# Induction, Deduction, and Abduction for Program Design and Maintenance

Ahmed Guessoum
Jan Komorowski

Knowledge Systems Group
Department of Computer Systems and Telematics
The Norwegian Institute of Technology
7034 Trondheim, Norway
email: {komorowski}@idt.unit.no
phone: +47 73 594567 fax: +47 73 594466

**Extended Abstract**

We introduce IDA, an approach to the program design problem which combines methods from inductive learning (Induction), refinement of programs (Deduction) and update of knowledge bases (Abduction). The approach allows to synthesize (logic) specifications from examples, to refine the specifications into logic programs and to test and modify the designed programs. When the need for a modification of the program arises, the update procedures will produce transactions that select the relevant design decisions and the appropriate parts of the synthesized specification. The framework is implemented and experimental results confirm the suitability of our proposal.

**Keywords** Synthesis of Logic Programs, Refinement Calculus, Knowledge Base Update, Inductive Logic Programming.

## 1 Introduction

In program synthesis, whether by deductive synthesis (see e.g. [6]) or by using Refinement Calculus, [13], it seems to be important to *debug* and to *reuse* designs. By a design we understand here a sequence of operations applied to a given starting object such as a set of axioms including examples, or a logic program. Debugging a design is understood as being the process of identifying the examples or the design operations that are responsible for unwanted behaviour of the synthesized program. In some cases, the designer may want to

change some operations and/or their arguments and then re-synthesize the new program. This process is referred to as reuse of designs. By maintenance we will mean debugging or reusing. Such re-usability may contribute to a better productivity of program developers through a higher level of automation.

A combination of inductive and deductive synthesis was proposed in the past ([7, 5]). The novelty of our proposal, however, lies in providing a framework for integrating the debugging and reuse aspects within the program development paradigm. We accomplished that by using procedures that update logic programs [8, 9].

In the present work we aim at creating a framework in which it would be possible to reuse a significant amount of the effort that is put into the process of constructing and maintaining programs synthesized using our Refinement Calculus. We envisage an environment (IDA) where three reasoning paradigms—induction, deduction, and abduction—are put together to allow semi-automatic program development and maintenance. Induction is used to synthesize from examples a part of a specification; deduction, to refine problem-solving strategies and the partial specification into a program; and abduction, to correct the refined program in terms of corrections to the refinement process.

**Structure of the (full, forthcomming) paper** The IDA framework is presented in the next section. It gives a first idea of the principles formulated in our work. The main results from the Refinement Calculus and knowledge base updates are briefly summarized in the following section. In Section 4, we present our method for representing the design decisions as a logic program and explain how the latter can be fed into the update procedures to relate requested changes on the synthesized program to the initial specifications and/or to design decisions. In the following section, a detailed example is then given which shows how design modification works in IDA. The paper ends with a section on related work and then a concluding section.

It is assumed that the reader is acquainted with the logic programming paradigm and its terminology. (See, for instance, [15].) In the subsequent examples, we adopt a Prolog-like syntax, where a constant, function, or predicate symbol is a string that starts with a lower-case letter and a variable is a string that starts with a capital letter.

# 2 A Methodology

Within the logic programming paradigm, we have been investigating the possibility of providing a framework for the concept of reusable design of (logic) programs synthesized using the Refinement Calculus (RC) (for details see [13]).

In the RC approach one starts with a logic program and refines it using

predefined operators. The refinement operators order the programs in a partial order and preserve some pre-defined properties of the programs such as, for example, model inclusion. The initial programs can be rather general. For example, they can be definitions of problem solving techniques such as divide and conquer or generate and test, together with the definitions of dividing and composing or generating and testing. It should be noted that refinement is a highly non-deterministic process. Several choices of operations, data structures, program schemata, etc., are usually possible.

Before the refinement process can start definitions of, among other things, specifications of data structures need to be provided. This can be done, for instance, by inducing specifications from (positive and negative) examples. A number of systems exist in Inductive Logic Programming (ILP) to perform this task. (The interested reader can find in [14] a good survey and explanation of the characteristics of various ILP systems.) For our purposes we have chosen to use MIS [21] to generate the specifications.

Once the specifications have been generated, they are given as input to the Refinement Calculus module. This module refines the specifications into a logic program, while preserving correctness. It is accomplished by our REFINERY system, an implementation of the Refinement Calculus.

In order to debug the refined program, the Refinement module is augmented with the capability of constructing a *Refinement Trace* (RT), which is explained in Section 4.1. This represents, as a logic program, the information about the refinement operations that are performed by the refinement module to produce the refined program. The latter and the RT are given as input to the update module along with an update request to find out what needs to be modified in the design (i.e. specifications and/or refinement operations) when some unwanted program behaviour is noted or a different program property is required. Upon this, some part of the specifications is corrected and/or refinement is undone, before another refinement is attempted. This leads to a cycle of refinement and update, until the (new) satisfaction criteria are met.

Note that if the update module points out that some specifications need to be checked/corrected and this can be achieved (by the update module) in a way that is consistent with the original specifications, then the latter are updated and the result is given as input to the refinement module. Otherwise, the induction module is triggered in such a way that the information output by the update module is used to further specialize the specifications or, if need arises, to generate new specifications.

# 3 Logic Program Refinement and Update

Intuitively, program refinement (See references in [6]) is a succession of transformations of programs or specifications into other programs while preserving

required properties of the initial program (or specification).

In [13] we introduced a refinement calculus for deriving logic programs. The calculus has a model-theoretic characterization (based on model inclusion of the corresponding programs) and a transformational characterization. The transformational characterization is founded on PD [12] and is shown to be sufficient to preserve the model-theoretic characterization. Refinement with PD can thus be considered as a tool for the step-by-step derivation of logic programs.

Using the Refinement Calculus we can correctly derive logic programs from high-level specifications of problem solving methods such as, for instance, *Divide and Conquer* and specifications of data structures, constructors and deconstructors. Using REFINERY the implementation of the Refinement Calculus, we have developed medium-size logic programs for synthesis of electrical circuits and compilers for hardware description languages. These programs consist of about 200 clauses [1].

The problem of insertion into (resp., deletion from) a logic program is as follows. Given a logic program $P$, a conjunction of literals $W$, and an integrity constraint theory $C$, find a *transaction* $T$, that is a sequence of additions of facts to and/or removal of clauses from $P$, so that $W$ is (resp., no longer is) a logical consequence of (the completion of) $P$. In [8, 9], procedures that update (*normal* and *arbitrary*) programs were given. The underlying idea is to look at an SLDNF-tree for $P \cup \{\leftarrow W\}$ and ways of completing some derivation into a refutation (resp., pruning all non-failed derivations. The correctness and other properties of these procedures were proved in [8, 9].

Due to space constraints, this extended abstract will not contain any further details on the refinement calculus or the update procedures.

# 4 Design Modifications

The Refinement Calculus operations were proven correct. Nevertheless, the synthesized program can be deficient due to, e.g. wrong or insufficient examples. Alternatively, the deficiency can be caused by an unsuitable choice of some design decisions, i.e. refinement operations.

As mentioned earlier, the IDA system relies on the update procedures for (1) finding out causes of errors that are observed in the refined program and (2) updating the refined program if some complementary property is sought. In both cases, especially in the former, it is important to find out those specifications and/or design decisions that led to the observed (unwanted) property. To enable the update procedures to achieve this, we suggest to construct a *Refinement Trace* (RT) during the refinement step.

## 4.1 Construction of the Refinement Trace

The idea underlying the Refinement Trace is to store enough information about the decisions that were taken during the program refinement stage and to represent this information as a logic program. By so doing, this program can be given as input to the update procedures.

For the purposes of conciseness, we will only explain what needs to be represented in cases of folding and unfolding operators. This should help the reader conceptualize how the remaining operators are handled.

**The RT construction procedure** The RT is initially empty. It is constructed as follows.

1. **For** *every clause (i.e. rule or fact) in the original specifications,* **do**

   *Generate for it a unique name $c_i$ and add the fact $c_i(\overline{t}_i) \leftarrow$ to the RT, where $\overline{t}_i$ is the union of all the non-ground arguments that appear in literals of $c_i$.*

2. **For** *every clause $c$ of the form $(A \leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta$ which is obtained by unfolding a clause $c_1$: $A \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$ with respect to the (standardized apart) clause $c_2$: $A'_i \leftarrow B_1, \ldots, B_m$ where $\theta = mgu(A_i, A'_i)$* **do**

   *Add to the RT the clause $c(\overline{t}) \leftarrow \theta', c_1(\overline{t}_1), c_2(\overline{t}_2)$ where $\overline{t}$, $\overline{t}_1$, and $\overline{t}_2$ are obtained as the union of all the non-ground arguments that appear in $c$, $c_1$ and $c_2$, respectively; and $\theta'$ is a conjunction of equalities obtained from $\theta$ so that each substitution $x/t_i$ in $\theta$ appears as $x = t_i$ in $\theta'$.*

3. **For** *every clause $c$ of the form $(A \leftarrow B_1, \ldots, B_i, B\theta, B_{i+1}, \ldots, B_n)$ which is obtained by folding a clause $c_1$: $A \leftarrow B_1, \ldots, B_i, A_1, \ldots, A_k, B_{i+1}, \ldots, B_n$ with respect to the (standardized apart) clause $c_2$: $B \leftarrow A'_1, \ldots, A'_k$* **do**

   *Add to the RT the clause $c(\overline{t}) \leftarrow \theta', c_1(\overline{t}_1), c_2(\overline{t}_2)$ where $\overline{t}$, $\overline{t}_1$ and $\overline{t}_2$ are obtained as the union of all the arguments that appear in $c$, $c_1$ and $c_2$, respectively; and $\theta'$ is a conjunction of equalities obtained from $\theta$ so that each substitution $x/t_i$ in $\theta$ appears as $x = t_i$ in $\theta'$.*

□

Note that in (2) and (3) above, any variable appears exactly once in $\overline{t}$ (the same holds for $\overline{t}_1$ and $\overline{t}_2$). Moreover, if a clause variable also appears as an argument of a function symbol (in this clause), then it suffices to represent the function symbol (with its arguments) as arguments of the corresponding RT clause.

EXAMPLE 1 Consider the unfolding of

$c_1$ : $evaluate(tree(L, Op, R), Val) \leftarrow$
$evaluate(L, Val_1), evaluate(R, Val_2), compose(Val_1, Op, Val_2, Val)$

with respect to

$c_2$ : $compose(Left, +, Right, Value) \leftarrow plus(Left, Right, Value)$

and

$c_3$ : $compose(Left, *, Right, Value) \leftarrow times(Left, Right, Value)$

which produces

$c_4$ : $evaluate(tree(L, +, R), Val) \leftarrow$
$evaluate(L, Val_1), evaluate(R, Val_2), plus(Val_1, Val_2, Val)$
$c_5$ : $evaluate(tree(L, *, R), Val) \leftarrow$
$evaluate(L, Val_1), evaluate(R, Val_2), times(Val_1, Val_2, Val)$

The RT is initialized with:

$c_1(tree(L, Op, R), Val) \leftarrow$
$c_2(Left, Right, Value) \leftarrow$
$c_3(Left, Right, Value) \leftarrow$

Then, the following two clauses are added to it:

$c_4(tree(L, +, R, ), Val, Val_1, Val_2) \leftarrow$
$Left = Val_1$, $Right = Val_2$, $Op = +$, $Val = Value$,
$c_1(tree(L, Op, R), Val), c_2(Left, Right, Value)$
$c_5(tree(L, *, R, ), Val, Val_1, Val_2) \leftarrow$
$Left = Val_1$, $Right = Val_2$, $Op = *$, $Val = Value$,
$c_1(tree(L, Op, R), Val), c_3(Left, Right, Value)$

By construction, every fact of RT denotes a clause (fact or rule) in the (original) specifications and every rule of RT denotes a refinement step (i.e. design decision).

## 4.2 Update of the Refinement Trace

It should be clear by now that an update of the RT would construct an SLDNF-tree which suitably restricts the search to the part of the design which is relevant to the update request. Let us illustrate this by a simple case; a more interesting case will be given in the next section.

EXAMPLE 2 Using the example of Section 4.1, suppose that one wants to delete $evaluate(tree(s(s(0)), *, 0), s(s(0)))$, if it turns out that the program evaluates the multiplication of 2 by 0 as being equal to 2.

Since $evaluate(tree(s(s(0)), *, 0), s(s(0)))$ unifies with the head of clause $c_5$, the following deletion request is issued:

$$delete(c_5(tree(s(s(0)), *, 0), s(s(0)), Val_1, Val_2), rt, \mathcal{T}).$$

This returns the following (alternative) transactions:

$$\begin{aligned}
t_1 &= [retract(c_5(tree(L, *, R, ), Val, Val_1, Val_2) \leftarrow \\
&\qquad Left = Val_1,\ Right = Val_2,\ Op = *, Value = s(s(0)), \\
&\qquad c_1(tree(L, Op, R), Val), c_3(Left, Right, Value)] \\
t_2 &= retract(c_1(tree(L, Op, R), Val) \leftarrow] \\
t_3 &= retract(c_3(Left, Right, Val)) \leftarrow]
\end{aligned}$$

Transaction $t_1$ suggests to undo the unfolding step which produced $c_5$, thus suggesting to reconsider a design decision, whereas transactions $t_2$ and $t_3$ suggest that clause $c_1$ or $c_3$, respectively, is incorrect. Had $c_1$ and $c_3$ been obtained by refinements from earlier specifications, the update procedures would have traced this back using the RT.

In the full paper, we will illustrate the sequencing of the different modules in IDA by means of an example wherein some of the specifications of an evaluator of expressions (which are represented as binary trees) are induced from examples, refined into a logic program, which is then updated to correct some error in its behaviour.

# 5 Related Work

Our work falls within the active area of *Logic Program Synthesis and Development*. (See, for instance, [4], [7], [10] and the good survey given in [6]). The main concern in this area is to automate (as many aspects as possible of) the process of (logic) program design. This includes formalizing the notions of specification (and specification languages), correctness of a logic description, program transformations and their correctness, etc. There has also been some work on the synthesis from incomplete information (e.g. [7] and [5]). In this case, specifications — including program schemata — are generated from examples (by induction) and properties (by deduction).

The emphasis in our work is slightly different than the work we are aware of in the area of logic program development. Though part of IDA deals with the generation of specifications from examples and their refinement into logic programs, it also tackles the problem of debugging and maintaining the refined program *taking into account* the specifications *and* the design decisions. It is noteworthy that this part is more general than work on *algorithmic program debugging* (e.g. [21]) where the debugging of a program does not extend to the design process.

Relevant to our work is research on *Multi-strategy Learning* [17] where the central concern is indeed to bring together different inference modes, e.g. induction, deduction, abduction, and analogy. In [16], for instance, an *Inferential*

*Theory of Learning* is presented, where a detailed study of the various inference types and their operators, called *transmutations*, was presented. One can also mention the idea of a *plausible justification tree* [22], where the aim was to integrate various elementary inferences at a micro-level, that is, at the level of the search tree that is constructed to prove or learn some concept. Closest to our focus among these studies on Multi-strategy Learning is [20], where a system, WHY, is presented, which learns from examples and domain knowledge, and updates the theories that are learned. In their case, knowledge is divided in two parts: (1) a *phenomenological theory* which describes taxonomies, structural information, and general knowledge; this part is used deductively; and (2) a *causal model* which represents effects as well as *constraints* and *contexts* for such cause-and-effect relationships to hold; this part is used abductively. Theory revision in [20] is an abductive process which makes use of the causal model to determine the relevant causal paths from which revisions can be decided.

To our knowledge, research on Multi-strategy Learning has not yet tackled the problem of program design and maintenance. Even [20] is only concerned with diagnosis within one theory (the resulting logic program) rather than a diagnosis or debugging across various theories (programs) as we do in the IDA framework to find various design decisions (successive refinements), or induced specifications that are responsible for some unwanted property of the program, or that should be revised for some different program property to be obtained. The same holds for work on *diagnosis* and *algorithmic debugging*, e.g. [3], [19], and [21], where the debugging of a program does not extend to the design process.

Also related to our work is research on *Intelligent Computer-Aided Design*, ICAD. (See, for instance, [2].) The *General Design Theory* [23] which, based on a set-theoretic model of human concept formation, was presented as the way any design process should proceed. Abduction is used to transform mathematical concepts into axioms which are then used deductively to produce theorems which can be used. Though an interesting step towards trying to understand the design process itself, this study was still at the conceptual level and we are not aware of any concretisation of the principle.

# 6 Conclusion

We have presented in this paper a paradigm, IDA, for program design and maintenance. It uses the three inference rules — Induction, Deduction, and Abduction — to generate (logic) program specifications, and to refine them into a target program. Since we represent the design decisions explicitly as a logic program, they can be modified using the update procedures if the refined program does not behave according to some intended specification. The update

procedures point out possible causes in the design or in the initial specifications (that were possibly induced) of observed symptoms in the design.

The IDA modules have already been implemented. We used MIS [21] as our induction module; the refinement module is implemented in the PAL system, an implementation of the Refinement Calculus [13]; and the update module is an implementation of the update procedures [9]. A compositional semantics for normal logic programs called the perfect $\Omega$-model semantics [11] is the formal foundation of IDA.

As experiments have shown, the IDA framework is a first step towards the (logic-based semi-) automation of the combined process of design and maintenance of logic programs. The fact that IDA is a logic programming system greatly enhances the declarativeness of the programs it represents and, hence, makes it possible to manipulate them and reason about them in a formal and correct way. This, in turn is bound to save a lot of time and effort and, hence, costs that are traditionally spent on the tasks of design and maintenance.

More work is required on IDA however. Firstly, we need to make the integration of the various modules as automated as possible. Though we have not tackled the issue of algorithmically generating tests for the refined program, it would be interesting to study this problem within our paradigm. The enrichment of our framework with *types* ([18]) as well as some capabilities like *modularity*, etc., should be an interesting further research effort. We also plan to test IDA for large programs and see if it would suffer from any bottlenecks and, if so, to find out how to deal with them in a formal way. We also intend to look more closely at the Induction module and compare different ILP systems, looking for ways of getting the most specific and correct and complete specifications.

**Acknowledgments** This research is supported in part by the ESPRIT BRA COMPUNET/NFR contract # 469.92/011 and by the Human Capital and Mobility NFR contract # 101341/410. Thanks to Tore Amble for his help with CLINT and MIS, and to Gunther Sablon for his patient work with the inductive synthesis of specifications from our examples.

# References

[1] A. Bjelland. Formal hardware synthesis via the refinement calculus of logic programs. Master's thesis, The Norwegian Institute of Technology, Knowledge Systems Group, Dept of Computer Systems and Telematics, The Norwegian Institue of Technology, N-7018 Trondheim, Norway, 1994.

[2] D. Brown, M. Waldron, and H. Y. (editors). *Intelligent Computer-Aided Design*. Elsevier Science Publishers B.V. (North Holland), 1992.

[3] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[4] Y. DeVille. *Logic Programming: Systematic Program Development.* Addison-Wesley, 1990.

[5] Y. DeVille and P. Flener. Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In J.-M. Jacquet, editor, *Constructing Logic Programs.* John Wiley and Sons, 1993.

[6] Y. DeVille and K.-K. Lau. Logic program synthesis. *The Journal of Logic Programming (Special Issue: Ten Years of Logic Programming*, 19/20:321–350, May/July 1994.

[7] P. Flener. *Logic Program Synthesis from Incomplete Information.* Kluwer Academic Publishers, 1995.

[8] A. Guessoum and J. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[9] A. Guessoum and J. Lloyd. Updating knowledge bases II. *New Generation Computing*, 10(1):73–100, 1991.

[10] J.-M. Jacquet, editor. *Constructing Logic Programs.* John Wiley and Sons, 1993.

[11] R. Klausen. An investigation of operators in inductive, deductive and abductive reasoning. Master's thesis, The Norwegian Institute of Technology, Knowledge Systems Group, Dept of Computer Systems and Telematics, The Norwegian Institue of Technology, N-7018 Trondheim, Norway, 1994.

[12] J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *Proc. of the ACM Symp. Principles of Programming Languages*, pages 255–267. ACM, 1982.

[13] J. Komorowski and S. Trcek. Towards refinement of definite logic programs. In *Proc. of the Int. Symp. on Methodologies for Intelligent Systems*, pages 315–325. Lecture Notes in Artificial Intelligence, Springer Verlag, 1994.

[14] N. Lavrač and S. Džeroski. *Inductive logic Programming.* Ellis Horwood, 1994.

[15] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[16] R. Michalski. Inferential theory of learning as a conceptual basis for multistrategy learning. In R. Michalski, editor, *Multistrategy Learning*. Kluwer Academic Publishers, 1993.

[17] R. Michalski. *Multistrategy Learning*. Kluwer Academic Publishers, 1993. (Reprinted from Machine Learning, Vol. 11, Nos. 2-3, 1993.).

[18] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[19] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[20] L. Saitta, M. Botta, and F. Neri. Multistrategy learning and theory revision. In R. Michalski, editor, *Multistrategy Learning*. Kluwer Academic Publishers, 1993.

[21] E. Shapiro. *Algorithmic Program Debugging*. The MIT press, 1982.

[22] G. Tecuci. Plausible justification trees: A framework for deep and dynamic integration of learning strategies. In R. Michalski, editor, *Multistrategy Learning*. Kluwer Academic Publishers, 1993.

[23] H. Yoshikawa. General design theory as a formal theory of design. In H. Yoshikawa and D. Gossard, editors, *Intelligent CAD I, Proceedings of the IFIP TC 5/WG 5.2 Workshop on Intelligent CAD, Boston, 1987*. North Holland, 1989.