-less properties:

$$r(tx_{ji},ty_{ji}) \qquad (E_i \text{ or } P_i)$$

such that:

$$nonMinimal(x_j) \wedge decompose(x_j,\mathbf{hx}_j,\mathbf{tx}_j) \qquad (*)$$

where *nonMinimal* is the *NonMinimal* predicate selected at Step 2, and *decompose* is the *Decompose* predicate selected at Step 3. Then infer:

$$procComp(\mathbf{hx}_j,\mathbf{ty}_j,y_j) \Leftarrow B_j \qquad (P_j')$$

as a property of *procComp*.

This works, because when unfolding, in P_j, the $r(x_j,y_j)$ atom using $LA_5(r)$ (where the *procComp* has already been added), and then simplifying the conclusion using E_i or P_i, (*), and the fact that, by Step 3, *decompose* is deterministic given a ground value of the induction parameter, we effectively obtain P_j'.

A logic algorithm *LA(procComp)* can now be synthesized from this inferred specification by examples and properties. The synthesis of *LA(r)* proceeds using *procComp*, assuming this predicate to be a primitive.

6.2. ILLUSTRATION: THE *firstPlateau*/3 RELATION

At Step 4 (see Figure 5), the actual values of the introduced parameters *TP* and *TS* are uniquely determined, so the recursive disjuncts are all determinate. The phrase “disjunct D_j” now stands for the disjunct of $LA_6(firstPlateau)$ that covers example E_j.

- According to the MSG Method, there are initially no subsets. We first consider disjunct D_2, and create a singleton subset $\{D_2\}$.
- We pursue with D_6. To see whether D_6 is compatible with subset $\{D_2\}$, we compute the msg of their <*HL*,*TP*,*TS*,*P*,*S*> value-tuples:

HL	*TP*	*TS*	*P*	*S*	
b	[*b*]	[]	[*b*,*b*]	[]	$msg\{D_2\}$
j	[*j*]	[*k*]	[*j*,*j*]	[*k*]	D_6
A	[*A*]	*U*	[*A*,*A*]	*U*	$msg\{D_2,D_6\}$

  Disjunct D_6 is compatible with subset $\{D_2\}$, because [*A*,*A*] and *U* are constructed in terms of *A*, *nil*, and *U*.
- We pursue with D_7. To see whether D_7 is compatible with subset $\{D_2,D_6\}$, we compute the msg of their <*HL*,*TP*,*TS*,*P*,*S*> value-tuples:

HL	*TP*	*TS*	*P*	*S*	
A	[*A*]	*U*	[*A*,*A*]	*U*	$msg\{D_2,D_6\}$
m	[*m*,*m*]	[]	[*m*,*m*,*m*]	[]	D_7
A	[*A* | *T*]	*U*	[*A*,*A* | *T*]	*U*	$msg\{D_2,D_6,D_7\}$

```
firstPlateau(L,P,S) ⇔
    L=[_]       ∧ P=L ∧ S=[] ∧ L=[_]
                ∧     L=[a] ∧ P=[a] ∧ S=[]                {E1}
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
                ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                    ∧ HL=c ∧ TL=[d]                       {E3}
                  ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                    ∧ HL=e ∧ TL=[f,g]                     {E4}
                  ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                    ∧ HL=h ∧ TL=[i,i]                     {E5}
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ firstPlateau(TL,TP,TS)
                ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
                ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                    ∧ HL=b ∧ TL=[b]
                    ∧ TP=[b] ∧ TS=[]                      {E2}
                  ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                    ∧ HL=j ∧ TL=[j,k]
                    ∧ TP=[j] ∧ TS=[k]                     {E6}
                  ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                    ∧ HL=m ∧ TL=[m,m]
                    ∧ TP=[m,m] ∧ TS=[]                    {E7}
```

Figure 8. *LA_6(firstPlateau)*

Disjunct D_7 is compatible with subset $\{D_2,D_6\}$, because $[A,A \mid T]$ and U are constructed in terms of A, T, and U.

There are no other disjuncts. We have partitioned the recursive disjuncts into one subset, namely $\{D_2,D_6,D_7\}$.

Let *procComp* be the chosen instantiation of *ProcComp*. It is implemented by re-expression, and subsequent simplification, of the msg:

$$procComp(HL,TP,TS,P,S) \Leftrightarrow P=[HL \mid TP] \wedge S=TS \wedge TP=[HL \mid \_]$$

This result is inserted into the corresponding disjuncts of *LA_5(firstPlateau)*, and *LA_6(firstPlateau)* thus looks as depicted in Figure 8.

7. Synthesis of the *Discriminate$_k$* (Step 7)

The *Discriminate$_k$*(***HX***,***TX***,*Y*) procedure performs, in the k^{th} sub-case of the non-minimal case, some tests so as to ensure that the parameters effectively belong to that sub-case. We first formally present the objective and methods of Step 7 (Section 7.1), and then illustrate them on the *firstPlateau*/3 relation (Section 7.2). Other examples can be found in (Flener and Deville, 1993).

```
r(X,Y) ⇔
                 minimal(X)     ∧ solve(X,Y)
                                ∧      ∨_{j∈I}   X=x_j ∧ Y=y_j
  ∨ ∨_{1≤k≤v}    nonMinimal(X)  ∧ decompose(X,HX,TX)
                                ∧ discriminate_k(HX,TX,Y)
                                ∧ solveNonMin_k(HX,TX,Y)
                                ∧      ∨_{j∈K_k} X=x_j ∧ Y=y_j
                                               ∧ HX=hx_j ∧ TX=tx_j
  ∨ ∨_{c-w<k≤c}  nonMinimal(X)  ∧ decompose(X,HX,TX)
                                ∧ discriminate_k(HX,TX,Y)
                                ∧ r(TX,TY)
                                ∧ procComp_k(HX,TY,Y)
                                ∧      ∨_{j∈L_k} X=x_j ∧ Y=y_j
                                               ∧ HX=hx_j ∧ TX=tx_j
                                               ∧ TY∈ty'_j
```

Figure 9. *$LA_7(r)$*

7.1. FORMALIZATION: OBJECTIVE AND METHODS

Given *$LA_6(r)$* as shown in Figure 7, the aim at Step 7 is to transform *$LA_6(r)$* into *$LA_7(r)$* that fits the schema of Figure 9. (Note that Step 7 is totally independent of the choice of Step 6 to merge the *$Process_k$* and *$Compose_k$* predicates.) This objective is achieved by two consecutive tasks:

- synthesis of specialized instantiations of the *$Discriminate_k$*;
- generalization of these specialized instantiations of the *$Discriminate_k$*.

These tasks are performed by a Proofs-as-Programs Method and a Generalization Method, respectively. We now discuss these methods in turn.

7.1.1. THE PROOFS-AS-PROGRAMS METHOD

The Proofs-as-Programs Method extracts specialized instantiations of the *$Discriminate_k$* from the proofs that the logic algorithm $\Gamma(LA_6(r))$ is complete wrt the properties. Indeed, the properties contain explicit information that has not yet been synthesized into any disjuncts.

Intuitively, the Proofs-as-Programs Method takes a property *$r(X,Y) \Leftarrow Body_i$*, and unfolds its *$r(X,Y)$* atom using a disjunct of $\Gamma(LA_6(r))$, to which a *$discriminate_k(\boldsymbol{HX},\boldsymbol{TX},Y)$* atom has been added. By resolution within the left-hand side, this eventually simplifies to *$discriminate_k(\boldsymbol{HX},\boldsymbol{TX},Y) \Leftarrow Body_i\sigma$*, where σ is an answer substitution. Each *$discriminate_k$* is defined by all the program clauses obtained by doing this for all properties and all disjuncts.

More formally, when considering property P_i, let $\mathcal{T}_i$ be a theory composed of:

- the generalized logic algorithm $\Gamma(LA_6(r))$;
- the specification $\mathcal{EP}(r) \setminus \{P_i\}$;
- logic algorithms for all primitive predicates.

The Proofs-as-Programs Method attempts to prove that each property P_i is a logical consequence of its theory $\mathcal{T}_i$. These proofs are done by an extension to SLD resolution (we use the terminology and notation of Lloyd, 1987). The extension is a subset of Kanamori and Seki's extended execution mechanism (1986), in the sense that goals are here *implicative goals* (statements of the form $\forall \mathbf{X} \exists \mathbf{Y}\ G^+(\mathbf{X},\mathbf{Y}) \leftarrow G^-(\mathbf{X})$, where conclusion G^+ and hypothesis G^- are conjunctions of atoms). In the sequel, "goal" stands for "implicative goal".

A definite program version of $\mathcal{T}_i$ has to be generated: this is straightforward due to the chosen formalisms for logic algorithms, examples, and properties.

The *initial goal* is property P_i, rewritten as an implicative goal.

DEFINITION 7.1. The rule of *definite clause inference* (denoted *DCI*) is a natural extension of SLD resolution to implicative goals. Given a goal G, the selected atom is chosen within G^+, and the mgu may only bind existential variables of G^+.

SLD resolution is parameterized on a computation rule and a search rule. These are, for the purpose of the Proofs-as-Programs Method, instantiated as follows:

- the *computation rule* satisfies the following condition: never select an atom with predicate r if there still are atoms with primitive predicates. Indeed, while theoretically not required, the delaying of the selection of recursive atoms generally results in less search;
- the *search rule* is as follows:
  - an atom with a primitive predicate is resolved according to its semantics;
  - the atom with predicate r in (the conclusion of) the root of the proof tree is resolved using the program clauses generated from $\Gamma(LA_6(r))$;
  - an atom with predicate r in (the conclusion of) a non-root node of the proof tree is resolved using the clauses generated from $\mathcal{T}_i \setminus \{\Gamma(LA_6(r))\}$.

Note that the search rule is context-dependent. For the resolution of the root, we use $\Gamma(LA_6(r))$ rather than the examples or properties, because that wouldn't make sense: we are trying to prove $\Gamma(LA_6(r))$ complete wrt $\mathcal{P}(r)$, but not to prove the specification internally consistent. For the resolution of atoms with predicate r in a non-root node, we use the examples and other properties rather than $\Gamma(LA_6(r))$ because the latter is in general not correct wrt $\mathcal{P}(r)$.

This can be easily extended to handle negated primitive predicates since they are resolved according to their semantics.

DEFINITION 7.2. The rule of *negation-as-failure inference* (denoted *NFI*) is a natural extension of the NAF rule (Clark, 1978) to implicative goals. Given a goal G, the selected atom A is chosen within G^-, and a conjunction of d resolvent goals is generated, namely $G\sigma_i$, where $A\sigma_i$ has been replaced by the conjunction $B_i\sigma_i$, with $H_i \leftarrow B_i$ being one of the d definite clauses whose head H_i unifies with A under mgu σ_i. All new variables introduced in the resolvent goals are free variables.

DEFINITION 7.3. Given a goal G, the rule of *simplification* (denoted *Sim*) selects two atoms A and B, in G^+ and G^- respectively, that unify with an mgu σ that only binds existential variables of G^+. The resolvent goal is obtained from $G\sigma$ by deleting A and B.

This subset of extended execution is sound wrt the Clark completion semantics (Kanamori and Seki, 1986, page 487).

DEFINITION 7.4. A derivation via the DCI, NFI, and Sim rules *succeeds* iff it ends in a goal whose conclusion is empty.

We partially define $discriminate_k$ by the program clause:

$$discriminate_k(\boldsymbol{HX},\boldsymbol{TX},y)\ \sigma \leftarrow Hyp$$

where:

- k is the number of the clause from $\Gamma(LA_6(r))$ that is used in the first DCI step;
- y is the value of parameter Y in the head of property P_i;
- σ is the computed answer substitution;
- Hyp is the hypothesis of the last goal of the derivation.

DEFINITION 7.5. A derivation via the DCI, NFI, and Sim rules *fails* iff it doesn't succeed.

Failure is detectable only in specific settings. For instance, no infinite derivation can occur if all primitive predicates have finite proofs for all directionalities.

After the computation of all successful derivations for all properties, the revealed procedures of the discriminants are translated into logic algorithms. Since the latter are non-recursive by construction, they are then inserted into the corresponding disjuncts of $LA_6(r)$, so as to yield $LA_7(r)$.

THEOREM 7.1. *$\Gamma(LA_7(r))$ is complete wrt P(r).*

PROOF. Let $\mathcal{T}_i$' be defined like $\mathcal{T}_i$, using $LA_7(r)$ rather than $LA_6(r)$. Let's prove that each property P_i is a logical consequence of its theory $\mathcal{T}_i$'. Without loss of generality, we can assume that the previous derivations of $\mathcal{T}_i \vdash P_i$ are prefixes of the new derivations of $\mathcal{T}_i' \vdash P_i$ (namely by a relaxation of the recommendation above for the computation rule): each new derivation thus eventually yields a goal whose conclusion only involves the discriminant atoms. By construction, these atoms are identical to the atoms in the hypothesis: by repeated application of the Sim rule, that goal can be simplified into the empty goal, i.e. the new derivation succeeds as well. By soundness of extended execution (Kanamori and Seki, 1986), the property set $\mathcal{P}(r)$ is thus a logical consequence of $\Gamma(LA_7(r))$, or, in other words, by Definition 2.6, $\Gamma(LA_7(r))$ is complete wrt $\mathcal{P}(r)$. ❑

Note that this is unlike classical program extraction from proofs, since the program is here extracted from the unique final results of several proofs, rather than on-the-fly (or a posteriori) from multiple steps of a single proof.

Also note that the Proofs-as-Programs Method is deterministic and idempotent. Assuming that all primitives used in $\mathcal{P}(r)$ are deterministic, its space complexity is $O(pc(m+p)^t)$, where c is the number of discriminants, m is the number of examples, p is the number of properties, and t is the number of tails of the induction parameter. Indeed, there are p proofs to be made, and each proof-tree has size $O(c(m+p)^t)$ because it has only two choice-points, namely the resolution of the root, where there are c possibilities, and the resolution of the recursive atoms, where there are $m+p-1$ possibilities, all other proof steps being deterministic. This assumes that there is a fixed maximum number of atoms for the definitions of the used primitives. Indeed, if that number is a function of c, m, p, or t, then this complexity analysis doesn't hold.

7.1.2. THE GENERALIZATION METHOD

Since properties only embody incomplete information, the obtained discriminants are too specific. The Generalization Method applies generalization heuristics to the discriminants, and substitutes the results for the original ones. If the modification indeed amounts to a generalization, then Theorem 7.1 still holds.

Here are two valuable generalization heuristics:

HEURISTIC 1. If parameter *Y* is not an auxiliary parameter (a parameter that doesn't change through recursion), then it is irrelevant for discrimination, and can be deleted from the discriminants by projection.

HEURISTIC 2. The parameters ***TX*** and *Y* (the latter only if it is an auxiliary parameter) should range across their entire domains: if necessary, some of their values should be generalized.

The application of these heuristics should be interactive, with the specifier.

7.2. ILLUSTRATION: THE *firstPlateau*/3 RELATION

The Proofs-as-Programs Method attempts to prove, by extended execution, that each property P_i is a logical consequence of its theory $\mathcal{T}_i$. Note that $\Gamma(LA_6(firstPlateau))$ has three disjuncts, whose corresponding definite clauses are named C_1 to C_3 in the sequel. Only the non-minimal clauses C_2 and C_3 need discriminants. The latter thus read *discriminate*$_k$*(HL,TL,P,S)*, for $k = 2, 3$. In a goal, the selected atom(s) for the next application(s) of the DCI rule is (are) written in boldface. For syntactic convenience, we write goals in quantifier-free form, prefixing existential variables by "?". Derivations starting from C_1 are useless since no discriminant is needed for the minimal clause.

- We start with P_1. The derivations starting from C_2 and C_3 lead to failure.
- We pursue with P_2. Clause C_2 leads to failure. Then, starting from C_3:

```
              firstPlateau([X,Y],[X,Y],[]) ← X=Y
                     DCI: C3       ↓       {}
         [X,Y]=[?_,?_|?_] & [X,Y]=[?HL|?TL] &
                 firstPlateau(?TL,?TP,?TS) &
   [X,Y]=[?HL|?TP] & []=?TS & ?TP=[?HL|?_]  ← X=Y
              4 × DCI: LA(=)       ↓       {HL/X, TL/[Y], TP[Y], TS/[]}
      firstPlateau([Y],[Y],[]) & [Y]=[X|?_]  ← X=Y
                 NFI: LA(=)        ↓       {Y/X}
        firstPlateau([X],[X],[]) & [X]=[X|?_]  ←
                 DCI: LA(=)        ↓       {}
                  firstPlateau([X],[X],[]) ←
                     DCI: P1       ↓       {}
                                   ❑
```

A specialized discriminant for C_3 is:

$$discriminate_3(X,[X],[X,X],[]).$$

- We pursue with P_3. Clause C_3 leads to failure. Then, starting from C_2:

```
firstPlateau([X,Y],[X],[Y]) ← X≠Y
          DCI: C2      ↓      {}
  [X,Y]=[?_,?_|?_] & [X,Y]=[?HL|?TL] &
[X]=[?HL] & [Y]=?TL & ?TL=[?_|?_] ← X≠Y
    5 × DCI: LA(=)     ↓      {HL/X, TL/[Y]}
                  ← X≠Y
```

A specialized discriminant for C_2 is:

$$discriminate_2(X,[Y],[X],[Y]) \leftarrow X \neq Y$$

There is no other property. There are no other derivations. The discriminants are rewritten as logic algorithms.

The Generalization Method applies Heuristic 1 to decide that the third and fourth parameters are irrelevant in both discriminants. It applies Heuristic 2 to generalize the second parameter of both discriminants into non-empty lists. After some renaming and rewriting, the discriminants read:

$$discriminate_2(HL,TL) \Leftrightarrow TL=[H\,|\,\_] \wedge HL \neq H$$
$$discriminate_3(HL,TL) \Leftrightarrow TL=[H\,|\,\_] \wedge HL = H$$

They are inserted into *LA_6(firstPlateau)*, and yield *LA_7(firstPlateau)* (see Figure 10).

8. Syntactic Generalization (Step 8)

We first formally present the objective and method of Step 8, and then illustrate them on the *firstPlateau*/3 relation.

Given *$LA_7(r)$* as shown in Figure 9, the aim at Step 8 is to transform *$LA_7(r)$* into *$LA_8(r)$* that fits the schema of Figure 11. All the predicate variables of the divide-and-conquer schema have already been instantiated until Step 7, and we have used all the information contained in the specification:

- examples are injected at Step 1, and are kept along all subsequent steps in the form of trailing atoms, so that they be present when needed;
- examples and properties are used at Step 4 to infer the values of the ***TX***;
- properties are used at Step 6 to infer specifications of sub-problems;
- properties are used at Step 7 to infer discriminants.

So one may consider synthesis finished. We postulate that *$LA_8(r)$* is *$\Gamma(LA_7(r))$*.

For instance, *LA_8(firstPlateau)* looks as depicted in Figure 12. It is totally correct wrt its intended relation, and equivalent to the version in Figure 3.

9. Conclusions

We evaluate the results given here (Section 9.1), discuss the implementation of the synthesis mechanism (Section 9.2), and outline future research (Section 9.3).

```
firstPlateau(L,P,S) ⇔
    L=[_]      ∧ P=L ∧ S=[] ∧ L=[_] ∧ L=[_]
               ∧     L=[a] ∧ P=[a] ∧ S=[]                {E1}
  ∨ L=[_,_|_]  ∧ L=[HL|TL]
               ∧ TL=[H|_] ∧ HL≠H
               ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
               ∧     L=[c,d] ∧ P=[c] ∧ S=[d]
                   ∧ HL=c ∧ TL=[d]                       {E3}
                 ∨   L=[e,f,g] ∧ P=[e] ∧ S=[f,g]
                   ∧ HL=e ∧ TL=[f,g]                     {E4}
                 ∨   L=[h,i,i] ∧ P=[h] ∧ S=[i,i]
                   ∧ HL=h ∧ TL=[i,i]                     {E5}
  ∨ L=[_,_|_]  ∧ L=[HL|TL]
               ∧ TL=[H|_] ∧ HL=H
               ∧ firstPlateau(TL,TP,TS)
               ∧ P=[HL|TP]  ∧ S=TS  ∧ TP=[HL|_]
               ∧     L=[b,b] ∧ P=[b,b] ∧ S=[]
                   ∧ HL=b ∧ TL=[b]
                   ∧ TP=[b] ∧ TS=[]                      {E2}
                 ∨   L=[j,j,k] ∧ P=[j,j] ∧ S=[k]
                   ∧ HL=j ∧ TL=[j,k]
                   ∧ TP=[j] ∧ TS=[k]                     {E6}
                 ∨   L=[m,m,m] ∧ P=[m,m,m] ∧ S=[]
                   ∧ HL=m ∧ TL=[m,m]
                   ∧ TP=[m,m] ∧ TS=[]                    {E7}
```

Figure 10. *LA$_7$(firstPlateau)*

```
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)
                                  ∧ procCompk(HX,TY,Y)     )
```

Figure 11. *LA$_8$(r)*

9.1. EVALUATION

After defining logic formalisms for incomplete specifications and logic algorithms, we defined logic algorithm correctness and comparison criteria. Then we proposed criteria for upward and downward progression, in order to state strategies for incremental and non-incremental stepwise synthesis. We showed how these strategies can be refined in

```
firstPlateau(L,P,S) ⇔
    L=[_]       ∧ P=L ∧ S=[] ∧ L=[_] ∧ L=[_]
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ TL=[H|_] ∧ HL≠H
                ∧ P=[HL] ∧ S=TL ∧ TL=[_|_]
  ∨ L=[_,_|_]   ∧ L=[HL|TL]
                ∧ TL=[H|_] ∧ HL=H
                ∧ firstPlateau(TL,TP,TS)
                ∧ P=[HL|TP] ∧ S=TS ∧ TP=[HL|_]
```

Figure 12. *LA_8(firstPlateau)*

order to be practical. Finally, we presented a particular synthesis mechanism that is non-incremental, both deductive and inductive, interactive, and schema-guided.

The main originality of this research is the development of a general framework of stepwise synthesis from incomplete specifications, and its particularization to a synthesis mechanism that is being implemented.

We have restricted the presentation of the synthesis mechanism so as to keep it simple. The actual system is based on a generalized divide-and-conquer schema that handles *n*-ary relations and optimizes the handling of auxiliary parameters.

It is important to understand that the MSG Method (as an inductive technique that is based on examples) and the Proofs-as-Programs Method (as a deductive technique that is based on properties) are not at all tied to Steps 6 and 7, respectively. They are actually often interchangeably applicable, whatever the underlying schema.

Among all the related research cited so far, the works of Drabent *et al*. (1988) and De Raedt and Bruynooghe (1992) come closest to ours in that they also start from examples and something similar to our properties. The main differences are that their systems perform incremental synthesis, and that they use their properties only for "bug-detection" purposes, but not in a constructive fashion.

9.2. THE SYNAPSE IMPLEMENTATION

A prototype of our synthesis mechanism is being implemented (in Quintus Prolog) as the SYNAPSE system (*SYNthesis of Algorithms from PropertieS and Examples*).

The system is modular in that implementations of methods can easily be added, deleted, or modified, and that it can be customized by extending the internal databases of available primitives for specifications, and of type-specific predicates, for Steps 2 and 3.

Given a specification *$\mathcal{EP}$(r)*, SYNAPSE prints out candidate versions of the logic algorithm *LA(r)*, and optionally the intermediate versions *LA_i(r)*, as well as questions to the specifier. Hints about what induction parameter or decomposition predicate to select are accepted. A straightforward naming schema is used to name new variables, so that it is easy to read synthesized logic algorithms.

Our experience with SYNAPSE shows that specifying relations by examples and properties is a viable approach. Moreover, in view of optimizing synthesis, we plan to develop a methodology of choosing “good” examples and properties, using our knowledge of the actual synthesis mechanism.

SYNAPSE seems to be quite efficient, proving thus the adequacy of properties for disambiguating situations where examples alone lack in expressive power. Exponential search is reduced as much as possible by interaction with the user.

9.3. FUTURE WORK

The schema of Figure 4 covers a wide range of divide-and-conquer algorithms. However, it only allows non-compound induction parameters that furthermore should only lead to two cases, a minimal and a non-minimal one. Other possible extensions are mutually recursive logic algorithms.

The divide-and-conquer schema is hard-wired into the synthesis mechanism: the support of alternative schemata is envisaged. An extension of the mechanism would be parameterized on schemata. Most of the methods used in the current mechanism are suitable whatever the underlying schema.

As is stands, the mechanism does not need negative examples. Handling these would require a total overhaul of the mechanism, but is an interesting alley for further research aiming at a more effective control of over-generalization.

Acknowledgments

The authors gratefully acknowledge many insightful discussions with Prof. Baudouin Le Charlier (Université de Namur, Belgium). Many thanks also to the anonymous reviewers for their perceptive comments on an earlier version of this paper. Parts of the results presented here were obtained while the authors were involved in the *Folon* research project (supported by the Belgian National Incentive Program for Fundamental Research in Artificial Intelligence), and while the first author was on leave at Duke University (NC, USA). The first author is supported by the Ministry of Scientific Research and Cultural Affairs of the Grand-Duchy of Luxembourg.

References

Biermann, A.W. (1984). Dealing with search. In: Biermann, A.W., Guiho, G., Kodratoff, Y. (eds) *Automatic Program Construction Techniques*. Macmillan, 1984, pp 375-392.

Biermann, A.W. (1992). Automatic Programming. In: Shapiro, S.C. (ed) *Encyclopedia of Artificial Intelligence*. Second, extended edition. John Wiley & Sons, 1992, pp 59-83.

Biermann, A.W., Smith, D.R. (1979). A production rule mechanism for generating LISP code. *IEEE Trans. on Systems, Man, and Cybernetics* (9)5:260-276, May 1979.

Bundy, A., Smaill, A., Wiggins, G. (1990). The synthesis of logic programs from inductive proofs. In: Lloyd, J.W. (ed) *Computational Logic*. Springer-Verlag, 1990, pp 135-149.

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

Clark, K.L. (1981). *The synthesis and verification of logic programs*. Research Report DOC 81/36, Imperial College, London (UK), 1981.

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

Deville, Y. (1990). *Logic Programming: Systematic Program Development*. International Series in Logic Programming, Addison Wesley, 1990.

Deville, Y., Burnay, J. (1989). Generalization and program schemata. In: Lusk, E.L., Overbeek, R.A. (eds) *Proc. of NACLP'*89, MIT Press, 1989, pp 409-425.

Deville, Y., Flener, P. (1993). Correctness criteria for logic program synthesis. Research Report, Université Catholique de Louvain, Unité d'Informatique, 1993 (in preparation).

Drabent, W., Nadjm-Tehrani, S., Maluszynski, J. (1988). Algorithmic debugging with assertions. In: Abramson, H., Rogers, M.H. (eds) *Meta-Programming in Logic Programming: Proc. of META'*88, MIT Press, 1988, pp 501-521.

Flener, P. (1993). *Algorithm Synthesis from Incomplete Specifications*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve (Belgium), 1993.

Flener, P., Deville, Y. (1992). Towards stepwise, schema-guided synthesis of logic programs. In: Clement, T., Lau, K.-K. (eds) *Proc. of LOPSTR'*91. Springer-Verlag, 1992, pp 46-64.

Flener, P., Deville, Y. (1993). Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In: Jacquet, J.-M. (ed) *Constructing Logic Programs*, John Wiley & Sons, 1993.

Fribourg, L. (1990). Extracting logic programs from proofs that use extended Prolog execution and induction. In: Warren, D.H.D., Szeredi, P. (eds) *Proc. of ICLP'*90, MIT Press, 1990, pp 685-699. Extension in: Jacquet, J.-M. (ed) *Constructing Logic Programs*, Wiley, 1993.

Gegg-Harrison, T.S. (1989). *Basic Prolog schemata*. Research Report CS–1989–20, Duke University, Durham (NC, USA), 1989.

Gold, E.M. (1967). Language identification in the limit. *Info. and Control* 10(5):447-474, 1967.

Hansson, Å. (1980). *A Formal Development of Programs*. PhD thesis, University of Stockholm (Sweden), 1980.

Hogger, C.J. (1981). Derivation of logic programs. *J. of the ACM* 28(2):372-392, April 1981.

Jantke, K.-P. (1991). Monotonic and non-monotonic inductive inference. *New Generation Computing* 8(4):349–360, 1991.

Kanamori, T., Seki, H. (1986). Verification of Prolog programs using an extension of execution. In: Shapiro, E. (ed) *Proc. of ICLP'*86, LNCS 225, Springer-Verlag, 1986, pp 475-489.

Lange, S., Wiehagen, R. (1991). Polynomial-time inference of arbitrary pattern languages. *New Generation Computing* 8(4):361–370, 1991.

Lassez, J.-L., Maher, M.J., Marriott, K. (1987). Unification revisited. In Boscarol, M., Carlucci Aiello, L., Levi, G. (eds) *Proc. of the* 1986 *Workshop on the Foundations of Logic and Functional Programming*, LNCS 306, Springer-Verlag, 1987, pp 67-113.

Lau, K.-K., Prestwich, S.D. (1990). Top-down synthesis of recursive logic procedures from first-order logic specifications. In: Warren, D.H.D., Szeredi, P. (eds) *Proc. of ICLP'*90, MIT Press, 1990, pp 667-684.

Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill, 1974.

Plotkin, G.D. (1970). A note on inductive generalization. In: Meltzer, B., Michie, D. (eds) *Machine Intelligence*, Edinburgh University Press (UK), 5:153-163, 1970.

Reynolds, J.C. (1970). Transformational systems and the algebraic structure of atomic formulas. In: Meltzer, B., Michie, D. (eds) *Machine Intelligence*, Edinburgh University Press (UK), 5:135-151, 1970.

Shapiro, E. (1982). *Algorithmic Program Debugging*. PhD thesis, Yale University, New Haven (CT, USA), 1982. Also: MIT Press, ACM Distinguished Dissertation Series, 1983.

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

Smith, D.R. (1988). *The structure and design of global search algorithms*. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto (CA, USA), 1988.

Summers, P.D. (1977). A methodology for LISP program construction from examples. *J. of the ACM* 24(1):161-175, 1977.

Tinkham, N.L. (1990). *Induction of Schemata for Program Synthesis*. PhD thesis, Duke University, Durham (NC, USA), 1990.

Wiggins, G. (1992). Synthesis and transformation of logic programs in the Whelk proof development system. In: Apt, K. (ed), *Proc. of JICSLP'*92, MIT Press, 1992, pp 351–365.