Towards Stepwise, Schema-Guided Synthesis of Logic Programs

Pierre Flener
Institut d'Informatique, Université de Namur
Rue Grandgagnage 21, 5000 Namur, Belgium

Yves Deville
Unité d'Informatique, Université Catholique de Louvain
Place Ste Barbe 2, 1380 Louvain-la-Neuve, Belgium

Abstract

We present a general strategy for stepwise, sound and progressive synthesis of logic programs from specifications by examples and properties. We particularize this to schema-guided synthesis, and state a generic synthesis theorem. We justify some design choices for the development of a particular synthesis mechanism that is guided by a Divide-and-Conquer schema, is inductive and deductive, is interactive, and features a non-incremental presentation of examples. Some crucial steps of this mechanism are explained, and illustrated by a sample synthesis. We draw some conclusions on our results so far, state some related work, and outline future research directions.

1 Introduction

Program synthesis research (see [1]) aims at automating the passage from specifications to programs, as opposed to traditional, manual programming techniques. The key question here is: "what is a specification?". Today, an emerging consensus is that one may speak of "*synthesis*" when the specification does not explicitly reveal recursion or iteration. Otherwise, the technique could be classified as "*transformation*". In this introductory section, we define the starting point and the result of synthesis from examples and properties in a logic programming framework.

Definition 1 A *specification by examples and properties* of a procedure *r* consists of:

- a set *E(r)* of ground (input/output) examples of the behavior of *r*;
- a set *P(r)* of properties (first-order logic statements) of *r*.

Let $\mathfrak{R}$ be the relation one has in mind when elaborating such a specification of *r*. We shall call $\mathfrak{R}$ the "*intended relation*", in contrast to the relation actually specified, called

the "*specified relation*". This distinction is very important to software engineering in general, but crucial with incomplete specifications, where one deliberately admits a gap between the two.

The motivation for this specification format will be given in Section 3.1. Usually, *E(r)* is empty, or non-empty only for illustrative purposes. Sometimes, *P(r)* is empty, and the resulting specification by examples is (often) incomplete. Until Section 3, we shall just assume that at least one of *E(r)* and *P(r)* is non-empty.

We are actually only interested in synthesizing *algorithms*, rather than full-fledged programs in an existing programming language. Indeed, algorithm design in itself is already very hard, and we do not want to encumber ourselves with the additional burdens of algorithm optimization, transformation and implementation, that are all well-researched topics anyway (see [2], and some papers in this volume). Algorithms expressed in a logic formalism are here called "*logic descriptions*" (see [2]).

Definition 2 A *logic description* of a procedure *r*, denoted by *LD(r)*, consists of a formula of the form: $r(X,Y) \Leftrightarrow Def[X,Y]$, where *Def* is a first-order logic statement[1,2].

Executable Prolog programs can easily be derived from such logic descriptions [2].

The rest of this paper is organized as follows. In Section 2, we present a general strategy for stepwise, sound and progressive synthesis of logic descriptions from specifications by examples and properties. We particularize this to schema-guided synthesis, and state a generic synthesis theorem. In Section 3, we justify some design choices for the development of a particular synthesis mechanism. Some crucial steps of this mechanism are explained, and illustrated by a sample synthesis. In Section 4, we draw some conclusions on our results so far, state some related work, and outline future research directions.

2 A General Strategy for Stepwise Synthesis

In this section, we outline a general strategy for logic description synthesis. We focus on stepwise synthesis, i.e. there is a series of refinements towards a correct logic description:

$$LD_1(r)\ ,\ LD_2(r)\ ,\ \ldots\ ,\ LD_i\ (r)\ ,\ \ldots\ ,\ LD_f\ (r).$$

At each step, we want to measure the current logic description against the intended relation: in Section 2.1, we introduce logic description correctness criteria useful for characterizing soundness of synthesis. Across several steps, we want to measure the progression of the synthesized logic descriptions towards the intended relation: in Section 2.2, we introduce logic description comparison criteria useful for characterizing progression of syn-

1. The variables *X* and *Y* are assumed to be universally quantified over *LD(r)*.
2. *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 *a* and *b*, respectively.

thesis. In Section 2.3, we present a strategy for stepwise, sound and progressive synthesis of logic descriptions. In Section 2.4, we particularize this strategy for schema-guided synthesis. In Section 2.5, we state a generic synthesis theorem.

2.1 Correctness Criteria of Logic Descriptions

It is important to measure a logic description against its intended relation. Since we are only concerned with the declarative semantics of logic descriptions, we shall assume model-theoretic criteria for doing so, rather than proof-theoretic ones.

A logic description *LD(r)* is (totally) correct wrt $\Re$ iff:

$$LD(r) \models r(a,b) \text{ iff } <a,b> \in \Re$$

$$LD(r) \models \sim r(a,b) \text{ iff } <a,b> \notin \Re$$

i.e. iff the predicate *r* is interpreted as the relation $\Re$ in all (Herbrand) models of *LD(r)*.

With logic description design by structural induction on some parameter, one only has to focus on a single (Herbrand) interpretation (because the truth value of a ground atom *r(a,b)* will be the same in all (Herbrand) models of *LD(r)*, see [2]).

Let thus $\Im$ be a Herbrand interpretation such that:

- *r(a,b)* is true in $\Im$ iff $\Re$*(a,b)*
- $\Im$ is a model of all primitive predicates (such as "=")
- $\Im$ is a model of all predicates used in the property set *P(r)*.

In the sequel, all logic formulas will be interpreted in $\Im$.

Let *LD(r)* be: $r(X,Y) \Leftrightarrow Def[X,Y]$.

There are three layers of correctness criteria. The (total) correctness of a logic description wrt its intended relation $\Re$ can now be redefined as follows:

Definition 3 *LD(r) is (totally) correct wrt* $\Re$ iff $r(X,Y) \Leftrightarrow Def[X,Y]$ is true in $\Im$.

(Total) correctness can be decomposed into *partial correctness* (the relation defined by *LD(r)* is included in $\Re$) and *completeness* ($\Re$ is included in the relation defined by *LD(r)*):

Definition 4 *LD(r) is partially correct wrt* $\Re$ iff $r(X,Y) \Leftarrow Def[X,Y]$ is true in $\Im$.

Definition 5 *LD(r) is complete wrt* $\Re$ iff $r(X,Y) \Rightarrow Def[X,Y]$ is true in $\Im$.

Next comes completeness of a logic description wrt an example set *E(r)*:

Definition 6 *LD(r) is complete wrt E(r)* iff *Def[a,b]* is true in $\Im$, for every example *r(a,b)* of *E(r)*.

Note that partial correctness of *LD(r)* wrt *E(r)* is an irrelevant concept, because we are not interested in logic descriptions that do not cover all the examples of *E(r)*.

Finally, there is consistency of examples and properties wrt the intended relation $\Re$:

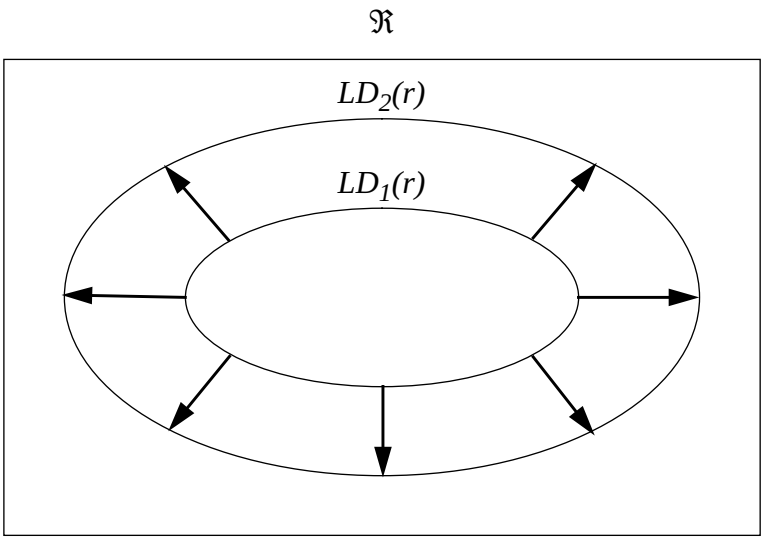

Figure 1 Partial Correctness Preserving (Upward) Progression

Definition 7 *E(r) is consistent with* ℜ iff every example of *E(r)* is true in ℑ.

Definition 8 *P(r) is consistent with* ℜ iff every property of *P(r)* is true in ℑ.

The specified relation of a consistent specification is a subset of the intended relation.

There is obviously no formal definition of the intended relation ℜ, so some correctness criteria cannot be applied in a formal way. But these correctness criteria can be used to state features and heuristics of the synthesis process.

2.2 Comparison Criteria of Logic Descriptions

It is also important to compare logic descriptions of the same predicate. Let:

- $LD_1(r)$: $r(X,Y) \Leftrightarrow Def_1[X,Y]$
- $LD_2(r)$: $r(X,Y) \Leftrightarrow Def_2[X,Y]$

be two logic descriptions. Intuitively, $LD_1(r)$ is more general than $LD_2(r)$ iff Def_1 is "more often" true than Def_2. More formally:

Definition 9 *$LD_1(r)$ is more general than $LD_2(r)$* iff $Def_2 \Rightarrow Def_1$ is true in ℑ.

This is denoted by $LD_1(r) \geq LD_2(r)$. The fact of being "*less general*" (≤) is defined dually. The set of logic descriptions of a given predicate is partially ordered under "≤" and "≥". Two logic descriptions, each more general than the other, are "*equivalent*" (≅).

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

Definition 10 If (see Figure 1):

- $LD_2(r) \geq LD_1(r)$
- $LD_2(r)$ is partially correct wrt ℜ

then $LD_2(r)$ is a *better (partially correct) approximation* of ℜ than $LD_1(r)$.

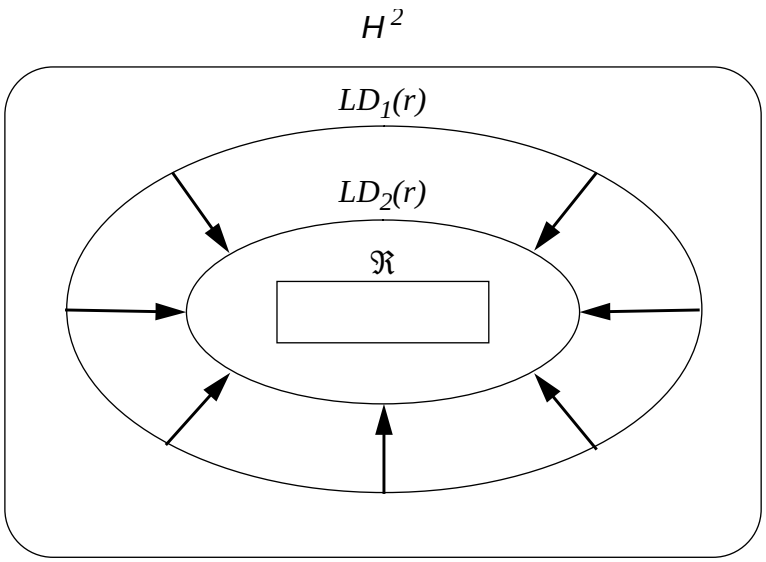

Figure 2 Completeness Preserving (Downward) Progression

Dually, there is a criterion for downward (or completeness preserving) progression:

Definition 11 If (see Figure 2, where H stands for the Herbrand universe):

- $LD_2(r) \leq LD_1(r)$
- $LD_2(r)$ is complete wrt $\Re$

then $LD_2(r)$ is a *better (complete) approximation* of $\Re$ than $LD_1(r)$.

2.3 A Strategy for Stepwise, Sound and Progressive Synthesis

We now have the terminology for defining a strategy for stepwise, sound and progressive synthesis. The first question is to determine in which direction we want to progress: upwards, or downwards? A natural choice seems to be upward progression.

A First Strategy

The first formulation of a strategy of f steps thus reads as follows.

At **Step *1***, "create" $LD_1(r)$ such that:

- $LD_1(r)$ is partially correct wrt $\Re$.

At **Step *i*** $(1<i\leq f)$, transform $LD_{i-1}(r)$ into $LD_i(r)$ such that:

- $LD_i(r)$ is a better (partially correct) approximation of $\Re$ than $LD_{i-1}(r)$.

Our particular synthesis mechanism (see Section 3) even has the following feature:

- $LD_i(r)$ is complete wrt $E(r)$, $(1\leq i\leq f)$.

The Completeness Issue

An objection arises: what about the completeness of the synthesized logic descriptions? We must remember that synthesis is here example-based, i.e. that constants from $E(r)$ will

inevitably appear in the synthesized logic descriptions, thus destroying all possible completeness. We are thus forced to raise the completeness issue. We do it by allowing simultaneous downward progression of the series of logic descriptions that progresses upwards.

Definition 12 Let γ be a total function in the set of logic descriptions, such that $\gamma(LD(r))$ is $LD(r)$ without its equality atoms involving constants introduced from $E(r)$.

It can be shown that γ is a generalization function, i.e. $\gamma(LD(r)) \geq LD(r)$.

Example Let $LD_3(compress)$ be:

```
compress(L,CL) ⇔
    L=[]      ∧ L=[] ∧ CL=[]
  ∨ L=[HL|TL] ∧ L=[a] ∧ CL=[<a,1>]
              ∧ HL=a ∧ TL=[]
  ∨ L=[HL|TL] ∧ L=[b,b] ∧ CL=[<b,2>]
              ∧ HL=b ∧ TL=[b]
  ∨ L=[HL|TL] ∧ L=[c,d] ∧ CL=[<c,1>,<d,1>]
              ∧ HL=c ∧ TL=[d]
```

where the **bold** atoms are synthesized atoms, and the other atoms stem from examples E_1 to E_4 of Figure 4 below. Thus, $\gamma(LD_3(compress))$ is:

```
compress(L,CL) ⇔
    L=[]
  ∨ L=[HL|TL]
```

We shall use this function γ to enhance our strategy.

An Enhanced Strategy

The strategy now reads:

At **Step *1***, "create" $LD_1(r)$ such that:

• $LD_1(r)$ is partially correct wrt $\Re$

• $\gamma(LD_1(r))$ is complete wrt $\Re$.

At **Step *i*** ($1<i\leq f$), transform $LD_{i-1}(r)$ into $LD_i(r)$ such that:

• $LD_i(r)$ is a better (partially correct) approximation of $\Re$ than $LD_{i-1}(r)$

• $\gamma(LD_i(r))$ is a better (complete) approximation of $\Re$ than $\gamma(LD_{i-1}(r))$.

At **Step *f***, "obtain" $LD_f(r)$ such that:

• $LD_f(r) \cong \gamma(LD_f(r))$.

Thus, the relation defined by $LD_f(r)$ is the intended relation $\Re$, hence $LD_f(r)$ is correct wrt $\Re$. Convergence of the synthesis process is thus achieved. Since there will usually be a gap between the intended relation $\Re$ and the specified one, the given strategy cannot be fully automated, but should serve as a guideline for interactive synthesis.

$$
\begin{aligned}
R(X,Y) \Leftrightarrow\ & Minimal(X) && \wedge\ Solve(X,Y)\\
\vee\ \vee_{1\le k\le C}\ & NonMinimal(X) && \wedge\ Decompose(X,\mathbf{HX},\mathbf{TX})\\
& && \wedge\ Discriminate_k(\mathbf{HX},\mathbf{TX},Y)\\
& && \wedge\ R(\mathbf{TX},\mathbf{TY})\\
& && \wedge\ Process_k(\mathbf{HX},\mathbf{HY})\\
& && \wedge\ Compose_k(\mathbf{HY},\mathbf{TY},Y)
\end{aligned}
$$

Figure 3 The Divide-and-Conquer Logic Description Schema

2.4 Schema-Guided Synthesis

Algorithm schemata are an old idea of computer science (see an early survey in [3]). They are template algorithms with fixed control flows. They embody the essence of algorithm design strategies (e.g. Divide-and-Conquer, Generate-and-Test, Global Search, …) and are thus an invaluable knowledge source for guiding (semi-)automated algorithm design.

Example Loosely speaking, 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, i.e. if *X* is non-minimal, we decompose *X* into a series ***HX*** of heads of *X* and a series ***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 heads ***HX*** are processed into a series of heads ***HY*** of *Y*. Finally, *Y* is composed from its heads ***HY*** and tails ***TY***. But it may happen that different process and compose patterns emerge for the non-minimal form of *X*: we have to discriminate between them according to the values of ***HX***, ***TX*** and *Y*, unless non-determinism requires such alternatives.

Logic description schemata can be expressed as second-order logic descriptions. For instance, logic descriptions designed by a Divide-and-Conquer strategy, and having one single minimal case and one single non-minimal case, will fit the schema of Figure 3.

A Refined Strategy

We can now further refine the above strategy of logic description synthesis:

At **Step *i*** (*1<i<f)*:

• synthesize instantiation(s) of some predicate variable(s) of the schema

• introduce some “trailing” equality atoms involving constants from *E(r).*

2.5 A Generic Synthesis Theorem

We now state a generic synthesis theorem explaining how synthesis step *i* can achieve sound and progressive synthesis.

Theorem 1 *Generic Synthesis Theorem*
Let $LD_{i-1}(r)$ be: $r(X,Y) \Leftrightarrow \vee_{1\leq j\leq m} A_j \wedge E_j$
and $LD_i(r)$ be: $r(X,Y) \Leftrightarrow \vee_{1\leq j\leq m} A_j \wedge B_j \wedge E_j \wedge F_j$
where A_j, B_j are formulas without equality atoms involving constants introduced from $E(r)$, and E_j, F_j are conjunctions of equality atoms with constants introduced from $E(r)$.
Thus $\gamma(LD_{i-1}(r))$ is: $r(X,Y) \Leftrightarrow \vee_{1\leq j\leq m} A_j$
and $\gamma(LD_i(r))$ is: $r(X,Y) \Leftrightarrow \vee_{1\leq j\leq m} A_j \wedge B_j$.
The following assertions hold:

(1.1) If $LD_{i-1}(r)$ is partially correct wrt $\Re$
and $A_j \wedge E_j \Rightarrow B_j \wedge F_j$ $(1\leq j\leq m)$
then $LD_i(r)$ is a *better (partially correct) approximation* of $\Re$ than $LD_{i-1}(r)$.

(1.2) If $\gamma(LD_{i-1}(r))$ is complete wrt $\Re$
and $\Re(X,Y) \wedge A_j \Rightarrow B_j$ $(1\leq j\leq m)$
then $\gamma(LD_i(r))$ is a *better (complete) approximation* of $\Re$ than $\gamma(LD_{i-1}(r))$.

Proof 1 The proof is straightforward (see [4]).

The second condition of assertion (1.1) ensures that the atoms introduced by Step *i* are redundant with the already existing ones: in other words, we actually have $LD_i(r) \cong LD_{i-1}(r)$. This is not a disaster, because strict progression is achieved by the generalizations. The second condition of assertion (1.2) ensures that the atoms introduced by Step *i* are "redundant" with the intended relation.

For a specific synthesis mechanism whose steps *2* to *f-1* fit into the framework of this theorem, it thus suffices to prove that the method of each step ensures its particular instantiations of the conditions of both assertions. The first conditions of both assertions actually need not be proved, because Step 1 establishes them, and Theorem 1 preserves them.

3 A Particular Synthesis Mechanism

In this section, we present a particular synthesis mechanism. In Section 3.1, we explain the design decisions taken while instantiating the parameters of the framework of Section 2. In Section 3.2 to Section 3.5, the steps of the synthesis mechanism are detailed. Let's first present a running example for this section:

Example *Data compression*. The *compress(L,CL)* procedure succeeds on facts such as:

compress([a, a, a, b, b, a, c, c, c, c], [<a, 3>, <b, 2>, <a, 1>, <c, 4>]).

Compression is performed without forward checking, i.e. each plateau is compressed regardless of whether another plateau of the same character occurs later in the list *L*.

3.1 Decisions

A series of decisions have to be taken before producing a specific synthesis mechanism within the framework of Section 2.

Which Schema?

We adopt the Divide-and-Conquer schema of Figure 3 for guiding synthesis. Indeed, the class of algorithms that can be designed by this strategy is fairly large and important. There will be eight steps to our mechanism:

- Step 1: Creation of a first approximation
- Step 2: Synthesis of *Minimal* and *NonMinimal*
- Step 3: Synthesis of *Decompose*
- Step 4: Synthesis of the recursive atoms
- Step 5: Synthesis of *Solve*
- Step 6: Synthesis of *Process* and *Compose*
- Step 7: Synthesis of *Discriminate*
- Step 8: Generalization

There are two phases: an "*expansion phase*" (Steps 1 to 4), where equality atoms involving constants introduced from *E(r)* are added, and a "*reduction phase*" (Steps 5 to 8), where the actual synthesis takes place. In this paper, we shall focus on the reduction phase.

Note that the schema is not an input to the synthesis mechanism. It is rather a convenient way of explaining how synthesis works.

Which Language for Specifications?

Examples are a very appealing means of conveying information about a relation: they are easy to elaborate/understand, and they implicitly reveal structural manipulations of the parameters. Our plan is thus to use non-empty example sets in specifications. However, examples alone are an incomplete information source, and they cannot explicitly convey semantic operations/tests on parameters. We believe the specifier knows these additional details, and ought thus to be able to provide them, rather than have the synthesis mechanism guess them! We depart thus from traditional example-based synthesis and add a non-empty property set to our specifications. But we would like to give properties the same appeal as examples. It turns out that non-recursive Horn clauses, plus negation, are a very convenient choice. Since synthesis should start from a non-recursive specification, we must disallow recursive properties.

```
E(compress)={ compress([],[])(E1)
              compress([a],[<a,1>])(E2)
              compress([b,b],[<b,2>])(E3)
              compress([c,d],[<c,1>,<d,1>])(E4)
              compress([e,e,e],[<e,3>])(E5)
              compress([f,f,g],[<f,2>,<g,1>])(E6)
              compress([h,i,i],[<h,1>,<i,2>])(E7)
              compress([j,k,l],[<j,1>,<k,1>,<l,1>])}(E8)
P(compress)={ compress([X],[<X,1>])(P1)
              compress([X,Y],[<X,2>]) ⇐ X=Y(P2)
              compress([X,Y],[<X,1>,<Y,1>]) ⇐ X≠Y}(P3)
```

Figure 4 Sample versions of *E(compress)* and *P(compress)*

Example Figure 4 gives sample versions of *E(compress)* and *P(compress)*. Note that properties P_1 to P_3 generalize examples E_2 to E_4, respectively. They also make explicit why examples E_3 and E_4 behave differently: equality/disequality of the first two elements, rather than any other criterion.

Our properties turn out to be an incomplete specification source, too. It is thus important that we do not strive for completely automated synthesis, but rather for interactive synthesis, so as to cope with incompleteness.

Examples give rise to inductive synthesis (generalization [5], [6], [7], learning [8], [9]), whereas axiomatic specifications (and thus properties) give rise to deductive synthesis (proofs-as-programs [10], [11], [12], rewriting [13], [14], [15]). Since we have both specification types, we want to avoid using only one kind of inference, and thus degrading the non-used information source into validation information. We shall therefore strive for inductive **and** deductive synthesis, using whichever inference type is best suited at each step. This approach of course precludes synthesis whenever only examples or only properties are given. It also gives a constructive role to each information type.

How to Present the Examples?

There are two extreme ways of presenting examples: *"one-by-one"* (*"incrementally"*), or "*all-at-once*". The former approach, advocated in [8] and by the machine learning school of thought, has some nice convergence properties. But we shall adhere to the school of thought advocated in [7], where the examples are presented all-at-once, so that a maximum of information is available at each step.

```
compress(L,CL) ⇔
    L=[]          ∧ CL=[]
  ∨ L=[H1]        ∧ CL=[<H1,1>]
  ∨ L=[H1,H2|TL]  ∧ H1=H2
                  ∧ compress([H2|TL],[<H2,M>|T])
                  ∧ add(M,1,N) ∧ CL=[<H1,N>|T]
  ∨ L=[H1,H2|TL]  ∧ H1≠H2
                  ∧ compress([H2|TL],TCL)
                  ∧ CL=[<H1,1>|TCL])
```

Figure 5 A sample version of *LD(compress)*

Which Language for Logic Descriptions?

Since the Divide-and-Conquer schema has a definition part in disjunctive normal form, we shall stick to such logic descriptions at all steps of synthesis.

Example To give a preliminary glimpse of what the synthesis mechanism is supposed to produce, Figure 5 gives a sample version of *LD(compress)*, constructed by the methodology of [2].

Let's now see how the eight synthesis steps identified above can produce an equivalent logic description from the specification by examples and properties of Figure 4.

3.2 The Expansion Phase of Synthesis (Steps 1 - 4)

During the *expansion phase*, the following steps are performed:

- creation of a first approximation (Step 1), via re-expression of the example set as a logic description. In order to ensure applicability of Theorem 1 for Steps 2 - 7, we must assume that *E(r)* and *P(r)* are consistent wrt the intended relation;
- synthesis of *Minimal* and *NonMinimal* (Step 2), from a set of predefined, type-specific, parameterized instantiations;
- synthesis of *Decompose* (Step 3), from a set of predefined, type-specific, parameterized instantiations;
- synthesis of the recursive atoms (Step 4), via deductive and/or analogical reasoning.

Steps 1 to 3 are quite straightforward, but Step 4 isn't (see the results in [4]).

Example Figure 6 shows $LD_4(compress)$, where disjunct D_i corresponds to example E_i. Note that there is one minimal case (disjunct D_1), and one non-minimal case (disjuncts D_2 to D_8). Also note that $\gamma(LD_4(compress))$ is:

```
compress(L,CL) ⇔
    L=[]      ∧ L=[] ∧ CL=[](D1)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[a] ∧ CL=[<a,1>]
              ∧ HL=a ∧ TL=[] ∧ TCL=[](D2)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[b,b] ∧ CL=[<b,2>]
              ∧ HL=b ∧ TL=[b] ∧ TCL=[<b,1>](D3)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[c,d] ∧ CL=[<c,1>,<d,1>]
              ∧ HL=c ∧ TL=[d] ∧ TCL=[<d,1>](D4)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[e,e,e] ∧ CL=[<e,3>]
              ∧ HL=e ∧ TL=[e,e] ∧ TCL=[<e,2>](D5)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[f,f,g] ∧ CL=[<f,2>,<g,1>]
              ∧ HL=f ∧ TL=[f,g]
              ∧ TCL=[<f,1>,<g,1>](D6)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[h,i,i] ∧ CL=[<h,1>,<i,2>]
              ∧ HL=h ∧ TL=[i,i] ∧ TCL=[<i,2>](D7)
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
              ∧ L=[j,k,l] ∧ CL=[<j,1>,<k,1>,<l,1>]
              ∧ HL=j ∧ TL=[k,l]
              ∧ TCL=[<k,1>,<l,1>](D8)
```

Figure 6 *LD_4(compress)*

```
compress(L,CL) ⇔
    L=[]
  ∨ L=[HL|TL] ∧ compress(TL,TCL).
```

Let's now have a close look at the steps of the *reduction phase*.

3.3 Synthesis of *Solve*, *Compose* and *Process* (Steps 5 - 6)

At Step 5, the synthesis of *Solve* for the minimal case is similar to what happens at Steps 6 and 7 for the non-minimal case, and we shall thus not delve into details. Just consider that *CL=L* is synthesized as an instance of *Solve(L,CL)*.

At Step 6, the aim is to transform $LD_5(r)$ into $LD_6(r)$ which fits the following schema:

$$
\begin{aligned}
\texttt{r(X,Y)} \Leftrightarrow\ & \texttt{minimal(X)} && \wedge\ \texttt{solve(X,Y)} \\
& && \wedge\ \vee_{1\le i<b}\ \texttt{X=}x_i \wedge \texttt{Y=}y_i \\
\vee\ & \vee_{1\le k\le \mathbf{C}}\ \texttt{nonMinimal(X)} && \wedge\ \texttt{decompose(X,HX,TX)} \\
& && \wedge\ \texttt{r(TX,TY)} \\
& && \wedge\ \mathbf{Process_k(HX,HY)} \wedge \mathbf{Compose_k(HY,TY,Y)} \\
& && \wedge\ \vee_{i\in|k|}\ \texttt{X=}x_i \wedge \texttt{Y=}y_i \\
& && \qquad \wedge\ \texttt{HX=}hx_i \wedge \texttt{TX=}tx_i \\
& && \qquad \wedge\ \mathbf{HY=}hy_i \wedge \texttt{TY=}ty_i
\end{aligned}
$$

i.e. we want to partition the non-minimal disjuncts into *C* equivalence classes *|1|*, …, *|C|* whose members have equal instantiations of *Process* and *Compose*.

For *Process*, we take the following approach:

- first assume that "=" is a suitable instantiation of *Process*, and synthesize a partition. Note that this assumption is also successful when "=" is actually not a suitable instantiation of *Process*, but the latter can be (loop-)merged with *Compose*;
- otherwise assume that the synthesis of *Process* requires other techniques (not mentioned here), and synthesize a partition, leaving *Process* uninstantiated.

We have two methods to synthesize a partition, and thus an instantiation of *Compose*:

- computation of most specific generalizations: this *MSG Method* will be successful if *Compose* can be expressed as a conjunction of equality atoms;
- synthesis from an inferred specification by examples and properties: this *Synthesis Method* applies when *Compose* itself needs a full-fledged recursive logic description. We shall not explain here when to apply this method, nor how a specification for *Compose* can be inferred from the current logic description (see [4] for details).

The MSG Method fulfills the conditions of Theorem 1. Since the Synthesis Method eventually boils down to using the MSG Method, it is, by induction, sound and progressive as well, provided the specification inference is sound.

Example Assuming *Process* is "=", the MSG Method synthesizes the disjunct partition:

$$\{\ \{D_2, D_4, D_7, D_8\}, \{D_3, D_5, D_6\}\ \}$$

- for the first class, we have the following instances of <*HCL, TCL, CL*>:

`<a,[],`	`[<a,1>]>`	(D_2)
`<c,[<d,1>],`	`[<c,1>,<d,1>]>`	(D_4)
`<h,[<i,2>],`	`[<h,1>,<i,2>]>`	(D_7)
`<j,[<k,1>,<l,1>],`	`[<j,1>,<k,1>,<l,1>]>`	(D_8)

Hence the MSG: *<H,T,[<H,1>|T]>*.

And *Compose$_1$(HCL,TCL,CL)* is thus: *CL=[<HCL, 1>|TCL]*

```
compress(L,CL) ⇔
    L=[]       ∧ CL=L
               ∧ L=[] ∧ CL=[]
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
               ∧ CL=[<HL,1>|TCL]
               ∧ HL=a ∧ TL=[] ∧ TCL=[] ∧ …
                 ∨ HL=c ∧ TL=[d] ∧ TCL=[<d,1>] ∧ …
                 ∨ HL=h ∧ TL=[i,i] ∧ TCL=[<i,2>] ∧ …
                 ∨ HL=j ∧ TL=[k,l] ∧ TCL=[<k,1>,<l,1>] ∧ …
  ∨ L=[HL|TL] ∧ compress(TL,TCL)
               ∧ TCL=[<HL,M>|TTCL] ∧ CL=[<HL,s(M)>|TTCL]
               ∧ HL=b ∧ TL=[b] ∧ TCL=[<b,1>] ∧ …
                 ∨ HL=e ∧ TL=[e,e] ∧ TCL=[<e,2>] ∧ …
                 ∨ HL=f ∧ TL=[f,g] ∧ TCL=[<f,1>,<g,1>] ∧ …
```

Figure 7 LD_6(compress)

• similarly, for the second class, *Compose*$_2$*(HCL,TCL,CL)* is:

TCL=[<HCL, M>|TTCL] ∧ *CL=[<HCL, s(M)>|TTCL]*.

Hence *LD*$_6$*(compress)* looks as depicted in Figure 7. Note that the assumption that *Process* is "=" works because *Process* and *Compose* could be merged.

3.4 Synthesis of *Discriminate* (Step 7)

At Step 7, the aim is to transform *LD*$_6$*(r)* into *LD*$_7$*(r)* which fits the following schema:

```
r(X,Y) ⇔ minimal(X)    ∧ solve(X,Y)
                       ∧ ∨1≤i<b X=xi ∧ Y=yi
  ∨ ∨1≤k≤c nonMinimal(X)∧ decompose(X,HX,TX)
                       ∧ Discriminatek(HX,TX,Y)
                       ∧ r(TX,TY)
                       ∧ processk(HX,HY) ∧ composek(HY,TY,Y)
                       ∧ ∨i∈|k|  X=xi ∧ Y=yi
                              ∧ HX=hxi ∧ TX=txi
                              ∧ HY=hyi ∧ TY=tyi
```

This objective is achieved by two consecutive tasks:

• synthesis of specialized instantiations of the *Discriminate*$_k$

• generalization of these specialized instantiations of the *Discriminate*$_k$.

```
compress(L,CL) ⇔
    L=[]      ∧ CL=L
              ∧ L=[] ∧ Cl=[]
  ∨ L=[HL|TL] ∧ (TL=[]) ∨ (TL=[HTL|TTL] ∧ HL≠HTL)
              ∧ compress(TL,TCL)
              ∧ CL=[<HL,1>|TCL]
              ∧ HL=a ∧ TL=[] ∧ TCL=[] ∧ …
                ∨ HL=c ∧ TL=[d] ∧ TCL=[<d,1>] ∧ …
                ∨ HL=h ∧ TL=[i,i] ∧ TCL=[<i,2>] ∧ …
                ∨ HL=j ∧ TL=[k,l] ∧ TCL=[<k,1>,<l,1>] ∧ …
  ∨ L=[HL|TL] ∧ TL=[HTL|TTL] ∧ HL=HTL
              ∧ compress(TL,TCL)
              ∧ TCL=[<HL,M>|TTCL] ∧ CL=[<HL,s(M)>|TTCL]
              ∧ HL=b ∧ TL=[b] ∧ TCL=[<b,1>] ∧ …
                ∨ HL=e ∧ TL=[e,e] ∧ TCL=[<e,2>] ∧ …
                ∨ HL=f ∧ TL=[f,g] ∧ TCL=[<f,1>,<g,1>] ∧ …
```

Figure 8 *LD_7(compress)*

The first task is done using a *Proofs-as-Programs Method* (see [10], [11], [12]): instantiations of the *Discriminate*$_k$ are extracted from the proof that:

$$\gamma(LD_6(r)) \mid\!- P(r) .$$

The second task is heuristic-driven, i.e. after applying some generalization heuristics, we postulate that the resulting discriminators are the intended ones.

Some theoretical aspects of this step are further detailed in [4].

Example For our *compress* procedure:

- the proof of P_1 reveals a partial, specialized discriminator for the first class:

$$discriminate_1(HL, [], [<HL, 1>]).$$

- the proof of P_3 reveals another partial, specialized discriminator for the first class:

$$discriminate_1(HL, [HTL], [<HL, 1>, <HTL, 1>]) \Leftarrow HL \neq HTL.$$

We join both, generalize *TL*, eliminate the *CL*=… atoms, and "postulate":

$$discriminate_1(HL, TL, CL) \Leftrightarrow TL=[] \vee (TL=[HTL|TTL] \wedge HL \neq HTL).$$

- the proof of P_2 reveals a specialized discriminator for the second class:

$$discriminate_2(HL, [HTL], [<HL, 2>]) \Leftarrow HL=HTL.$$

We generalize *TL*, eliminate the *CL*=… atom, and "postulate":

$$discriminate_2(HL, TL, CL) \Leftrightarrow TL=[HTL|TTL] \wedge HL=HTL.$$

Hence *LD_7(compress)* looks as depicted in Figure 8.

```
compress(L,CL) ⇔
    L=[]      ∧ CL=L
  ∨ L=[HL|TL] ∧ TL=[]
              ∧ compress(TL,TCL)
              ∧ CL=[<HL,1>|TCL]
  ∨ L=[HL|TL] ∧ TL=[HTL|TTL] ∧ HL≠HTL
              ∧ compress(TL,TCL)
              ∧ CL=[<HL,1>|TCL]
  ∨ L=[HL|TL] ∧ TL=[HTL|TTL] ∧ HL=HTL
              ∧ compress(TL,TCL)
              ∧ TCL=[<HL,M>|TTCL] ∧ CL=[<HL,s(M)>|TTCL]
```

Figure 9 *LD_8(compress)*

3.5 Generalization (Step 8)

At Step 8, the aim is to transform *LD_7(r)* into *LD_8(r)* which looks like:

$$
\begin{aligned}
r(X,Y) \Leftrightarrow\ & minimal(X) && \wedge\ solve(X,Y)\\
\vee\ & \vee_{1\le k\le c}\ nonMinimal(X) && \wedge\ decompose(X,HX,TX)\\
& && \wedge\ discriminate_k(HX,TX,Y)\\
& && \wedge\ r(TX,TY)\\
& && \wedge\ process_k(HX,HY) \wedge\ compose_k(HY,TY,Y)
\end{aligned}
$$

This is simply achieved by postulating that *LD_8(r)* is *γ(LD_7(r))*.

Example *LD_8(compress)* looks as depicted in Figure 9. Note that this logic description can be proven to be equivalent to the one given in Figure 5.

4 Conclusions

We have shown how to perform a stepwise, schema-guided, inductive and deductive, non-incremental synthesis of logic descriptions from examples and properties. Most steps are non-deterministic. In this last section, we shall present the framework, results and contributions of this research, and mention some related research, as well as future research.

4.1 Framework, Results and Contributions

This research is led within the framework of the FOLON research project (Université de Namur, Belgium) whose objective is twofold. First, it aims at elaborating a methodology of logic program development, such as described in [2]. Second, it aims at designing an integrated set of tools supporting this methodology. Our research tackles the aspects of logic program synthesis from examples and properties.

The main results of our work on logic program synthesis so far are the definition of a synthesis calculus, the identification of a particular synthesis mechanism, and the development of methods for each of its steps. The descriptions of Steps 5 to 7 in this paper are only target scenarios,a complete survey of all results can be found in [4].

One of the originalities of our approach is the combination of examples with properties, so as to cope with some classical problems of example-based synthesis.

4.2 Related Research

Pointers to related research in program synthesis have been given throughout the text, and we have already stressed in detail how our approach differs from the state of the art. The use of schemata is also advocated in [16], [17], [18], [9] (Divide-and-Conquer), [19] (Global Search), and others, although sometimes in different contexts (e.g. programming tutors/assistants). An early study of the concept of "most specific generalization" is [20].

4.3 Future Research

In the future, we plan to pursue research on the following aspects:

- development of a "proof-of-concept" implementation (in Prolog) of the synthesis mechanism. This should allow the identification of points of interaction with the specifier so that we can cope with incompleteness: wherever inductive reasoning is used, the specifier should be able to give his feedback. It is important to keep this dialogue easy, i.e. a question/answer method asking for the classification of ground atoms as examples/counter-examples seems to be an appropriate choice;
- incorporation of counter-examples in the specifications, the general synthesis strategy, and the synthesis mechanism. Indeed, negative information is quite useful in avoiding over-generalization during inductive reasoning;
- formulation of a choice methodology for examples and properties: it is important to guide the specifier towards choosing relevant examples and properties. This reduces interaction with the specifier, and results thus in highly automated synthesis.

Acknowledgments

The authors gratefully acknowledge many insightful discussions with B. Le Charlier (Université de Namur, Belgium). Parts of the results presented here were found while the first author was on leave at Duke University (NC, USA): many thanks to Prof. A. W. Biermann and Prof. D. W. Loveland for their interest in our research. The first author and the FOLON project are supported by the Belgian National Incentive Program for Fundamental Research in AI.

References

[1] Biermann AW. *Automatic Programming*. In: Encyclopedia of Artificial Intelligence. John Wiley & Sons, 1987, pp 18-35. (A second, extended version is in print).

[2] Deville Y. *Logic Programming - Systematic Program Development*. Addison Wesley, Reading (MA, USA), 1990.

[3] Manna Z. *Mathematical Theory of Computation*. McGraw-Hill, New York (NY, USA), 1974.

[4] Flener P. *Towards Programming by Examples and Properties*. Research Report CS-1991-09, Duke University, Durham (NC, USA), 1991.

[5] Biermann AW. *Dealing with Search*. In: Biermann AW, Guiho G and Kodratoff Y (eds) *Automatic Program Construction Techniques*. Macmillan Publishing Company, New York (NY, USA), 1984, pp 375-392.

[6] Biermann AW and Smith DR. *A Production Rule Mechanism for Generating LISP Code*. IEEE Transactions on Systems, Man and Cybernetics 1979; 5:260-276.

[7] Summers P. *A Methodology for LISP Program Construction from Examples*. Journal of the ACM 1977; 1:161-175.

[8] Shapiro E. *Algorithmic Program Debugging*. PhD thesis. MIT Press, Cambridge (MA, USA), 1982.

[9] Tinkham NL. *Induction of Schemata for Program Synthesis*. PhD thesis, Research Report CS-1990-14, Duke University, Durham (NC, USA), 1990.

[10] Bundy A, Smaill A and Wiggins G. *The Synthesis of Logic Programs from Inductive Proofs*. In: Lloyd JW (ed) *Computational Logic*. Springer Verlag, 1990, pp 135-149.

[11] Fribourg L. *Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction*. In: Proceedings of ICLP-90, MIT Press, Cambridge (MA, USA), 1990, pp 685-699.

[12] Manna Z and Waldinger R. *Synthesis: Dreams ⇒ Programs*. IEEE Transactions on Software Engineering 1979; 4:294-328.

[13] Clark KL. *The Synthesis and Verification of Logic Programs*. Research Report DOC 81/36, Imperial College, London (UK), 1981.

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

[15] Hogger CJ. *Derivation of Logic Programs*. Journal of the ACM 1981; 2:372-392.

[16] Burnay J and Deville Y. *Generalization and Program Schemata*. In: Proceedings of NACLP-89, MIT Press, Cambridge (MA, USA), 1989, pp 409-425.

[17] Gegg-Harrison TS. *Basic Prolog Schemata*. Research Report CS-1989-20, Duke University, Durham (NC, USA), 1989.

[18] Smith DR. *Top-Down Synthesis of Divide-and-Conquer Algorithms*. Artificial Intelligence 1985, 27:43-96.

[19] Smith DR. *The Structure and Design of Global Search Algorithms*. Technical Report KES.U.87.12, Kestrel Institute, Palo Alto (CA, USA), 1988.

[20] Plotkin GD. *A Note on Inductive Generalization*. Machine Intelligence 1970; 5:153-163, Edinburgh University Press (Scotland), 1970.