LOGIC PROGRAM SYNTHESIS

YVES DEVILLE AND KUNG-KIU LAU

▷ This paper presents an overview and a survey of logic program synthesis. Logic program synthesis is interpreted here in a broad way; it is concerned with the following question: given a specification, how do we get a logic program satisfying the specification? Logic programming provides a uniquely nice and uniform framework for program synthesis since the specification, the synthesis process, and the resulting program can all be expressed in logic.

Three main approaches to logic program synthesis by formal methods are described: constructive synthesis, deductive synthesis, and inductive synthesis. Related issues such as correctness and verification, as well as synthesis by informal methods, are briefly presented.

Our presentation is made coherent by employing a unified framework of terminology and notation, and by using the same running example for all the approaches covered. This paper thus intends to provide an assessment of existing work and a framework for future research in logic program synthesis. ◁

1. INTRODUCTION

Program synthesis refers to the elaboration of a program in some systematic manner, starting from a specification, that is a statement describing what the program should do. A specification may have many forms. We can distinguish formal specifications from informal ones. For the latter, the synthesis process cannot be totally formalized and can only be partially automated. We thus distinguish between *synthesis by formal methods* and *synthesis by informal methods*.

Program synthesis in general has been an active area of research outside logic programming. See [8, 3, 50, 10], for example, for a presentation of the major achievements. In the

Address correspondence to Yves Deville, Unité d'Informatique, Université Catholique de Louvain, Place Ste Barbe, 2, B-1348 Louvain-la-Neuve, Belgium or Kung-Kiu Lau, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, England.

Received May 1993; accepted January 1994.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

early days of logic programming, logic program synthesis was one of the first areas of active research, mainly focusing on manual program derivation. By the middle eighties, however, this work had dwindled considerably. More recently, it has become an active area of research once again, this time focusing mainly on automated or semi-automated synthesis. In this paper, we give a brief survey of the work to date in logic program synthesis.

Logic program synthesis is concerned with the following:

> Given a (nonexecutable) specification, how do we get an (executable) logic program satisfying the specification?

where the notions of "specification" and "executability" are interpreted broadly. For tackling this problem, logic programming provides a uniquely uniform framework because the specification, the synthesis process, and the resulting program can all be expressed in logic.

There are three main approaches to logic program synthesis by formal methods:

- In the constructive approach,[1] a conjecture based on the specification is constructively proved, and from this proof, the specified program is extracted. We call this approach *constructive synthesis*.
- A more direct approach is to deduce clauses for the specified program directly from the specification. We shall call this approach *deductive synthesis*.
- Another approach can induce a program from a partial specification of the program. The program is a generalization of the partial specification. We shall call this approach *inductive synthesis*.

This survey covers all the above approaches. Related issues such as correctness and verification, as well as synthesis by informal methods, will briefly be presented. This paper does not cover some other synthesis approaches, such as knowledge-based synthesis [109] or synthesis by inspection (see [7, 8]), because very little work has been done using these approaches in logic programming.

Thus, this survey covers only the *program synthesis* part of the so-called *program development* area. Program development is a much larger area since it also includes other aspects such as program analysis (e.g., abstract interpretation, termination) and program transformation (e.g., fold/unfold, partial evaluation), *inter alia* (see related papers in this Special Issue for other aspects of Program Development). Indeed, for the whole process of program development, we should consider the realistic scenario depicted by the spiral: (informal specification → incomplete formal specification → unsatisfactory program → better specification → more satisfactory program, and so on). This scenario shows the development process as a life-cycle, within which program synthesis enables a transition from specification to program. We will not deal with this entire life-cycle in this survey. Rather, we concentrate on synthesis methods. However, the reader should bear in mind the larger context of program development.

The distinction between deductive program synthesis and program transformation is rather subjective and often depends on the context of the research. A possible difference could be that synthesis starts from some nonexecutable specification, which usually means a nonrecursive description of the problem, while transformation usually starts from an already executable description. Program transformation will be covered by a separate survey in this volume. We shall not cover other related logic programming topics such as programming

[1] Also known as the *proofs-as-programs* approach, after [4].

environments, inductive logic programming, and semantics.

Our aim for this survey is to provide a broad, extensive (and hopefully complete) survey in the field of logic program synthesis. We shall give a coherent presentation using a unified framework of terminology and notation, and the same running example for all the approaches covered. By so doing, we shall attempt to expose the similarities and dissimilarities between the different approaches, and to evaluate their respective strengths and weaknesses.

Given the space limit, it is not possible to describe all the theoretical background necessary for a full treatment of the examples, or the theoretical difficulties and unsolved theoretical problems for each method. Nevertheless, a basic knowledge of logic programming will be sufficient to get a precise idea of the synthesis methods presented. However, the reader will need to read the original papers for a deeper understanding.

We hope this paper provides an assessment of existing work and a framework for future research in logic program synthesis.

The paper is organized as follows. In Section 2, we define and explain the basic concepts. In Sections 3, 4, 5, and 6, we present an account of constructive synthesis, deductive synthesis, inductive synthesis, and synthesis by informal methods, respectively. In each of these sections, we describe the approach, show the running example for this approach, and give a brief overview of existing methods using the approach. Finally, in Section 7, we conclude with an assessment of existing work and proffer our views on the way ahead.

2. BASIC CONCEPTS

In this section, we first define and describe basic concepts such as specifications and programs, and then we introduce commonly used notions of program correctness, in the framework of logic programming. Finally, we give an overview of correctness criteria and verification methods that have been proposed.

2.1. Specifications and Programs

A specification can be formal or informal; a program can be either a pure logic program or a program in an existing logic programming language such as Prolog. An informal specification describes what will be called the intended relation. This is the relation the programmer/specifier has in mind when synthesizing a program.

Definition 1. The *intended relation* for a predicate r (of arity n), denoted by $\mathcal{I}(r)$, is a set (of n-tuples) of ground terms.

Example 2. A pair $\langle l_1, l_2 \rangle$ belongs to $\mathcal{I}(included)$ iff l_1, l_2 are ground lists, and all the elements of l_1 belong to l_2.

The intended relation for a predicate r can also be seen as a specific (Herbrand) interpretation for the predicate r.

In logic programming, formal specifications are usually expressed in some logic. Such a specification will be called a logic specification.

Definition 3. A *logic specification* of a predicate r, denoted $Spec(r)$, is a set of logical formulas involving r.

In the above definition, the form of the logic formulas defining a logic specification is deliberately vague, to allow specifications by examples and incomplete specifications. The logic formulas of a logic specification $Spec(r)$ necessarily involve predicate r, but can also contain other predicates defined within the specification, or defined elsewhere (primitives or other specifications).

In Example 4, the predicate *member* is assumed to be a primitive such that $member(H, L)$ holds iff H is a member of list L. In practice, *member* can be specified in another specification, or together with the specification of *included*. The second logic specification in Example 4 is a "specification by examples."

Example 4.

$Spec_1(included) = \{ \quad included(L_1, L_2) \Longleftrightarrow \forall X(\ member(X, L_1) \Rightarrow member(X, L_2)\ ) \ \}$

$Spec_2(included) = \{ \quad included([\,], [2, 1])\,,\ included([1, 2], [1, 3, 2, 1]),\ \neg included([2, 1], [\,]) \ \}$

A (pure) logic program can be seen as a particular case of a logic specification, where the language is restricted to definite Horn clauses (also called program clauses). This is called a program because it also has a procedural or operational semantics (SLD-resolution). Sometimes, normal program clauses are used [80], allowing negations in the bodies of the clauses.

Definition 5. A *logic program* for a predicate r, denoted by $Prog(r)$, is a set of program clauses.

A logic program for a predicate r will normally contain clauses with the predicate r in the head. In Example 6, it is assumed that the predicate *member* and *remove_all* are primitives (with $remove\_all(H, L, NL)$ holding iff NL is the list L without all the occurrences of H). This restriction amounts to assuming that the subproblems involved have been, or will be, correctly implemented. They can thus be seen as primitives for $Prog(r)$. This simplification for programs (and for logic specifications) is made for ease of understanding, and can be overcome by simultaneously considering $Prog(r)$ (or $Spec(r)$) and its subproblems [33]. In the context of synthesis, such predicates are often included in a *background theory*.

Example 6.

$Prog_1(included) = \{ \quad included([\,], L_2) \leftarrow$
$included([H|T], L_2) \leftarrow member(H, L_2),\ included(T, L_2)\ \}$

$Prog_2(included) = \{ \quad included([\,], [\,]) \leftarrow$
$included(L_1, [H|T]) \leftarrow remove\_all(H, L_1, NL_1),\ remove\_all(H, T, NT_2),\ included(NL_1, NT_2)\ \}$

$Prog_3(included) = \{ \quad included(L_1, L_2) \leftarrow \neg q(L_1, L_2)$
$q(L_1, L_2) \leftarrow member(X, L_1),\ \neg member(X, L_2)\ \}$

Finally, an (executable) Prolog program is a program written in the Prolog language. The difference between Prolog programs and (pure) logic programs is that most Prolog programs contain control information and/or side-effect procedure calls that destroy the correspondence between the declarative semantics and the procedural one.

Most of the existing synthesis methods focus on the declarative aspects of logic programming, and thus usually produce logic programs rather than (executable) Prolog programs. This is not a weakness of the methods, but rather a deliberate separation of concerns. The transition from a correct logic program to an executable correct Prolog program usually amounts to the introduction of suitable control information [33]. We will not cover this issue in this survey.

2.2. *Correctness*

Correctness (and verification) is complementary to program synthesis. We have to verify that the synthesized program fulfills its specification, or more generally, that the synthesis method is correct (or sound), that is, always producing correct programs.

Correctness criteria relate the intended relation, the logic specification and the logic program, one to another. A logic specification and a logic program denote some relation, according to some semantics. We here introduce correctness criteria which are parametric with respect to the chosen underlying semantics.

Definition 7. Let $Spec(r)$ be a logic specification. The *meaning* of $Spec(r)$ is the set of ground terms[2]

$$\mathcal{S}(r) = \{ t \mid Spec(r) \models_s r(t) \}$$

where $\models_s$ denotes the chosen semantics for logic specifications.

Definition 8. Let $Prog(r)$ be a logic program. The *meaning* of $Prog(r)$ is the set of ground terms

$$\mathcal{P}(r) = \{ t \mid Prog(r) \models_p r(t) \}$$

where $\models_p$ denotes the chosen semantics for logic programs.

The semantics defines how a logic specification (program) should be interpreted. The meaning of a logic specification (program) defines how the predicate r should be interpreted, according to the specification (program) and the semantics.

The chosen semantics for logic specifications and logic programs also involve interpreting (primitive) predicates not defined in the specification or in the program—for instance, a possible approach is to add suitable formulas defining these predicates. An example of possible semantics for a logic specification could be its classical logical consequences. For a logic program, the semantics could be defined by its least Herbrand model, or by the set of models of the completed programs $Comp(P)$. We refer the reader to the survey on semantics for a detailed presentation of semantic issues.

[2] We assume an underlying Herbrand universe built out of the specification language.

2.2.1. LOGIC PROGRAMS VERSUS LOGIC SPECIFICATIONS. We can define three correctness criteria for logic programs with respect to logic specifications.

Definition 9. A logic program $Prog(r)$ is *partially correct* wrt a logic specification $Spec(r)$ iff $\mathcal{P}(r) \subseteq \mathcal{S}(r)$.

Partial correctness requires that the meaning of the program be included in the meaning of the specification. In other words, every answer computed by the program belongs to the specification.

Definition 10. A logic program $Prog(r)$ is *complete* wrt a logic specification $Spec(r)$ iff $\mathcal{P}(r) \supseteq \mathcal{S}(r)$.

Completeness is the converse of partial correctness. It requires that the meaning of the specification be included in the meaning of the program. It ensures that every specified answer is computed by the program.

Definition 11. A logic program $Prog(r)$ is *totally correct* wrt a logic specification $Spec(r)$ iff $\mathcal{P}(r) = \mathcal{S}(r)$.

Total correctness is thus the combination of partial correctness and completeness. Note that total correctness does not necessarily imply, from a procedural point of view, that all SLD-derivations for a given query are finite. By the completeness of SLD-resolution, each correct answer corresponds with at least one finite SLD-derivation.

2.2.2. LOGIC SPECIFICATIONS VERSUS INTENDED RELATIONS. The usual correctness criterion for logic specifications with respect to intended relations is the following:

Definition 12. A logic specification $Spec(r)$ is *totally correct* wrt an intended relation $\mathcal{I}(r)$ iff $\mathcal{S}(r) = \mathcal{I}(r)$.

Total correctness here can be decomposed into partial correctness (also called *consistency* here) ($\mathcal{S}(r) \subseteq \mathcal{I}(r)$) and completeness ($\mathcal{S}(r) \supseteq \mathcal{I}(r)$). In program synthesis, one usually assumes the consistency of the specification, but not its completeness. Some inductive synthesis methods explicitly consider incomplete specifications (e.g., specification by examples). For instance, $Spec_2(included)$ is consistent with $\mathcal{I}(included)$, but not complete.

2.2.3. LOGIC PROGRAMS VERSUS INTENDED RELATIONS. Similarly, the usual correctness criterion for logic programs with respect to intended relations is the following:

Definition 13. A logic program $Prog(r)$ is *totally correct* wrt an intended relation $\mathcal{I}(r)$ iff $\mathcal{P}(r) = \mathcal{I}(r)$.

Total correctness here can also be decomposed into partial correctness ($\mathcal{P}(r) \subseteq \mathcal{I}(r)$) and completeness ($\mathcal{P}(r) \supseteq \mathcal{I}(r)$). When the intended relation is informal, such a criterion cannot be established either formally or automatically. However, since the intended relation is a mathematical one, the correctness criterion can be established by mathematically

rigorous, although not formal, methods. Such correctness criteria can thus be used to guide the synthesis process, especially when incomplete logic specifications are considered.

Similarly, one could also define equivalence criteria between logic specifications, or between logic programs. Equivalence could then be decomposed into specialization and generalization.

When negations are allowed in the logic programs and in the queries, the correctness criteria (and the meaning of programs) are usually strengthened. For instance, besides having $\mathcal{P}(r) = \mathcal{S}(r)$, one may also require some equivalence between the negative atoms in the meaning of the program, and the negative atoms in the meaning of the specification.

Whereas verification of a program proves its correctness, *testing* a program can show its incorrectness. Testing a program can detect differences (if any) between the executable program and the specification. Tools for debugging can provide some help to locate where the problems are. Declarative debugging aims at detecting inconsistencies between the logic program (i.e., its declarative meaning) and the specification. See the survey on programming environments for a development of this subject.

2.3. *Overview of Correctness Literature*

Most of the literature on correctness could be described in terms of the above framework. Depending on the specification language, the form of the logic programs (with or without negations), and the chosen semantics, various approaches can be taken. For definite programs, the underlying semantics is usually the least Herbrand model (or something equivalent). We will not give a precise account of each piece of existing work. The reader is referred to the cited references for a detailed presentation.

A first formulation of correctness criteria appeared in [30]. This has been systematized and extended in [25] (see also [26, 29]). The programs considered here are definite programs. Partial correctness and completeness include the idea of preconditions on the input. A termination criterion is also proposed. Various verification methods are described: consequence verification, computational induction, and structural induction.

Hogger's work [57–59] is also based on definite programs. He proposes two families of criteria: the first one relating the declarative semantics of the logic programs to the logic specification, and the second one relating the procedural semantics of the logic program to the logic specification. He also describes verification methods based on transformation and derivation rules.

A similar approach is proposed in [5, 6], which discusses verification methods and program equivalence. Program equivalence is also treated in [89, 78, 79]. Notice that Lever considers programs with negations. In [49], equivalence captures some observational behavior (i.e., computation) of programs. Generalizations of programs, called extensions, are proposed in [102] in a framework for program development.

In [33], correctness criteria relate intended relations to logic programs with possible negations. A first set of criteria relates the intended relation to the declarative semantics of the (completion of the) logic program. A second set of criteria relates the intended relation to the sequence of computed answer substitutions of the Prolog programs (derived from the logic program). These criteria are used within a methodology for constructing programs, rather than for verifying programs. The methodology also addresses the problem of types in correctness criteria. Type problems in correctness criteria and in verification are also handled in [94, 95].

In [23, 34, 35], correctness criteria and verification methods based on proof trees are proposed for definite programs. Programs with negations are also treated in [37].

Other verification methods are proposed in [62, 67, 73, 69]. However, the specification language is here restricted.

In [2], a verification technique based on a Hoare-like inference rule is proposed for verifying the partial correctness and the completeness of a definite program. A similar approach is described in [28]. Another verification method, based on annotations and procedural semantics, is presented in [36].

Correctness criteria have also been proposed in the framework of Inductive Logic Programming (ILP). In this framework, the logic specification is a set of positive examples (denoted by $Spec^+(r)$), and possibly a set of negative examples (denoted by $Spec^-(r)$). Correctness criteria includes coverage of the positive examples by the logic program, what is also called consistency or completeness ($\mathcal{P}(r) \supseteq Spec^+(r)$), and consistency of the logic program with respect to the negative examples ($\overline{\mathcal{P}(r)} \subseteq Spec^-(r)$), where $\overline{\mathcal{P}(r)}$ is the complement of $\mathcal{P}(r)$. See [90] for further references.

For defining the meaning of logic programs with negations, semantics other than the usual completed program have been proposed, but not especially in the context of correctness: the perfect model semantics [101], the stable model semantics [48], the iterated least fixpoint model [1], the well-founded semantics [113], and others.

3. CONSTRUCTIVE SYNTHESIS

Constructive synthesis is an approach that originated in the functional programming paradigm,[3] and is also known as the *proofs-as-programs* approach [4]. It has been the basis of various existing program synthesis systems [93, 24, 53]. Over the past few years, constructive synthesis has also become an active area of research within logic programming [17, 42].

In functional programming, constructive synthesis is based on the Curry–Howard isomorphism in constructive type theory [60], which states that there is a one-to-one relationship between a constructive proof of an existence theorem and a program (i.e., a function) that computes witnesses of the existentially quantified variables of the theorem. That is, from a (constructive) proof of a formula of the form

$$\forall i.\, \exists o.\, r(i, o) \tag{1}$$

one can extract a program such that for all inputs i, it computes an output o that satisfies the specified relation r. Thus, the constructive synthesis process consists of two steps:

1. construct the formula (1) and prove it in a constructive logic;
2. extract from the proof a program for computing r.

It is worth emphasizing that the type theory is usually a higher-order (typed) logic, and the extracted program is a function.

[3] We use this term in a wide sense, to encompass any related work such as Kleene's work on realizable predicates and Martin-Löf's type theory, for instance.

3.1. *Description of Approach*

The original constructive synthesis approach based on the functional programming paradigm may be adapted for logic program synthesis, as has been done by Bundy and Wiggins [17, 114] for instance. A different constructive synthesis formulation for logic program synthesis, used by Fribourg [42], is based more directly on logic programming. However, to acknowledge its origin in functional programming, we shall present constructive logic program synthesis along the lines of the work of Bundy and Wiggins.

The key idea of Bundy and Wiggins' adaptation is that a predicate (in a typed logic) can be regarded as a truth-valued function, i.e., a predicate $p(X_1 : t_1, \ldots, X_n : t_n)$ can be regarded as a function of type $t_1 \times \cdots \times t_n \rightarrow boole$. This enables them to use the constructive synthesis approach to synthesize predicates as functions. To extract a (first-order) typed logic program from such a synthesis proof, they use a proof system based on a first-order (typed) logic with a set of rules specially devised[4] for constructing logic program fragments from the proof rules (see [116] for their definition and proof of their correctness). It is worth emphasizing that Bundy and Wiggins extract (first-order) typed, *pure* logic programs.

3.1.1. THE STARTING POINT. In the functional paradigm, constructive synthesis of a program to compute a relation r starts from a theorem of the form

$$\vdash \forall X_1 : t_1, \ldots, X_n : t_n \,.\, \exists Y_1 : t'_1, \ldots, Y_m : t'_m \,.\, r(X_1, \ldots, X_n, Y_1, \ldots, Y_m) \qquad (2)$$

where $X_1, \ldots, X_n$ are *input variables* of types $t_1, \ldots, t_n$ and $Y_1, \ldots, Y_m$ are *output variables* of types $t'_1, \ldots, t'_m$, respectively. This specification thus defines a relation between the input and output variables of the program.

This theorem is usually referred to as the *specification theorem*. However, it is more accurate to call it the *synthesis conjecture* since it is the start of the *synthesis* process and it has yet to be proved.

To adapt this synthesis conjecture for (typed) logic program synthesis, we encounter two problems due to the differences between functional and logic programs:

- Unlike a functional program, a logic program can be used in more than one way, i.e., in different input–output modes.
- Even for a chosen input–output mode, a logic program may produce many outputs or none at all.

A solution to these problems is to consider only the *all-ground* mode [17]. The relation r can then be seen as a Boolean-valued function. Thus, the synthesis conjecture (2) becomes

$$\vdash \forall X_1 : t_1, \ldots, X_n : t_n \,.\, \exists B : boole \,.\, r(X_1, \ldots, X_n) \hookrightarrow B \qquad (3)$$

where we have no output variables (as in (2)), but only input variables $X_1, \ldots, X_n$ (with types $t_1, \ldots, t_n$, respectively); $boole = \{true, false\}$.

The meaning of the operator $\hookrightarrow$ is defined by

$$\vdash Formula \hookrightarrow B \;\; iff \;\; \left\{ \begin{array}{l} \vdash \; Formula \Longleftrightarrow (B =_{boole} true) \\ \vdash \neg Formula \Longleftrightarrow (B =_{boole} false) \end{array} \right\}$$

[4] After the Curry–Howard isomorphism.

Thus, the function to be synthesized from (3) will be a logic program that is a decision procedure for the predicate $r(X_1, \ldots, X_n)$.

Note that in this *all-ground mode* approach, after a logic program has been synthesized, it is necessary to verify that the program can be used in some specific mode. This is usual, and it amounts to separating the logic part from the procedural one. Such a verification of modes can be performed by using existing tools such as abstract interpretation.

3.1.2. THE END RESULT. Thus, a successful proof of the synthesis conjecture means that a logic program exists which can answer a *goal* of the form $r(X_1, \ldots, X_n)$. The (pre-defined) construction rules corresponding to the proof rules used in the steps of the proof allow us to extract such a logic program. The end result of constructive logic program synthesis is usually a typed, first-order, pure logic program.

3.1.3. THE SYNTHESIS PROCESS. Starting from a synthesis conjecture, the first step of constructive synthesis is to produce a proof of this conjecture, for instance, in a typed (first-order) constructive logic. Obviously, this is not a simple task in itself. It is usually carried out on a mechanized proof system (which embodies the typed constructive logic), and requires the use of very sophisticated (mechanized) development tools and proof assistants. For instance, Bundy and Wiggins [17, 114, 116] use a proof planner to guide and automate parts of their proofs carried out in a proof development system.

From the proof of the synthesis conjecture, a program can be extracted. One possible approach consists of regarding the proof itself as a program. This interpretive approach is possible if we have an operational semantics for such proofs. Usually, though, a logic program is mechanically extracted from the proof. This is possible because each proof rule used in the proof system has an associated construction rule which has been pre-defined and proved to be correct, allowing each step of the proof to generate the corresponding logic clause(s).

Programs extracted from the constructive proofs are totally correct, assuming that the proof system is sound, and that the construction rules associated with the proof system are correct.

3.2. *Example*

For a logic program for $included(L_1, L_2)$, the synthesis conjecture is

$$\vdash \forall L_1 : lists,\ L_2 : lists\ .\ \exists B : boole\ .\ included(L_1, L_2) \hookrightarrow B$$

or, using the logic specification $Spec_1(included)$ in Example 4 (Section 2.1),

$$\vdash \forall L_1 : lists,\ L_2 : lists\ .\ \exists B : boole\ .\ \forall X\ .\ (member(X, L_1) \Rightarrow member(X, L_2)) \hookrightarrow B.$$

Applying induction on L_1 gives two subconjectures:

- (base case)

$$\vdash \forall L_2 : lists\ .\ \exists B : boole\ .\ \forall X\ .\ (member(X, [\,]) \Rightarrow member(X, L_2)) \hookrightarrow B \qquad (4)$$

• (step case)

$$\begin{array}{l} T : \mathit{lists} \\ \forall L_2 : \mathit{lists} \,.\, \exists B : \mathit{boole} \,.\, \forall X \,.\, (\mathit{member}(X, T) \Rightarrow \mathit{member}(X, L_2)) \hookrightarrow B \\ \vdash \forall L_2 : \mathit{lists} \,.\, \exists B : \mathit{boole} \,.\, \forall X \,.\, (\mathit{member}(X, [H|T]) \Rightarrow \mathit{member}(X, L_2)) \hookrightarrow B \end{array} \quad (5)$$

The program fragment that can be extracted at this point of the proof is

$$\begin{array}{l} \mathit{included}(L_1, L_2) \leftarrow L_1 = [], \ldots \\ \mathit{included}(L_1, L_2) \leftarrow L_1 = [H|T], \ldots \end{array} \quad (6)$$

Since the base case is always true, the first "..." can be replaced by *true*.

By using the definition of *member*, the step case gives rise to two more conjectures:

$$\vdash \exists B : \mathit{boole} \,.\, \forall X \,.\, (X = H \Rightarrow \mathit{member}(X, L_2)) \hookrightarrow B \quad (7)$$

$$\vdash \exists B : \mathit{boole} \,.\, \forall X \,.\, (\mathit{member}(X, T) \Rightarrow \mathit{member}(X, L_2)) \hookrightarrow B \quad (8)$$

(7) can be proved by a further application of induction, and (8) can be proved using the induction hypothesis

$$\forall L_2 : \mathit{lists} \,.\, \exists B : \mathit{boole} \,.\, \forall X \,.\, (\mathit{member}(X, T) \Rightarrow \mathit{member}(X, L_2)) \hookrightarrow B$$

in the step case.

The second "..." in (6) can now be replaced by

$$\mathit{member}(H, L_2) \wedge \mathit{included}(T, L_2)$$

and so the complete program extracted from the proof is

$$\begin{array}{l} \mathit{included}(L_1, L_2) \leftarrow L_1 = [] \\ \mathit{included}(L_1, L_2) \leftarrow L_1 = [H|T], \mathit{member}(H, L_2), \mathit{included}(T, L_2) \end{array}$$

The synthesized logic program is thus $Prog_3(included)$ from Example 6. It is worth noting that a proof by induction on L_2 of the synthesis conjecture would lead to $Prog_2(included)$ [88].

3.3. Overview of Methods

Our description of constructive logic program synthesis is based on the work of Bundy and Wiggins. They use a proof development system called *Whelk*, which is based on a Gentzen sequent calculus and a first-order typed constructive logic. *Whelk* has been implemented in a proof development environment called *Mollusc*. The precise notation and the details of their proof system and proof planning techniques can be found in [17, 20, 14, 22, 16, 21, 114, 116]. It is worth noting that they synthesize programs either in Prolog, or the new logic programming language Gödel [56]. An analysis of modes for the synthesized programs is made in [115].

In contrast, Fribourg [42, 44] uses a constructive approach based directly on logic programming. His method starts from a set P of logic procedures for pre-defined predicates

and a goal G of the form[5]

$$\forall X \,.\, \exists Y \,.\, q(X, Y) \Leftarrow r(X)$$

where $q(X, Y)$ and $r(X)$ are conjunctions of atoms defined in P. The variables in X and Y are regarded as *input* and *output* variables, respectively. A proof of this goal will define a procedure (for a new predicate) for computing Y from X. Fribourg performs the proof using the *extended execution system* of Kanamori and Seki [73] on the pre-defined predicates. The extended execution system is the standard Prolog interpreter with an extended form of SLD-resolution and a restricted form of structural induction. Each inference rule applied during the proof yields a corresponding logic procedure, thus enabling a logic program to be extracted for the new predicate on completion of the proof.

Unusually, Fribourg's method synthesizes programs that are guaranteed not only to be (partially) correct with respect to the specification, but also to terminate.[6]Moreover, tail-recursive programs can also be synthesized, although termination is not guaranteed for such programs. To help automate proofs, his method makes use of simplification lemmas [43].

In [41], constructive synthesis techniques are used to deductively add atoms to a logic program so that some correctness criteria with respect to a set of given logic properties are satisfied.

4. DEDUCTIVE SYNTHESIS

Deductive synthesis starts from a specification, and derives or deduces a logic program according to some pre-defined deduction rules. If the specification is a set of logic sentences, then the synthesis process consists of deducing program clauses directly from the specification. In this case, we can exploit fully the uniquely uniform framework provided by logic programming for program synthesis. Verification of partial correctness of the synthesized program reduces to showing that the deduction rules used are sound with respect to the underlying specification semantics. The resulting logic program will then be a logical consequence of the logic specification. It is therefore hardly surprising that almost all existing deductive synthesis methods fall into this category. In general, however, synthesis may have to be done using more sophisticated deduction strategies (possibly involving theorem proving) which will ensure that the logic program synthesized will be correct with respect to the specification.

4.1. Description of Approach

To give a general description of (first-order) deductive synthesis of logic programs, we follow a formalization along the lines of [81, 82].

4.1.1. THE STARTING POINT. The starting point for deductive synthesis is a pair $\langle \mathcal{M}, Q \rangle$ where

1. $\mathcal{M}$ is a set of axioms (in some first-order language), containing the logic specification $Spec(r)$. This specification is a predicate defined by means of a *definition axiom*,

[5]Fribourg calls such a goal an *implicative goal*.

[6]He considers a notion of *existential* termination.

i.e., an *iff*-formula (in the language of $\mathcal{M}$) whose head is the defined predicate. Other auxiliary predicates can also be defined in this way.

2. Q is an instance (or a set of instances)[7] of the specified relation r for which we want a program. Q thus represents a *query*.

$\mathcal{M}$ provides a general mathematical framework in which we can specify a large class of programs (or problems). For example, $\mathcal{M}$ may contain a theory of *lists* (complete with induction schemas, for instance).

The meaning of the pair $\langle \mathcal{M}, Q \rangle$ is the set of atoms that are instances of Q (and hence of r) and that logically follow from $\mathcal{M}$, according to the underlying specification semantics.

4.1.2. THE END RESULT. To deduce a logic program only for the query specified by Q, a small subset of the initial axiomatization $\mathcal{M}$ suffices in general. The synthesis process tries to derive (step by step) such a subset in the form of a set of definite or normal clauses (i.e., a logic program P), in such a way that SLD or SLDNF (instead of full first-order logic) can be used on these clauses to compute the answers to Q in an efficient way. In other words, the synthesis process derives (using some methods) the program P in such a way that the set of atoms that are instances of Q and that logically follow from $\mathcal{M}$ (under the specification semantics) is equivalent to the set of atoms which logically follow from the completion of P, $Comp(P)$ (under the program semantics). The synthesized program P is thus totally correct with respect to its logic specification $Spec(r)$ for the query Q considered.

4.1.3. THE SYNTHESIS PROCESS. Starting from the pair $\langle \mathcal{M}, Q \rangle$, a typical deductive synthesis method performs a synthesis process that can be formalized as a sequence of the form

$$\langle \mathcal{M} \cup D_0 \cup Comp(P_0), Q \rangle \Rightarrow \langle \mathcal{M} \cup D_1 \cup Comp(P_1), Q \rangle \Rightarrow \cdots \Rightarrow \langle \mathcal{M} \cup D_n \cup Comp(P_n), Q \rangle$$

where $D_0 \subseteq \cdots \subseteq D_n$ are sets of definition axioms for defining (new) predicates, $D_0 = \{\ \}$; $P_0 \subseteq \cdots \subseteq P_n$ are logic programs, $P_0 = \{\ \}$, such that

$$\mathcal{M} \cup D_k \cup Comp(P_k) \models_s Comp(P_{k+1}), \quad \text{for } 0 \leq k < n\ , \tag{9}$$

where $\models_s$ denotes the underlying specification semantics.

Condition (9) ensures that every program P_k of the sequence is *partially correct* with respect to $Spec(r)$.

Each step of the synthesis process thus consists of adding either a definition axiom or a program clause that has been derived. The logic programs $P_1, \ldots, P_n$ are thus derived incrementally, clause by clause, so that

$$P_1 \subseteq \cdots \subseteq P_n\ .$$

Any chosen method for deriving the clauses will guarantee partial correctness as long as it satisfies (9).

Total correctness, however, is a much more complex issue. In the formalization in [81, 82], for example, an alternative to the standard completion of P is considered. This

[7] Not necessarily ground instances.

yields a criterion for determining when the synthesized program is totally correct (and hence when to stop the synthesis process).

Finally, it is worth pointing out that, in general, this description of deductive synthesis according to [81, 82] does not apply to partial deduction (partially evaluating a logic program), although at first sight it may appear to do so. The distinction, formalized in [83], is that partial deduction derives $\langle Comp(P_2), Q\rangle$ from $\langle Comp(P_1), Q\rangle$, where P_1 and P_2 are logic programs and Q is the chosen goal, whereas deductive synthesis derives $\langle Comp(P), Q\rangle$ from $\langle \mathcal{M}, Q\rangle$ where $Comp(P)$ (or the completion of any program in general) is only a small subsystem of $\mathcal{M}$ (or a *specification framework* [83] in general).

Similarly, deductive synthesis can be distinguished from program transformation (based on unfold or fold rules, for instance).

4.2. Example

Now, we show an example of a typical deductive synthesis process, where first-order logic and SLD provide the specification and program semantics, respectively.

Suppose we have a theory of lists, S_{list}, and we want to synthesize a program for the query $included(L_1, L_2)$ from the logic specification $Spec_1(included)$ in Example 4 (Section 2.1). Then, the starting point will be

$$\langle S_{list} \cup D_{member} \cup D_{included}, included(L_1, L_2)\rangle$$

where D_{member} is the following definition axiom for *member*

$$member(X, L) \Longleftrightarrow L = [H|T] \wedge X = H \vee member(X, T) \tag{10}$$

and $D_{included}$ is, of course, just $Spec_1(included)$.

The condition for partial correctness (9) here is

$$S_{list} \cup D_{member} \cup D_{included} \cup D_k \cup Comp(P_k) \models Comp(P_{k+1}), \quad \text{for } 0 \leq k < n\ , \tag{11}$$

where $\models$ denotes first-order consequence.

To synthesize a clause for *included*, we deduce it directly from $Spec_1(included)$

$$included(L_1, L_2) \Longleftrightarrow \forall X.\, (member(X, L_1) \Rightarrow member(X, L_2))$$

as follows. Since we have

$$\begin{aligned} included([H|T], L_2) &\Longleftrightarrow \forall X.\, (member(X, [H|T]) \Rightarrow member(X, L_2)) \\ &\Longleftrightarrow \forall X.\, (\neg member(X, [H|T]) \vee member(X, L_2)) \end{aligned}$$

we can deduce

$$included([H|T], L_2) \Longleftrightarrow \forall X.\, ((\neg X = H \wedge \neg member(X, T)) \vee member(X, L_2))$$

from D_{member} (10).

Applying the distributivity law to the right-hand-side, we get

$$\begin{aligned} included([H|T], L_2) &\Longleftrightarrow \forall X.\, ((\neg X = H \vee member(X, L_2)) \wedge \\ &\qquad (\neg member(X, T) \vee member(X, L_2))) \\ &\Longleftrightarrow \forall X.\, ((X = H \Rightarrow member(X, L_2)) \wedge \\ &\qquad (member(X, T) \Rightarrow member(X, L_2))) \end{aligned}$$

which gives

$$included([H|T], L_2) \Longleftrightarrow member(H, L_2) \wedge included(T, L_2) \tag{12}$$

Thus, we have deduced (12) from the axioms in

$$S_{list} \cup D_{member} \cup D_{included}$$

and so we can use its *if*-part, namely, the clause

$$included([H|T], L_2) \leftarrow member(H, L_2), included(T, L_2)$$

and put it in P_1. That is, we have performed the synthesis step

$$\begin{aligned} &\langle S_{list} \cup D_{member} \cup D_{included}, included(L_1, L_2)\rangle \\ \Rightarrow\ &\langle S_{list} \cup D_{member} \cup D_{included} \cup Comp(P_1), included(L_1, L_2)\rangle \end{aligned}$$

in such a way that (9) is satisfied, that is,

$$S_{list} \cup D_{member} \cup D_{included} \models Comp(P_1) \tag{13}$$

thus ensuring the partial correctness of

$$P_1 = \{included([H|T], L_2) \leftarrow member(H, L_2), included(T, L_2)\}\ .$$

Similarly, we could also derive the clauses

$$\begin{aligned} included([], []) &\leftarrow \\ included([H], [H]) &\leftarrow \\ included([], L_2) &\leftarrow \end{aligned}$$

from

$$\langle S_{list} \cup D_{member} \cup D_{included} \cup Comp(P_1), included(L_1, L_2)\rangle$$

and add them to P_1 to get the final program. The partial correctness of this program is ensured by condition (9).

4.3. Overview of Methods

As mentioned earlier, most existing methods use (first-order) logic sentences for $\mathcal{M}$, and deduce logic clauses by correct inference rules directly from these logic sentences. That is, first-order logic and SLD, respectively, provide the underlying specification and program semantics.

Hansson and Tärnlund [61, 52], and Clark [30, 25, 26] axiomatize the relations that they wish to compute, as well as the data structures involved, as definition axioms in the form of *iff*-formulas in predicate logic. That is, their $\mathcal{M}$ is a set of *iff*-formulas, and their Q is a single atomic query defined (or definable) in terms of atoms already defined in $\mathcal{M}$. They then deduce logic programs from the axiomatization either by logical deduction (natural deduction in the case of [61, 52]) or by symbolic execution (or re-writing) of Q together with other rules for simplifying formulas such as equivalence substitutions (as in the case of [30, 25, 26]).

These methods guarantee partial correctness, but they require proofs of total correctness.

Hogger [57–59] also starts with a set of *iff*-formulas for $\mathcal{M}$ and a single iff formula for *Spec*(r). He then treats the *if*-part without the head as a goal which is to be solved by a resolution-like mechanism using a "logic procedure" consisting of the *only-if* part of *Spec*(r), as well as other relevant "clauses" from $\mathcal{M}$ which are necessary for solving this goal. He calls this *goal-directed derivation.* Alternatively, he also carries out the derivation using nonresolution inference rules. Hogger's method also guarantees partial correctness, but it requires proofs of total correctness.

Similar methods to the above have been proposed that are based on standard techniques for logic program transformation, for example, by Kanamori and Horiuchi [68].

In contrast, the method of Lau and Prestwich [84, 85] is designed to be mechanizable. Here, *Spec*(r) is also an *iff*-formula and $\mathcal{M}$ is a set of *iff*-formulas. However, the deduction of the clause(s) for solving Q is automatically decomposed into subdeductions which, when completed, are automatically composed into their parent deductions. This automation is possible because the user has to specify the recursion pattern in the required procedure. This method also guarantees partial correctness. Lau and Ornaghi [83] have proposed a method for synthesizing totally correct programs using their formalization of deduction synthesis in terms of SLDNF in [81, 82].

For specifications expressed by restricted classes of first-order logic formulas, it is possible to synthesize totally correct programs automatically. Such methods have been proposed by Dayantis [31], Sato and Tamaki [111] who have implemented a compiler for translating a class of first-order formulas directly into logic programs, and Kawamura [63, 64].

Finally, there are some methods which may not at first sight seem to fall into this category. Starting from the work of Bundy and Wiggins (see previous section), Kraan [65, 66] developed a method for program synthesis that is based on proof planning. In planning the proof that a (not yet synthesized) program meets its given specification, the program's body is represented by a meta-variable. The proof plan is completed by instantiating this meta-variable to logical formulas deduced from the specification.

In the LOPS synthesis system [9], the specification is really an "input–output" synthesis conjecture (as in constructive synthesis), and the specified program is derived by (various strategies for) re-writing formulas as well as using domain knowledge to generate relevant theorems. However, we may regard the domain knowledge as $\mathcal{M}$, and Neugebauer [96, 97] has shown that LOPS can be re-cast as a deductive synthesis method for logic programs (as well as programs in other target languages, even C!).

At this point, it is worth noting that Kreitz [70, 71] has studied program synthesis at a meta-level, and has shown that the constructive and deductive approaches are fully equivalent.

5. INDUCTIVE SYNTHESIS

Inductive synthesis refers to the process of formulating general rules from incomplete information, such as examples. Inductive synthesis of programs is performed by means of inductive inference, and is part of machine learning, a branch of AI. Inductive inference is related to the concept of generalization (deductive synthesis is related to specialization) and has received much attention in functional programming during the 1970s. It has been an active area of research in logic programming since the early 1980s.

We shall first give a more precise description of inductive synthesis and show that inductive synthesis of recursive logic programs has a specific niche within Inductive Logic

Programming (ILP). As an example of inductive synthesis, we shall then briefly present the Model Inference System [105, 106], before overviewing other existing approaches.

This section will not cover the entire Inductive Logic Programming area. Focus will be put on methods aiming at solving "programming problems" (rather than at concept learning), that is, problems where some recursion has to be synthesized. See [90, 92] for a complete survey and references on ILP.

5.1. Description of Approach

In the specific framework of (recursive) program synthesis from examples, it will also be assumed here that the specifier/programmer "knows" (even if only informally) the intended relation $\mathcal{I}(r)$. He is thus able to decide whether a given example belongs to the intended relation or not.

In an inductive synthesis of logic programs, the logic specification $Spec(r)$ is usually a set of positive examples (denoted by $Spec^+(r)$), and possibly a set of negative examples (denoted by $Spec^-(r)$). Examples are ground atoms. In some methods, the specification can be constructed incrementally during the synthesis process.

The assumption that the specifier "knows" the intended relation is formalized by assuming the consistency of the logic specification with respect to the intended relation. More precisely:

$$Spec^+(r) \subseteq \mathcal{I}(r)$$
$$Spec^-(r) \subseteq \overline{\mathcal{I}(r)}$$

where $\overline{\mathcal{I}(r)}$ denotes the complement of $\mathcal{I}(r)$. It is clear that a specification by examples is usually intrinsically incomplete (i.e., $Spec^+(r) \neq \mathcal{I}(r)$).

The objective of inductive synthesis is to infer a logic program $Prog(r)$ that covers at least all the examples: $Prog(r)$ must be consistent with respect to $Spec^+(r)$ (i.e., $\mathcal{P}(r) \supseteq Spec^+(r)$) and with respect to $Spec^-(r)$ (i.e., $\overline{\mathcal{P}(r)} \subseteq Spec^-(r)$). Given the incompleteness of the specification, the synthesized program must also cover other unspecified examples. Partial correctness with respect to the logic specification is thus irrelevant here. The objective is to get a program that is totally correct with respect to the intended relation, although such an objective cannot always be achieved in a fully automatic way. Inductive synthesis thus aims at inferring some "natural" extension of the given examples.

Within the methods for inductive program synthesis, one can distinguish between the trace-based approach and the model-based approach. In the trace-based approach, example traces are first generated. A trace is a sequence of instructions executed by an unknown program on some given input data. Then the traces are generalized into a program. This program may be obtained by folding, matching, and generalizing the traces. Generalization is required since traces are related to some specific inputs; folding is required in order to form loops and recursion. In the model-based approach, synthesis aims at constructing a finite axiomatization of a model of the examples. It thus makes an intensional representation of a relation (i.e., a program) from the given (incomplete) extensional representation (i.e., the examples).

The model-based approach to inductive synthesis of logic program is better known as Inductive Logic Programming (ILP). ILP is at the intersection of empirical (inductive) learning and logic programming [91]. By empirical learning, we mean the elaboration of a concept description from incomplete definitions. However, we concentrate here on a specific class of logic programs, namely, the recursive ones. In this specific case, we

assume that a human specifier knows the intended relation. This underlines the algorithmic focus of inductive synthesis compared to the more general scope of ILP (which also covers concept learning).

5.2. *Model Inference System*

One of the first systems for synthesizing logic programs from examples is Shapiro's Model Inference System (MIS) [105, 106]. It can also be seen as a special case of program debugging [106], where the initial program is empty. MIS is model-based. It is also incremental in the sense that examples are introduced one by one. For each new example, the program induced from the previous examples is updated to correctly handle this new example. A key feature of MIS is the clause generator, which has the capacity of "enumerating" possible program clauses according to some subsumption relation computed by a refinement operator. Such an enumeration is actually performed by searching the refinement graph induced by the chosen refinement operator.

The general strategy behind a synthesis with MIS is the following. The initial program is empty. For each new example, if it is a positive example that is not covered by the program, a new clause covering this example is added to the program. If the new example is a negative example that is covered by the program, then the covering clause is removed. If the resulting program is inconsistent with respect to the previous examples, the program is modified, using the above strategy. The resulting new program is then proposed to the user. The generated programs are always consistent with respect to all the introduced examples.

Example 14. Let us sketch a possible dialogue between the specifier and MIS to synthesize the *included* relation.

(Type) $included(list, list)$
(Mode) $included(+, +), determinate$
(Possibly used predicates) $member(\_, \_)$, $included(\_, \_)$
(E1) $included([], [1, 2])$

(P1) $included(L_1, L_2) \leftarrow$

(E2) $\neg included([1], [2])$

(P2) $included([], L_2) \leftarrow$

(E3) $included([1], [2, 1])$

(P3) $included([], L_2) \leftarrow$
$included([H|T_1], L_2) \leftarrow member(H, L_2)$

(E4) $\neg included([1, 2], [1, 3])$

(P4) $included([], L_2) \leftarrow$
$included([H|T_1], L_2) \leftarrow member(H, L_2),$
$included(T_1, L_2)$

The specifier must first declare the predicate to be synthesized, its type, mode, as well as the possible predicates used by the program. The declaration of the possibly used predicates is necessary for the system to limit the size of the refinement graph. This will only contain clauses involving the *included* or *member* predicates. After the presentation of example (E1), the synthesized program is (P1), the most general clause for *included*. Example (E2) forces the system to review this choice, and to take something less general. With example (E3), the program must be generalized to cover the new example. A new clause is chosen by the clause generator. It is as general as possible, while yielding a program consistent with

the previous examples. The presentation of example (E4) forces the system to reconsider the second program clause, and the clause generator produces a less general one.

5.3. *Overview of Methods*

Fundamental notions for inductive synthesis are subsumption and generalization, as developed by [99, 100, 103]. Plotkin's idea of least general generalization has been the basis of most model-based approaches to the induction of logic programs. Generalization can be used in two different ways, bottom-up or top-down. In a bottom-up approach, the example $included([], [1, 2])$ would yield the bottom element (least general) among the generalizations of the examples, that is, the program clause $included([], [1, 2]) \leftarrow$. In a top-down approach, such as in MIS, the clauses are enumerated from the most general to the most specific. The example $included([], [1, 2])$ would yield the top elements (most general) among the generalization of the example, that is, the program clause $included(L_1, L_2) \leftarrow$.

Top-down approaches as well as extensions and improvements of MIS have received much attention in the ILP framework. A complete account of this work can be found in [90].

Among the possible improvements of the MIS method, we mention the definition of more sophisticated refinement operators for the clause generator, the introduction of background knowledge, and predicate invention for the used predicates.

The combination of MIS and program schemata allows a further organization of the search space [112, 75]. This approach is especially adaptable to our specific case where recursive programs have to be synthesized since recursive programs can often be classified according to their design strategy (see Section 6.2).

The trace-based approach to program synthesis has received much attention in the context of functional programming (see the survey [107]). In the logic programming context, the trace-based approach has been reformulated in [51] by means of higher-order unification in a type theory with recursion. There, logic program synthesis from examples is actually also based on the constructive paradigm. A constructive proof for a concrete example of the theory is first generated, then the proof is generalized into an inductive proof from which a program can be extracted. In the context of program transformation, Compiling Control techniques (e.g., [15]), is also related to the trace-based approach.

Specifications by examples can also be extended by allowing examples and properties (i.e., logic formulas). In [38, 40, 39, 41], logic program synthesis is performed by instantiating a divide-and-conquer program schema. The specification is composed of examples and properties. The whole synthesis process combines inductive, deductive, and constructive synthesis. Different synthesis methods are used for instantiating the different place-holders of the program schema. One of the proposed methods, called Most Specific Generalization, aims at inductively inferring a logic program from examples, but within a restrictive setting. This method can successfully be applied to synthesize parts of a divide-and-conquer schema.

6. SYNTHESIS BY INFORMAL METHODS

Some of the methods that have been studied in the literature are informal in the sense that they start from an informal description of the intended relation. The primary objective of such methods is not necessarily the full automation of the synthesis process, but rather the elaboration of practical methods for the construction of logic programs. Usually, parts of

such construction processes are, or can be, automated, hence providing a computer-aided environment for the development of logic programs (see also the survey on Programming Environments).

Broadly speaking, there are two main informal approaches. The first one constructs a logic program by structural induction, starting from the intended relation. It is informal in the sense that the resulting logic program cannot be *formally* proven correct with respect to the intended relation. We will not consider here the direct construction of Prolog programs where the construction process is based on the operational semantics of Prolog. The second approach starts with a program schema and "instantiates" it to obtain a logic program. In logic programming, methods based on program schemata basically fall into the category of informal methods because logic specifications are usually absent.

6.1. *Program Construction by Structural Induction*

Structural induction [18, 19] is a major technique for the construction and the proof of correctness of programs. Basically, structural induction is a proof-by-induction method, where the induction is on the structural form of some terms. The construction of a program by structural induction is a construction where the reasoning is based on the structure of some input parameter. Such a construction implicitly contains a correctness proof by structural induction. Although initially introduced in the context of functional programming, it is also well-adapted for logic program construction. Structural induction is also used in constructive synthesis.

The construction of a logic program by structural induction can be seen as a framework allowing a precise presentation of the "natural" (manual) construction of a logic program, but based purely on declarative semantics.

To simplify notation, let us assume that we are dealing with a binary relation $r(X, Y)$. Given an intended relation $\mathcal{I}(r)$, the constructed logic program will have the following form:

$$\begin{array}{ll} r(X,Y) \leftarrow C_1 & \wedge F_1 \\ \quad\vdots & \quad\vdots \\ r(X,Y) \leftarrow C_n & \wedge F_n \end{array}$$

where, typically, each $C_i \wedge F_i$ will deal with one of the various cases of the induction parameter, with C_i determining a particular case of the induction parameter and the corresponding F_i verifying that the intended relation holds in this case. In practice, each C_i will often be a literal and each F_i a conjunction of literals (otherwise, a straightforward transformation can lead directly to the form of a logic program).

The construction process consists of the following:

1. Choice of an induction parameter (X or Y).
2. Choice of a well-founded relation[8] over the type of the induction parameter.
3. Construction of the structural forms C_i of the induction parameter.
4. Construction of the structural cases.

In the construction process, the predicate r as well as the other predicates involved are

[8] A relation $<$ is *well-founded* over a set E iff there is no infinite decreasing sequence $x_1 > x_2 \ldots > x_i > \ldots$ of elements of E.

interpreted according to their intended relations. The construction process is thus performed within some intended Herbrand interpretation $\mathcal{H}$.

The structural forms of the induction parameter must cover all the possible cases. More formally, the formula[9]

$$\forall X, Y \ . \ \exists(C_1 \vee \ldots \vee C_n)$$

must be true in the intended interpretation $\mathcal{H}$. The formula F_i should satisfy the condition that

$$\forall X, Y \ . \ \exists(C_i) \ \Rightarrow \ (\ r(X, Y) \Leftrightarrow \exists(C_i \wedge F_i) \ )$$

is true in the intended interpretation $\mathcal{H}$.

Such an F_i formula can be obtained by reduction to simpler subproblems (because of the particular form of the induction parameter) and/or by a recursive use of $r(s, t)$. It is, however, crucial to show that s (or t) is smaller than the induction parameter according to the chosen well-founded relation. The construction of the F_i is certainly one of the creative tasks.

One can show that under the hypothesis that the construction process has been correctly applied, the (completion of the) resulting logic program is totally correct with respect to the intended relation [33]. We also have that the intended interpretation $\mathcal{H}$ is a model of $Comp(P)$, and that the interpretation of r is the same (i.e., the intended relation $\mathcal{I}(r)$) in all the Herbrand models of $Comp(P)$. The choice of an induction parameter, a well-founded relation, and the structural forms are important since different choices can lead to different, although correct, logic programs.

The role of a well-founded relation is crucial to the correctness of the resulting program. Without a well-founded relation, (incorrect) programs of the form $r(X, Y) \leftarrow r(X, Y)$ could be constructed. From a procedural point of view, the well-founded relation also ensures the termination properties of the program when the induction parameter is ground in the query and in the recursive calls.

Example 15. Let us consider the intended relation $\mathcal{I}(included)$ specified in Example 2. In the intended interpretation $\mathcal{H}$, *included* is interpreted as $\mathcal{I}(included)$, $member(H, L)$ is true iff H is a member of the list L, and $X = Y$ is true iff Y and Y are syntactically identical. The construction proceeds as follows:

1. *Choice of an induction parameter:*
   We choose L_1 (arbitrarily).
2. *Choice of a well-founded relation (over lists):*
   Given two lists l_1, l_2, we define $l_1 < l_2$ iff l_1 is the tail of l_2.
3. *Construction of the structural forms of L_1:*
   The possible structural forms of L_1 are L_1 empty and L_1 nonempty. Hence, the two cases:

   - $L_1 = []$
   - $L_1 = [H|T]$

   These two forms covers all the possible forms because the following formula is true

[9]The subformula $\exists(F)$ in the formula $\forall X, Y \ldots \exists(F) \ldots$ denotes the existential closure of F, except for the variables X and Y.

in the intended interpretation $\mathcal{H}$:

$$\forall L_1 \,.\, \exists H, T. \,(\, L_1 = [] \vee L_1 = [H|T] \,)$$

4. *Construction of the structural cases:*
For each structural form, we have to find a necessary and sufficient condition to have $included(L_1, L_2)$ true in the intended relation.

   - For $L_1 = []$, the intended relation holds whatever the list L_2 is. The structural form is thus simply $true$. One can easily verify that the following formula is true in the intended interpretation $\mathcal{H}$:

     $$\forall L_1, L_2 \,.\, L_1 = [] \;\Rightarrow\; (included(L_1, L_2) \Leftrightarrow true)$$

   - For $L_1 = [H|T]$, a necessary and sufficient condition to have all the elements of L_1 belonging to L_2 is to have H belonging to L_2 and, all the elements of T belonging to L_2. That is,

     $$member(H, L_2) \wedge included(T, L_2)$$

     Notice that T is smaller than L_1 according to the well-founded relation.
     One could also easily verify that the following formula is true in the intended interpretation $\mathcal{H}$:

     $$\forall L_1, L_2 \,.\, (\, \exists H, T. \, L_2 = [H|T] \,) \Rightarrow (\, included(L_1, L_2) \Leftrightarrow L_2 = [H|T] \wedge member(H, L_2) \wedge included(T, L_2) \,)$$

The resulting program is then

$$\begin{array}{lll} included(L_1, L_2) \leftarrow & L_1 = [] & \\ included(L_1, L_2) \leftarrow & L_1 = [H|T] \,, & member(H, L_2) \,, \\ & & included(T, L_2) \end{array}$$

This program can be easily transformed into $Prog_1(included)$ given in Example 6. The alternative choice for the induction parameter, that is L_2, would lead to $Prog_2(included)$ in Example 6.

6.2. *Schema-Guided Program Construction*

Programs can be classified according to their design strategies (divide-and-conquer, generate-and, test, and so on). Informally, a program schema is a program template representing a whole family of particular programs, all based on the same design strategy. These programs can be obtained by instantiating the place-holders in the template to particular parameters or predicates. It is therefore interesting to guide the construction of a program by a schema capturing the essence of the chosen strategy.

Example 16 presents a (simplified) version of the divide-and-conquer schema, where the "divide" (i.e., induction) is performed on the second parameter.

Example 16.

$$
\begin{array}{lll}
r(X, Y) \leftarrow & Minimal(Y), & Solve(X, Y) \\
r(X, Y) \leftarrow & NonMinimal(Y), & Decompose(Y, FirstY, RestY), \\
 & & r(RestX, RestY), \\
 & & Process(FirstY, FirstX), \\
 & & Compose(FirstX, RestX, X)
\end{array}
$$

Various methods can be used (knowledge-based, schema composition, deductive/constructive/inductive synthesis, uses of algebraic properties of the specification, etc.), and they can be combined for instantiating the different parts of the schema. For example, a possible instantiation for the *included* problem could be the following:

$$
\begin{array}{lll}
included(L_1, L_2) \leftarrow & L_2 = [], & L_1 = [] \\
included(L_1, L_2) \leftarrow & L_2 = [H_2|T_2], & remove\_all(H_2, L_2, NL_2), \\
 & & included(NL_1, NL_2), \\
 & & H_2 = H_1, \\
 & & insert(H_1, NL_1, L_1)
\end{array}
$$

where $remove\_all(H, L, NL)$ holds iff NL is the list L without all the occurrences of H, and $insert(H, NL, L)$ holds iff L is the list NL where k occurrences (for some $k > 0$) of H have been added.

Given that NL_1 has no occurrence of H_1, the atoms $insert$ can be replaced by $remove\_all$, yielding $Prog_2(included)$ in Example 6.

6.3. Overview of Methods

Structural induction in logic program construction has already been seen in [30]. The construction of an axiomatic definition of a relation is performed by case analysis on the structural form of a parameter.

In Prolog textbooks, the usual guidelines for program construction are mainly based on a very procedural approach, and mostly disconnected with structural induction [27, 13, 47]. It should be noted that in [110], there is a clear distinction between the concept of a logic program and a Prolog program.

The above presentation of program construction by structural induction is based on [33] where methods are proposed for the systematic development of logic programs. These methods cover the whole process, starting from the intended relation, constructing a logic program, and deriving an executable Prolog program.

In functional programming, program schemata are used in deductive synthesis, such as in the KIDS system [108, 109], or in program transformation [55]. A formalization of a strategy deriving global search algorithms from specifications is described in [71]. Details can also be found in [72]. In the context of logic programming, schemata were mostly used for assisting the manual construction of logic programs.

In [45, 46], a hierarchy of logic program schemata is proposed. These are set in a second-order logic framework, and reflect a divide-and-conquer design strategy. Divide-and-conquer schemata are also proposed in [32, 33] which incorporate generalization techniques. The schemata are integrated in an environment for logic program development [54]. Various divide-and-conquer logic program schemata are carefully detailed in [41]. These

are used to guide inductive logic program synthesis.

Logic program schemata proposed in [98], cover different classes of problems and different design strategies.

Stepwise enhancement is proposed in [86, 87, 77, 76, 74] as a structured and procedural approach to Prolog program development. Program schemata, called "skeletons," isolate the basic control flow structures. Skeletons can be extended by means of "techniques" which can be applied to include extra computations in the skeletons. Different extensions can also be combined. [12] discusses a similar system based on what the authors call Clichés.

7. EVALUATION AND PERSPECTIVES

As we pointed out earlier, in this survey, we only intend to give a short introduction to the various synthesis methods. It would be folly to pretend that these methods on their own can tackle all the remaining problems or unsolved theoretical difficulties in logic program synthesis, let alone program synthesis in general! The synthesis of a program from a specification cannot be reduced to the choice of a method and the application of well-defined rules to synthesize a correct program. With this caveat, we now conclude with a brief summary, assessment, and discussion of existing work and potential future trends.

It is generally recognized that to achieve the goals of program synthesis, the best formalisms to use are declarative ones, such as functional and logic programming. The functional programming community has been very actively pursuing this objective, mainly doing constructive synthesis based on constructive type theory, and inductive synthesis from examples. In contrast, logic programmers have mainly concentrated on deductive synthesis. Each of these approaches has its own strengths and weaknesses.

In constructive synthesis, although program extraction can be mechanized, producing the proofs remains a nontrivial task and needs human interaction. In deductive synthesis, program extraction is unnecessary (in logic programming at any rate), and each deduction step can be automated, but the overall deduction strategy also needs human guidance.

Constructive and deductive synthesis are usually applied starting with a complete logic specification to begin with. A problem with these approaches [11] is that writing a logic specification for a program is sometimes very much like writing the program itself. A precise syntax has to be devised to completely codify the desired behavior, and one might prefer to write the program directly in this syntax rather than using automated synthesis systems. On the other hand, inductive synthesis from examples works very well. However, it can create programs automatically only if they are small (two or three or four lines of code), and the cost in execution time is exponential!

Synthesis by informal methods stresses what are the crucial creative steps within the design of a program. It also enables us to abstract programming concepts such as program schemas. As the starting specification is informal, these methods cannot be totally automated, but can yield tools supporting interactive program synthesis.

Logic programming provides a nice uniform framework for program synthesis. On the one hand, the specification, the synthesis, and the resulting program can all be expressed in logic. On the other hand, logic specifications can describe complete specifications as well as incomplete ones such as examples or properties of the relation to compute. The logic programming paradigm thus offers a chance to present both kinds of information within the same language, and treat them uniformly in a synthesis process.

Although presented separately in this paper, the different methods can be combined in various ways. Constructive and deductive synthesis do not have to start with complete

specifications. It is reasonable to believe that the key to a general synthesis method lies in a combination of the strengths of the different synthesis approaches. By studying these different approaches in the framework of logic programming, we hope we have taken a first step in the right direction.

Finally, in order to suggest or predict the future trends or directions of program synthesis, it is useful to return to the general context of program synthesis. If we view computer programming as a process of constructing executable code from (fragmentary) information, then program synthesis shares with automatic programming the same objective of using a machine to do computer programming. However, to paraphrase [104], it would be in vain to hope that, thanks to automatic synthesis, there will be no more programming. It is impossible to have user-oriented, general-purpose, and fully automatic programming systems. At least one of these three desirable qualities has to be sacrificed. The required input of such automatic systems needs to be carefully crafted, debugged, and maintained. Thus, some "programming" tasks will still have to be done. To quote [104],

> "Automatic programming systems of the future will be more like vacuum cleaners than like self-cleaning ovens."

Realistically, then, program synthesis aims at abstracting the programming process, letting the programmer concentrate on the really creative tasks involved. In this perspective, the synthesis system thus becomes a partner rather than an independent agent, and we have IA (Intelligence Amplification) rather than AI (Artificial Intelligence) [11]. Automatic programming will begin to have an impact on realistic programming by offering users tools for interactive synthesis, and not by delivering some ultimate solution.

We would like to thank the referees for their valuable, detailed comments, and for their constructive criticisms and suggestions, which have greatly improved this paper. We also thank Pierre Flener and Geraint Wiggins for reading the draft and for their helpful remarks.

REFERENCES

1. Apt, K. R., Blair, H., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases*, Morgan Kaufmann, 1988.
2. Balogh, K., On an Interactive Program Verifier for Prolog Programs, in: *Colloquia Mathematica Societatis Janos Bolyai 26, Mathematical Logic in Computer Science*, Salgotarjan, Hungary, 1978, pp. 111–142.
3. Balzer, R., A 15 Year Perspective on Automatic Programming, *IEEE Transactions on Software Engineering* 11(11):1257–1268 (Nov. 1985).
4. Bates, J. L., and Constable, R. L., Proofs as Programs, *ACM Transactions on Programming Languages and Systems* 7(1):113–136 (Jan. 1985).
5. Bossi, A., and Cocco, N., Verifying Correctness of Logic Programs, in: *Proceedings of TAPSOFT '89, LNCS 352*, Springer-Verlag, 1989, pp. 96–110.
6. Bossi, A., Cocco, N., and Dulli, S., A Method for Specializing Logic Programs, *ACM Transactions on Programming Languages and Systems* 12(2):253–302 (1990).
7. Barr, A., and Feigenbaum, E. A., (eds.), *The Handbook of Artificial Intelligence*, Morgan Kaufmann, 1982.
8. Biermann, A. W., Guiho, G., and Kodratoff, Y., (eds.) *Automatic Program Construction Techniques*, Macmillan, 1984.

9. Bibel, W., and Hörnig, K. M., LOPS—a System Based on a Strategical Approach to Program Synthesis, in: A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), *Automatic Program Construction Techniques*, Macmillan, 1984, ch. 3, pp. 69–89.
10. Biermann, A. W., Automatic Programming, in: S. C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, John Wiley, 2nd extended edition, 1992, pp. 59–83.
11. Biermann, A. W., Personal Communication, 1993.
12. Barker-Plummer, D., Cliché Programming in Prolog, in: M. Bruynooghe (ed.), *Proceedings of Meta'90*, 1990, pp. 47–256.
13. Bratko, I., *PROLOG Programming for Artificial Intelligence*, International Computer Science, Addison-Wesley, 1986.
14. Bundy, A., Smaill, A., and Hesketh, J., Turning Eureka Steps into Calculations in Automatic Program Synthesis, in: Clarke, S. L. H., (ed.), *Proceedings of UK IT 90*, 1990, pp. 221–226.
15. Bruynooghe, M., De Schreye, D., and Krekels, B., Compiling Control, *Journal of Logic Programming* 6(1-2):135–162 (1989).
16. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A., Rippling: A Heuristic for Guiding Inductive Proofs, Research Paper 567, DAI, 1991. Submitted to *Artificial Intelligence*.
17. Bundy, A., Smaill, A., and Wiggins, G., The Synthesis of Logic Programs from Inductive Proofs, in: J. W. Lloyd (ed.), *Proceedings of Esprit Symposium on Computational Logic*, Springer-Verlag, 1990, pp. 135–149.
18. Burstall, R. M., Proving Properties of Programs by Structural Induction, *The Computer Journal* 72:41–48 (1969).
19. Burstall, R. M., Program Proving as Hand Simulation with a Little Induction, in: *IFIP 74*, North-Holland, 1974, pp. 308–312.
20. Bundy, A., van Harmelen, F., Horn, C., and Smaill, A., The Oyster-Clam System, in: M. E. Stickel (ed.), *Proceedings of the 10th International Conference on Automated Deduction*, Springer-Verlag, 1990, pp. 647–648, Lecture Notes in Artificial Intelligence No. 449.
21. Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A., Experiments with Proofs Plans for Induction, *Journal of Automated Reasoning* 7:303–324 (1991).
22. Bundy, A., van Harmelen, F., Smaill, A., and Ireland. A., Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs, in: M. E. Stickel (ed.), *Proceedings of the 10th International Conference on Automated Deduction*, Springer-Verlag, 1990, pp. 132–146, Lecture Notes in Artificial Intelligence No. 449.
23. Courcelle, B., and Deransart, P., Proofs of Partial Correctness for Attribute Grammars with Applications to Recursive Procedures and Logic Programming, *Information and Computation* 78(1):1–55 (1988).
24. Constable, R. L., et al., *Implementing Mathematics with the NuPRL Proof Development System*, Prentice-Hall, 1986.
25. Clark, K. L., Predicate Logic as a Computational Formalism, Technical Report 79/59, Imperial College of Science and Technology, Univ. of London, 1979.
26. Clark, K. L., The Synthesis and Verification of Logic Programs, Technical Report DOC 81/36, Imperial College, Sept. 1981. (Revised version of a document which first appeared in 1977.)
27. Clocksin, W. F., and Mellish, C. S., *Programming in Prolog*, Springer-Verlag, New York, 2nd edition, 1984.
28. Colussi, L., and Marchiori, E., Proving Correctness of Logic Programs Using Axiomatic Semantics, in: K. Furukawa (ed.), *Proceedings of ICLP'91*, MIT Press, 1991, pp. 629–642.
29. Clark, K. L., and Sickel, S., Predicate Logic: A Calculus for Deriving Programs, in: *Proceedings of IJCAI-77*, 1977, pp. 419–420.
30. Clark, K. L., and Tärnlund, S.-Å., A First Order Theory of Data and Programs, in: *Proceedings of IFIP 77*, North-Holland, 1977, pp. 939–944.
31. Dayantis, G., Logic Program Derivation for a Class of First Order Logic Relations, in: *Proceedings of IJCAI-87*, 1987, pp. 9–14.

32. Deville, Y., and Burnay, J., Generalization and Program Schemata: A Step Towards Computer-Aided Construction of Logic Programs, in: *Proceedings of North American Conference on Logic Programming 1989*, MIT Press, 1989, pp. 409–425.
33. Deville, Y., *Logic Programming: Systematic Program Development*, International Series in Logic Programming. Addison-Wesley, 1990.
34. Deransart, P., and Ferrand, G., Logic Programming: Methodology and Teaching, in: K. Fuchi and L. Kott (eds.), *Proceedings of the French Japan Symposium*, Aug. 1988, pp. 133–147.
35. Deransart, P., and Ferrand, G., A Methodological View of Logic Programming with Negation, Technical Report RR 1011, INRIA, Apr. 1989.
36. Drabent, W., and Maluszynsky, J., Inductive Assertion Method for Logic Programs, in: *Proceedings of TAPSOFT '87, 2, LNCS 250*, Springer-Verlag, 1987, pp. 167–181.
37. Ferrand, G., and Deransart, P., Proof Methods of Partial Correctness and Weak Completeness for Normal Logic Programs, in: K. Apt (ed.), *Proceedings of JICSLP'92*, MIT Press, 1992, pp. 161–174.
38. Flener, P., and Deville, Y., Towards Stepwise, Schema-Guided Synthesis of Logic Programs, In T. P. Clement and K. K. Lau (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing, Springer-Verlag, 1992, pp. 46–64.
39. Flener, P., and Deville, Y., Synthesis of Composition and Discriminate Operators for Divide-and-Conquer Logic Programs, in: J.-M. Jacquet (ed.), *Constructing Logic Programs*, Wiley & Sons, 1993.
40. Flener, P., and Deville, Y., Logic Program Synthesis from Incomplete Specifications, *Journal of Symbolic Computation: Special Issue on Automatic Programming*, W. Bibel and A. W. Biermann (eds.), 1993 accepted for publication.
41. Flener, P., *Logic Algorithm Synthesis from Examples and Properties*, Ph.D. thesis, Unité d'Informatique, Université Catholique de Louvain, Belgium, 1993.
42. Fribourg, L., Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction, in: *Proceedings of 7th Int. Conference on Logic Programming*, MIT Press, 1990, pp. 685–699.
43. Fribourg, L., Automatic Generation of Simplification Lemmas for Inductive Proofs, in: V. Saraswat and K. Ueda (eds.), *Proceedings of 1991 International Logic Programming Symposium*, MIT Press, 1991, pp. 103–116.
44. Fribourg, L., Extracting Logic Programs that Use Extended Prolog Execution and Induction. in: J.-M. Jacquet (ed.), *Constructing Logic Programs*, Wiley & Sons, 1993.
45. Gegg-Harrison, T. S., Basic Prolog Schemata, Technical Report CS-1989-20, Dept. of Computer Science, Duke University, 1989.
46. Gegg-Harrison, T. S., A Scheme-Based Approach to Teaching Recursive Prolog Programming, Technical Report CS-1990-4, Dept. of Computer Science, Duke University, 1990.
47. Giannesini, F., Kanoui, H., Passero, R., and Van Caneghem, M., *Prolog*, Addison-Wesley, 1986.
48. Gelfond, M., and Lifschitz, V., The Stable Model Semantics for Logic Programming, in: B Kowalski and K. Bowen (eds.), *Proceedings of 1988 International Logic Programming Symposium*, MIT Press, 1988, pp. 1070–1080.
49. Gabrielli, M., Levi, G., and Meo, M. C., Observational Equivalences for Logic Programs, in: K. Apt (ed.), *Proceedings of JICSLP'92*, MIT Press, 1992, pp. 131–145.
50. Goldberg, A. T., Knowledge-Based Programming: A Survey of Program Design and Construction Techniques, *IEEE Transactions on Software Engineering* 12(7):752–768 (July 1986).
51. Hagiya, M., Programming by Example and Proving by Example Using Higher-Order Unification, in: M. E. Stickel (ed.), *Proceedings of CADE'90*, Springer-Verlag LNCS 449, 1990, pp. 588–602.

52. Hansson, Å., A Formal Development of Programs, Ph.D. thesis, Dept. of Information Processing and Computer Science and Computer Science, The Royal Institute of Technology and the University of Stockholm, 1980.
53. Hayashi, S., A System Extracting Programs from Proofs, in: *3rd Working Conference on the Formal Description of Programming Concepts*, Ebberup, Denmark, 1986.
54. Henrard, J., and Le Charlier, B., Folon: An Environment for Declarative Construction of Logic Programs, in: M. Bruynooghe and M. Wirsing (eds.), *Proceedings of PLILP'92*, Springer-Verlag LNCS 631, 1992, pp. 217–231.
55. Huet, G., and Lang, B., Proving and Applying Program Transformations Expressed with Second-Order Patterns, *Acta Informatica* 11:31–55 (1978).
56. Hill, P., and Lloyd, J., The Gödel Report, Technical Report TR-91-02, Department of Computer Science, University of Bristol, Mar. 1991.
57. Hogger, C. J., Program Synthesis in Predicate Logic, University of Hamburg in: *Proceedings of AISB/GI Conference on Artificial Intelligence*, 1978, pp. 22–27.
58. Hogger, C. J., Derivation of Logic Programs, *J. ACM* 28(2):372–392 (Apr. 1981).
59. Hogger, C. J., *Introduction to Logic Programming*, Academic Press, 1984.
60. Howard, W. A., The Formulae-as-Type Notion of Construction, in: J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry; Essays on Combinatory Logic, Lamda Calculus and Formalism*, Academic Press, 1980, pp. 479–490.
61. Hansson, Å., and Tärnlund, S.-Å., A Natural Programming Calculus, in: *Proceedings of IJCAI-79*, 1979, pp. 348–355.
62. Kanamori, T., Soundness and Completeness of Extended Execution for Proving Properties of Prolog Programs, Technical Report TR-175, ICOT, May 1986.
63. Kawamura, T., Derivation of Efficient Logic Programs by Synthesizing New Predicates, in: V. Saraswat and K. Ueda (eds.), *Proceedings of 1991 Int. Logic Programming Symposium*, MIT Press, 1991, pp. 611–612.
64. Kawamura, T., Logic Program Synthesis from First Order Logic Specifications, in: *Proceedings of Fifth Generation Computer Systems 92*, Tokyo, 1992, pp. 463–472, Ohmsha.
65. Kraan, I., Basin, D., and Bundy, A., Logic Program Synthesis via Proof Planning, in: K. K. Lau and T. Clement (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing, Springer-Verlag, 1993, pp. 1–14.
66. Kraan, I., Basin, D., and Bundy, A., Middle-Out Reasoning for Logic Program Synthesis, in: D. S. Warren (ed.), *Proceedings of 10th International Conference on Logic Programming*, MIT Press, 1993, pp. 441–455.
67. Kanamori, T., and Fujita, H., Formulation of Induction Formulas in Verification of Prolog Programs, in: J. H. Siekmann (ed.), *Proceeding of *th International Conference on Automated Deduction*, LNCS 225, Springer-Verlag, 1986, pp. 281–299.
68. Kanamori, T., and Horiuchi, K., Construction of Logic Programs Based on Generalized Unfold/Fold Rules, in: *Proceedings of 4th International Conference on Logic Programming*, Melbourne, 1987, pp. 744–768.
69. Kanamori, T., and Kawamura, T., Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation (ii), Technical Report TR-403, ICOT, June 1988.
70. Kreitz, C., Towards a Formal Theory of Program Construction, *Revue d'Intelligence Artificielle* 4(3):53–79 (1990).
71. Kreitz, C., Formal Mathematics for Verifiably Correct Program Synthesis, 1993, Forschungsbericht, Technische Hochschule Darmstadt, Germany.
72. Kreitz, C., Meta-synthesis — Deriving Programs that Develop Programs, 1993.
73. Kanamori, T., and Seki, H., Verification of Prolog Programs Using an Extension of Execution, in: E. Shapiro (ed.), *Proceedings of 3rd International Conference on Logic Programming*, Springer-Verlag, 1986, pp. 475–489. LNCS 255.
74. Kirschenbaum, M., and Sterling, L., Prolog Programming Using Skeletons and Techniques, Technical Report TR-90-109, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1990.

75. Kirschenbaum, M., and Sterling, L., Refinement Strategies for Inductive Learning of Simple Prolog Programs, in: *Proceedings of IJCAI'91*, 1991, pp. 757–761.
76. Lakhotia, A., *A Workshop for Developing Logic Programs by Stepwise Enhancement*, Ph.D. thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, 1989.
77. Lakhotia, A., Incorporating "Programming Techniques" into Prolog Programs, in: E. L. Lusk and R. A. Overbeek (eds.), *Proceedings of NACLP'89*, MIT Press, 1989, pp. 426–440.
78. Lever, J. M., Combining Induction with Resolution in Logic Programming, Technical Report, Ph.D. Thesis, Dept. of Computing, Imperial College, London, 1991.
79. Lever, J. M., Proving Program Properties by Means of SLS-Resolution, in: K. Furukawa (ed.), *Proceedings of ICLP'91*, MIT Press, 1991, pp. 614–628.
80. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, New York, 2nd edition, 1987.
81. Lau, K. K., and Ornaghi, M., Towards a Formal Framework for Deductive Synthesis of Logic Programs, Technical Report UMCS-92-11-2, Dept. of Computer Science, University of Manchester, Nov. 1992.
82. Lau, K. K., and Ornaghi, M., An Incompleteness Result for Deductive Synthesis of Logic Programs, in: D. S. Warren (ed.), *Proceedings of 10th International Conference on Logic Programming*, MIT Press, 1993, pp. 456–477.
83. Lau, K. K., and Ornaghi, M., A Formal View of Specification, Deductive Synthesis and Transformation of Logic Programs, in: Y. Deville (ed.), *Logic Program Synthesis and Transformation*, Workshops in Computing, Springer-Verlag, 1994, *Proceedings of Third International Workshop on Logic Program Synthesis and Transformation.*
84. Lau, K. K., and Prestwich, S. D., Top-Down Synthesis of Recursive Logic Procedures from First-Order Logic Specifications, in: D. H. D. Warren and P. Szeredi (eds.), *Proceedings of 7th International Conference on Logic Programming*, MIT Press, 1990, pp. 667–684.
85. Lau, K. K., and Prestwich, S. D., Synthesis of a Family of Recursive Sorting Procedures, in: V. Saraswat and K. Ueda (eds.), *Proceedings of 1991 International Logic Programming Symposium*, MIT Press, 1991, pp. 641–658.
86. Lakhotia, A., and Sterling, L., Composing Recursive Logic Programs with Clausal Join, *New Generation Computing* 6:211–225 (1988).
87. Lakhotia, A., and Sterling, L., Stepwise Enhencement: A Variant of Incremental Programming, in: *Proceecdings of Conference on Software Engineering*, 1990.
88. Lombart, V., Wiggins, G., and Deville, Y., Guiding Synthesis Proofs, in: *LOPSTR'93*, 1993, Extended Abstract.
89. Maher, M. J., Equivalences of Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 1988, pp. 627–658.
90. Muggleton, S., and De Raedt, L., Inductive Logic Programming: A Survey, *Journal of Logic Programming, Special Issue on "Ten Years of Logic Programming,"* 1993 (submitted).
91. Muggleton, S., Inductive Logic Programming, *New Generation Computing*, 8(4):295–317, 1991.
92. Muggleton, S., (ed.), *Inductive Logic Programming*, Volume APIC-38, Academic Press, 1992.
93. Manna, Z., and Waldinger, R., A Deductive Approach to Program Synthesis, *ACM Transactions on Programming Languages and Systems* 2(1):90–121, (Jan. 1980).
94. Naish, L., Specification = Program + Types, in: *Proceedings of FST & TCS*, Springer-Verlag, LNCS, 1987.
95. Naish, L., Verification of Logic Programs and Imperative Programs, in: J.-M. Jacquet (ed.), *Constructing Logic Programs*, John Wiley & Sons, 1993.
96. Neugebauer, G., The IO-Graph Method: Algorithm Design and Implementation, Submitted to the *Journal of Symbolic Computation*, 1992.
97. Neugebauer, G., The LOPS Approach: A Transformation Point of View, in: K. K. Lau and T. P. Clement (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing, Springer-Verlag, 1993, pp. 80–81.

98. O'Keefe, R. A., *The Craft of Prolog*, Logic Programming Series. MIT Press, 1990.
99. Plotkin, G. D., A Note on Inductive Generalization, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, Edinburgh University Press, Edinburgh (UK), 1970, pp. 153–163.
100. Plotkin, G. D., A Further Note on Inductive Generalization, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 6*, Edinburgh University Press, Edinburgh (UK), 1971, pp. 101–124.
101. Przymunsinski, T., Perfect Model Semantics, in: B Kowalski and K. Bowen (eds.), *Proceedings of 1988 International Logic Programming Symposium*, MIT Press, 1988, pp. 1081–1096.
102. Power, A. J., and Sterling, L., A Notion of Map Between Logic Programs, in: *Proceedings of 7th International Conference on Logic Programming*, MIT Press, 1990, pp. 390–404.
103. Reynolds, J. C., Transformational Systems and the Algebraic Structure of Atomic Formulas, in: B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, Edinburgh University Press, Edinburgh (UK), 1970, pp. 135–151.
104. Rich, C., and Waters, R. C., Automatic Programming: Myths and Prospects, *IEEE Computer* 21(8):40–51 (Aug. 1988).
105. Shapiro, E. Y., An Algorithm that Infers Theories from Facts, in: *Proceedings of IJCAI'81*, 1981, pp. 446–451.
106. Shapiro, E. Y., *Algorithmic Program Debugging*, MIT Press, 1983.
107. Smith, D. R., The Synthesis of LISP Programs from Examples, in: A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), *Automatic Program Construction Techniques*, Macmillan, 1984, ch. 15, pp. 307–324.
108. Smith, D. R., The Structure and Design of Global Search Algorithms, Technical Report KES.U.87.12, Kestrel Institute, Palo Alto, CA, 1988.
109. Smith, D. R., KIDS: A Semiautomatic Program Development System, *IEEE Transactions on Software Engineering* 16(9):1024–1043 (1990).
110. Sterling, L., and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.
111. Sato, T., and Tamaki, H., First Order Compiler: A Deterministic Logic Program Synthesis Algorithm, *J. Symbolic Computation* 8:605–627 (1989).
112. Tinkham, N. L., Induction of Schemata for Program Synthesis, Ph.D. thesis, Duke University, Durham, NC, 1990.
113. Van Gelder, A., Ross, K., and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, in: *Proceedings of 7th Symposium on Principles of Database Systems*, 1988, pp. 221–230.
114. Wiggins, G., Bundy, A., Kraan, I., and Hesketh, J., Synthesis and Transformation of Logic Programs from Constructive, Inductive Proof, in: T. P. Clement and K. K. Lau (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing, Springer-Verlag, 1992, pp. 27–45.
115. Wiggins, G. A., Negation and Control in Automatically Generated Logic Programs, in: A. Pettorossi (ed.), *Proceedings of META-92*, 1992.
116. Wiggins, G., Synthesis and Transformation of Logic Programs in the Whelk Proof Development System, in: K. Apt (ed.), *Proceedings of JICSLP'92*, MIT Press, 1992, pp. 351–365.