Effective Algorithmic Debugging for Inductive Logic Programming

Jukka Paakki
Department of Computer Science and Information Systems
University of Jyväskylä
P.O.Box 35, FIN-40351 Jyväskylä, Finland
email: paakki@cs.jyu.fi

Tibor Gyimóthy *Tamás Horváth*
Research Group on the Theory of Automata
Hungarian Academy of Sciences
Aradi vértanúk tere 1, H-6720 Szeged, Hungary
email: {h42gyi,h158hor}@ella.hu

Abstract

As suggested by Shapiro, an inductive learning process can be assisted by an algorithmic debugging phase. Algorithmic debugging is a semi-automatic technique where the user acts as an oracle who can answer queries concerning the intended meaning of the program on the abstract level of procedures. The central problem with the basic algorithm is the potentially large number of queries to the oracle. The method presented in this paper improves Shapiro's original scheme by integrating testing and (static) slicing with algorithmic debugging. This integrated technique is more effective than the basic one, in that the number of questions to the oracle is reduced. First, the oracle is not asked for verification of the results of a procedure call if they can be checked from a test database. And second, only such program execution paths are analyzed that may have affected the value of an incorrect output. The test database component of the system is based on the category partition testing method, and the slicing component applies techniques originally developed for attribute grammars.

1 Introduction

As originally suggested by Shapiro [Sha83], an incremental inductive logic programming approach for learning can be assisted by the user as an "oracle" who can answer questions pertaining to the correctness and consistency of the developed program. Therefore, an obvious way to improve the effectivity of an inductive learner (in terms of query complexity) is to reduce the number of questions to the oracle. The process of semi-automatically verifying consistency in this context is traditionally called *algorithmic debugging*. Note also that algorithmic debugging is a powerful program analysis technique also by itself, outside of a learning framework.

The algorithmic program debugging technique introduced by Shapiro can isolate an erroneous procedure, given a program and an input on which it behaves incorrectly. Shapiro's model was originally applied to Prolog programs to diagnose the following three types of errors: termination with incorrect output, termination with missing output, and nontermination. A major drawback of this debugging method is the great number of queries to the user about the correctness of intermediate results of procedure calls.

An algorithmic debugging method GADT (Generalized Algorithmic Debugging and Testing) for imperative languages is discussed in [FGK91]. A major improvement in the bug localization process is realized in GADT by combining the *category partition testing method* [OsB88] with the algorithm introduced in [Sha83]. The main idea is the following: During the debugging of a program the user has to answer many "difficult" questions. If the program has already been tested, the test results for the procedures of the program can be directly applied in the debugging process without consulting the user. The GADT method also applies dynamic *program slicing* to compute the suspect part of a buggy program, thus further improving the performance of bug localization.

In this paper we present an advanced algorithmic debugging method for logic programs, called IDTS (*Integrated Debugging, Testing, and Slicing*). In the style of GADT, our method integrates category partition testing and slicing into the debugging process, so as to reduce the number of questions to the user.

The program slicing part of our method is based on the *annotation inference* technique introduced in [BPM93b]. Using this technique an annotating specification of directionality (*input/output*) can be automatically generated for the functional part of a logic program.[1] The user may annotate more positions according to his/her intended use of the program, and IDTS checks the functional correctness of the final annotation.

From an annotation a *dependence graph* is constructed for the logic program. A proof (refutation) tree is produced for a buggy program and using the dependence graph the tree is sliced, removing those parts that have no influence on the visible symptom of a bug. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program. The annotation of the program is used for preparing the test database as well: The user may provide test cases over input arguments of the annotated program.

The use of category partition testing in IDTS requires to supply the system with extra knowledge about the input properties of the program, that is, an initial test configuration. However, for the learning process the main properties of central predicates and their arguments can be specified in advance. On the basis of such a higher-order specification the initial test configuration can be automatically defined, without a separate manual preparation phase.

We proceed as follows. First some central notations are presented in Section 2. A description of category partition testing and its integration with algorithmic debugging is given in Section 3. Section 4 presents the static slicing technique. In Section 5 the integrated IDTS method is presented, and finally Section 6 gives conclusions and some remarks on future work.

2 Preliminaries

In this section we present the concepts and notations that are needed for establishing a relation to Shapiro's algorithmic debugging technique and to attribute grammars. In what follows, P will denote a logic (Prolog) program and M its interpretation (that is, intended meaning). T will denote a *proof tree* in the conventional sense where each node stands for a resolved atom, and each subtree with $p(\ldots)$ as the root and $p_1(\ldots), \ldots, p_n(\ldots)$ as its sons corresponds to a clause $p(\ldots) :- p_1(\ldots), \ldots, p_n(\ldots)$ in the program P. The root node of a proof tree corresponds to a query atom. A conjunctive query gives rise to several proof trees, one for each atom in the query.

Let p be a predicate in P of arity n. Each atom $p(t_1, \ldots, t_n)$ is associated with an input arity n_i and an output arity n_o such that $n_i + n_o = n$. Intuitively, this means that the resolved procedure call $p(t_1, \ldots, t_n)$ has n_i input arguments and n_o output arguments. The triple $< p, x, y >$ will denote a (terminating) call of predicate p with input x and output y. When there is no confusion, we will make no distinction between $< p, x, y >$ and the corresponding atom $p(t_1, \ldots, t_n)$ where $(t_1, \ldots, t_m)$ is a permutation of the sequence (x, y).

The *top-level trace* of a resolved triple $< p, x, y >$ is a finite (possibly empty) sequence $(< p_1, x_1, y_1 >, < p_2, x_2, y_2 >, \ldots, < p_n, x_n, y_n >)$, where p on input x first calls the predicate p_1 with input x_1 and output y_1, then the predicate p_2 with input x_2 and output y_2, etc., and finally the predicate p_n with input x_n and output y_n, resulting in output y from p. Hence, the top-level trace corresponds to the body of a clause $p(\ldots) :- p_1(\ldots), p_2(\ldots), \ldots, p_n(\ldots)$ in the program, and to the leaves of a subtree in the proof tree with $< p, x, y >$ as the root.

Let a' be a ground atom and let $C = (a :- a_1, \ldots, a_n)$ be a clause in P. We say that the clause C *covers* the atom a' iff there is a substitution θ such that $a\theta = a'$ and $a_i\theta \in M$ for each $i = 1, \ldots, n$. A clause C is *correct* in M iff all the ground atoms covered by C are in M; otherwise C is *erroneous* (*incorrect*) in M.

We let the clauses of P be numbered $C_1, C_2, \ldots$, and the atoms of every clause be numbered $0, 1, \ldots$, so that the 0:th atom is the head, the 1st atom is the leftmost atom in the body, and so forth. Let q be a predicate in P. We let the argument positions of q be numbered $1, 2, \ldots$, and denote the k:th argument position in q with q_k. Let $\mathcal{A}rgpos(q)$ be the set of argument positions in q, and let $\mathcal{A}rgpos(P) = \bigcup_{q \text{ in } P} \mathcal{A}rgpos(q)$.

Let C be a clause in P of the form $a_0 :- a_1, \ldots, a_n$. We can now unambiguously refer to the k:th argument position in a_j with the tuple (C, j, q, k), where q is the predicate symbol of a_j. We call such a tuple a (*program*) *position*. Let $\mathcal{P}os(P)$ denote the set of positions in P, and $\mathcal{P}os(C)$ the set of positions in C. We will not make a distinction between positions and terms contained in them; e.g. if γ is a position, we write $\gamma\theta$ instead of $t\theta$, where t is the term in γ.

[1]The method has originally been developed for functional logic programs. In IDTS (applied on Prolog) the functional entities are represented by conventional extra–logical primitives, such as arithmetics.

For applying techniques of attribute grammars on logic programs, we must be able to model the data flow within a program. The idea is that some of the predicate arguments in P are *annotated*, either as *inherited* ($\downarrow$) or *synthesized* ($\uparrow$), that is: there is a function (an *annotation*) $\mu : \mathcal{A}rgpos(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \square\}$, where $\updownarrow$ is read *dual*, and $\square$ is read *unannotated*. The flags $\downarrow, \uparrow, \updownarrow, \square$ are called *modes*. An annotation is *partial* if some positions are dual or unannotated. Intuitively, inherited arguments represent input information for a predicate, synthesized arguments represent output information, dual arguments represent information that may be input or output depending on the context, and unannotated arguments represent auxiliary information whose direction is not relevant for the user. Another characterization is that the inherited arguments are *ground* on entry to the predicate and synthesized ones are ground on success of the predicate. For an n-ary predicate q, the annotation $\mu(q_k) = a_k$ for each $k = 1, \ldots, n$ is denoted by $q : (a_1 \ldots a_n)$.

Another important concept is the notion of *input* and *output* arguments of a *clause* C: If $\mu(q_k) = \downarrow$ and a_j is the head atom of C, or if $\mu(q_k) = \uparrow$ and a_j is a body atom in C, we call (C, j, q, k) an *input* position. If $\mu(q_k) = \uparrow$ and a_j is the head atom in C, or if $\mu(q_k) = \downarrow$ and a_j is a body atom in C, we call (C, j, q, k) an *output* position. The intuitive explanation for these names is that data is brought *in* to a clause through the input positions, and sent *out* through the output positions. Notice that dual and unannotated positions do not express a definite direction of data flow, and that is why they are not classified as strictly input or output ones. Let $\mathcal{I}(C)$ and $\mathcal{O}(C)$ denote the input and output positions of C, respectively. We denote $\mathcal{I}(P) = \bigcup_{C \in P} \mathcal{I}(C)$ and $\mathcal{O}(P) = \bigcup_{C \in P} \mathcal{O}(C)$.

In a triple $< p, x, y >$, the input component x must contain at least all the inherited ($\downarrow$) arguments of p. x may also contain some dual ($\updownarrow$) arguments that are used as input in the context of calling $< p, x, y >$. The arguments not in x are contained in the output component y. Hence, the annotation can be used as an aid in preparing the test material.

We write $C_1 \mapsto C_2$ if the clause C_2 can be called from C_1, i.e. there is an atom a in the body of C_1 such that unification of a and the head of C_2 (with variables renamed) succeeds. The relation $\mapsto$ is called the *static call graph*.

Next a Prolog program is given that will be used as the main example throughout the paper.

Example 2.1

(C_1) `program(L1,L2,C,S) :- compute(L1,L2,C,S), print(C), print(S).`
(C_2) `compute(L1,L2,C,S) :- complement(L1,L2,C), sigma(L1,L2,S).`
(C_3) `complement(L,[],L).`
(C_4) `complement(L,[H|T],U) :- member(H,L), delete(H,L,L1), complement(L1,T,U).`
(C_5) `member(X,[X|T]).`
(C_6) `member(X,[Y|T]) :- member(X,T).`
(C_7) `delete(X,[X|T],T).`
(C_8) `delete(X,[Y|Z],[Y|U]) :- delete(X,Z,U).`
(C_9) `sigma(L1,L2,S) :- sum(L1,S1), sum(L2,S2), S is S1+S2.`
(C_{10}) `sum([],0).`
(C_{11}) `sum([X|T],S) :- sum(T,S1), S is S1+X.`

This Prolog program has two input lists of integers `L1` and `L2`. The program computes the difference of the lists by the predicate `complement` (fails if `L2` $\not\subseteq$ `L1`) and the sum of their elements by the predicate `sigma`. Let us introduce the following buggy version of clause C_2 (where `S` has been mistyped as `C` for the predicate `compute`):

(C_2) `compute(L1,L2,C,C) :- complement(L1,L2,C), sigma(L1,L2,S).` $\square$

3 Interactive Diagnosis and Testing

3.1 Shapiro's Method

In this section we will give a short overview of the basic *single-stepping* debugging algorithm presented in [Sha83]. Shapiro's algorithm can isolate an erroneous procedure (clause), given a program and an input on which it behaves incorrectly. The algorithm traverses the proof tree of a program and asks the user about the expected behavior of each resolved goal. The user has to verify the correctness of each goal, in which case a bug inside a certain procedure can be identified.

Let p be a predicate that terminates on input x and returns y as output such that $< p, x, y > \notin M$. This implies that the program P has at least one incorrect clause. For finding it, the proof tree T rooted by $< p, x, y >$ is traversed in a postorder manner. During the traversal over T, at each node $< q, u, v >$ a question is issued whether or not $< q, u, v > \in M$. Let us suppose that the first negative answer is received at the node $< q, u, v >$. Let the direct descendants of $< q, u, v >$ in T be $< q_1, u_1, v_1 >, \ldots, < q_m, u_m, v_m >$. Since a postorder traversal strategy was applied, it holds that $< q_i, u_i, v_i > \in M$ for all i $(1 \leq i \leq m)$. From this follows that the clause $q(\ldots) :- q_1(\ldots), \ldots, q_m(\ldots)$ covers the triple $< q, u, v >$ which is not in M. The algorithm stops at the node $< q, u, v >$ and returns the clause instance $< q, u, v > :- < q_1, u_1, v_1 >, \ldots, < q_m, u_m, v_m >$. By this method an erroneous clause can always be identified assuming that the answers to the membership questions are correct.

Example 3.1 We demonstrate the behavior of the single-stepping method by the buggy program introduced in Example 2.1.

Calling the predicate `program` with arguments $([1,2],[2],$ `C`, `S`$)$ results in `C` $= [1]$ and `S` $= [1]$. Thus, the output value of `S` is incorrect. Let us try to find the erroneous clause using the single-stepping algorithm. The algorithm works as follows: It executes the Prolog program and at each success step asks the user whether the newly determined goal should be *correct* or *incorrect*.

The following proof tree is obtained:

```
program([1,2],[2],[1],[1])
    compute([1,2],[2],[1],[1])
        complement([1,2],[2],[1])
            member(2,[1,2])
                member(2,[2])
            delete(2,[1,2],[1])
                delete(2,[2],[])
            complement([1],[],[1])
        sigma([1,2],[2],5)
            sum([1,2],3)
                sum([2],2)
                    sum([],0)
                    2 is 0+2
                3 is 2+1
            sum([2],2)
                sum([],0)
                2 is 0+2
            5 is 3+2
    print([1])
    print([1])
```

The sequence of queries to the user is given below. The built-in predicates (here, `print` and `is`) are assumed to be correct and no verification of their validity is requested.

Query: Is it ok?	`member(2,[2])`	*correct*
Query: Is it ok?	`member(2,[1,2])`	*correct*
Query: Is it ok?	`delete(2,[2],[])`	*correct*
Query: Is it ok?	`delete(2,[1,2],[1])`	*correct*
Query: Is it ok?	`complement([1],[],[1])`	*correct*
Query: Is it ok?	`complement([1,2],[2],[1])`	*correct*
Query: Is it ok?	`sum([],0)`	*correct*
Query: Is it ok?	`sum([2],2)`	*correct*
Query: Is it ok?	`sum([1,2],3)`	*correct*
Query: Is it ok?	`sigma([1,2],[2],5)`	*correct*
Query: Is it ok?	`compute([1,2],[2],[1],[1])`	**incorrect**

After 11 questions an erroneous procedure call has been identified. The buggy clause instance is

(C_2) `compute` $([1,2],[2],[1],[1])$:- `complement` $([1,2],[2],[1])$, `sigma` $([1,2],[2],5)$.

□

3.2 Category Partition Testing

In this section we give a formalization of the category partition testing method (*CPM*) introduced in [OsB88]. During the process of functional testing the programs (procedures) cannot be tested over all possible properties of their parameters. Hence, the tester's first task is to define the critical properties of input parameters. These critical properties – called *categories* – are investigated in the testing process.

The categories can be divided into disjoint classes – called *choices* – presuming that all the elements within a choice have an identical observable behavior with respect to correctness. When the categories and choices for a program have been defined, then all the possible *test frames* can be generated. Intuitively, a test frame expresses a certain syntactic property of input elements with the same semantic behavior, expressed as a combination of input properties. Hence, a test frame covers exactly one choice from each category. In general, there may be many superfluous test frames among the generated ones, that is, property combinations that are not sensible. These frames can be eliminated by associating *selector expressions* with the choices. A choice can be included in a test frame if the associated selector expression is *true* with respect to the combination of choices for the frame.

The selector expressions make use of *property names*. A property is also associated with a choice and can be considered as a Boolean variable. The value of this variable is *true* if the given frame contains that choice.

In Example 3.2 we give a simple *CPM* specification with the generated test frames for the predicate `delete` of Example 2.1. We assume that the two first arguments of `delete` are input ones. The syntax of the specification is adopted from the T-GEN testing system [SzG91].

Example 3.2 [A category-partition specification for 'delete']

Test specification: `delete:`$(\downarrow\downarrow\uparrow)$
  Category: `number_of_the_elements`
    `zero:` **property** `empty`
    `one:` **property** `singleton`
    `more:` **property** `tuple`
  Category: `position_in_the_list`
    `first:` **if not** `empty`
    `last:` **if** `tuple`
    `inside:` **if** `tuple`
    `none:` **if not** `empty`
End of specification

The category `number_of_the_elements` expresses the view that empty lists (choice `zero`, property `empty`), lists of one element (choice `one`, property `singleton`), and lists of several elements (choice `more`, property `tuple`) probably behave differently and shall therefore be tested separately against `delete`.

The category `position_in_the_list` stresses that when deleting an element e from a list l (whose abstracted syntactic properties are specified by the category `number_of_the_elements`), it should be taken into account whether e is the first element of l (choice `first`), the last element of l (choice `last`), some internal element of l (choice `inside`), or e does not appear in l at all (choice `none`). Therefore, these cases shall be tested separately. The selector expressions, e.g. "**if not** `empty`", reduce the testing space by excluding redundant combinations of input properties, such as "e is the first element of the empty list l".

The generated test frames for all the admissible choice combinations from the two categories are:
{ (`zero`, _), (`one`, `first`), (`one`, `none`), (`more`, `first`), (`more`, `last`), (`more`, `inside`), (`more`, `none`) }.□

In the following we give a short formal description of the category partition method in the context of logic programming. We assume that the domain of interpretation is the set D. Then every predicate p

in P can be considered as a mapping from D^n into D^m, where n and m are the input and output arity of p, respectively.

We assign to every predicate a finite set of input categories each of which is considered as a finite set of syntactic choices. Every choice of a category determines an equivalence class of D^n.

Definition 3.1 [Category] Let p be a predicate in P and n an input arity of p. An equivalence relation over D^n is called a *category* of p. $\square$

Definition 3.2 [Choice] Let p be a predicate in P and c a category of p. An equivalence class of c is called a *choice* of c. $\square$

A test frame can be considered as an element of the products of equivalence relations over D^n. The category partition method checks the behavior of a predicate p against a given test frame by investigating the result of p on a representative input tuple (*test case*) of this frame. The set of test frames may be partially evaluated (tested), i.e., there may be test frames which have not been evaluated yet (this is the usual situation in practice). Therefore we employ an evaluation function (*CPM-testing*) that maps the test frames into the set $\{correct, incorrect, undefined\}$. Having the CPM-testing value *correct* (*incorrect*), the test frame for p has been exploited with a test case against which p behaved correctly (incorrectly). A test frame which has not been tested has *undefined* as its CPM-testing value.

Definition 3.3 [Test specification] Let p be a predicate in P, and let $\Sigma_p = \{\sigma_{p1}, \ldots, \sigma_{pk}\}$ denote the set of categories of p where each $\sigma_{pi} = \{c_{i1}, \ldots, c_{in}\}$ (the c_{ij}'s are the choices). Σ_p is a *test specification* of the predicate p. A test specification of the program P is denoted $\Sigma_P = \{\Sigma_p | p \in P\}$. $\square$

Definition 3.4 [Test frame] Let p be a predicate in P, and let $\Sigma_p = \{\sigma_{p1}, \ldots, \sigma_{pk}\}$ be a test specification of p. Let $F(\Sigma_p)$ denote a subset of the Cartesian product $\sigma_{p1} \times \ldots \times \sigma_{pk}$. (We assume that $F(\Sigma_p)$ covers the whole input space D^n.) The set $F(\Sigma_p)$ is a classification of D^n and therefore an equivalence relation over D^n. An element in $F(\Sigma_p)$ is called a *test frame* of p. The set of test frames of the program P is denoted $F(\Sigma_P) = \{F(\Sigma_p) | p \in P\}$. $\square$

Definition 3.5 [Test case] Let p be a predicate in P with input arity n and with the test specification $\Sigma_p = \{\sigma_{p1}, \ldots, \sigma_{pk}\}$. Furthermore, let $f = (c_{p1}, \ldots, c_{pk})$ be a test frame of p. A *test case* t of f is an element of D^n for which $t \in c_{pi}, 1 \leq i \leq k$. (If f denotes an empty product then such a test case does not exist.) $\square$

Definition 3.6 [CPM-testing] Let p be a predicate of P and Σ_p a test specification of p. Then a *CPM-testing* of p by Σ_p is a mapping $\phi_p \colon F(\Sigma_p) \to \{correct, incorrect, undefined\}$. Let f be an arbitrary test frame of p and t a test case of f. $\phi_p(f)$ is *correct/incorrect* iff p has been applied on t and the respective indication of correctness was obtained from the oracle, otherwise $\phi_p(f)$ is *undefined.* (If f denotes an empty product then $\phi_p(f) = correct$). $\square$

The category partition testing method is based on the assumption that the observable behavior of the program is the same (that is, correct or incorrect) for *all* the different test cases within each categorized test frame. This is an engineering principle rather than a formally verified property, and may thus be invalidated during the testing process. In that case the category partition is inconsistent and must be updated by allocating the conflicting test cases into separate consistent test frames. Note that such a refinement may introduce new test frames and, consequently, new choices as well.

The test frames with their testing evaluation are maintained in a test database (a *CPM test configuration*) that is incrementally refined during debugging and upon recognition of inconsistency within the current category partition. For consulting the database, a *searching function* is applied that maps the concrete test cases into their representative test frame.

Definition 3.7 [Consistent test frame] Let p be a predicate of P and f a test frame of p. f is *consistent* iff $\phi_p(f) \neq undefined$ and $\phi_p(f)$ is independent of the test case for f, and *inconsistent* otherwise. $\square$

Definition 3.8 [Consistent CPM-testing] Let p be a predicate in P and Σ_p a test specification of p. The CPM-testing ϕ_p of p by Σ_p is *consistent* iff every test frame in $F(\Sigma_p)$ is consistent, and *inconsistent* otherwise. □

Definition 3.9 [Searching function] Let p a predicate in P of input arity n and Σ_p a test specification of P. Let $\chi_p\colon D^n \to F(\Sigma_p)$ be a mapping, such that $\chi_p(x) = f$ iff f is the test frame corresponding to x (x is a test case). From the previous definitions follows that χ_p is a function. (That is, χ_p is the *searching function.*) □

Definition 3.10 [CPM test configuration] A *CPM test configuration* of the program P over the set of test frames $F(\Sigma_p)$ is a finite set of five-tuples, $T(F(\Sigma_P)) = \{(p, f, i, o, e) \mid p \text{ is a predicate, } f \in F(\Sigma_p),$ i is a test case of f, o is the output of predicate p on input i, and $e = \phi_p(f)\}$. A CPM test configuration is considered as a *test database*. For a predicate p, the CPM test configuration $T(F(\Sigma_p))$ contains all the elements of $T(F(\Sigma_P))$ whose first component is p. □

Definition 3.11 [Refinement of a test specification] Let σ be an equivalence relation over D^n and let x be an arbitrary element of D^n. Let $\sigma[x]$ denote the class of σ containing x. Let σ_1 and σ_2 be two equivalence relations over D^n. We write $\sigma_2 \geq \sigma_1$ iff $\sigma_2[a] = \sigma_2[b]$ implies that $\sigma_1[a] = \sigma_1[b]$ for any $a, b \in D^n$. Let $F(\Sigma_p)$ and $F(\Sigma'_p)$ be two sets of test frames of a predicate p in P. We say that the CPM test configuration of p, $T(F(\Sigma'_p))$, is a *refinement* of $T(F(\Sigma_p))$ iff $F(\Sigma'_p) \geq F(\Sigma_p)$. □

Proposition 3.1 If the CPM test configuration $T(F(\Sigma'_p))$ of a predicate p is a refinement of $T(F(\Sigma_p))$, then the number of consistent test frames in $T(F(\Sigma'_p))$ is greater than or equal to those in $T(F(\Sigma_p))$. □

3.3 The Interactive Diagnosis and Testing Algorithm

In Section 3.1 the single-stepping method for algorithmic debugging of logic programs was summarized. This algorithm can be applied on those programs that terminate on some input and result in an incorrect output. The number of queries to the user depends on the position of the incorrect clause instances within the proof tree. In order to reduce the number of user interactions an extended method called *divide-and-query* has been proposed in [Sha83]. In this section we introduce an alternative interactive debugging technique, IDT [HGA93]. The presented method is founded on the single-stepping technique. Notice, however, that IDT can be integrated with divide-and-query as well.

Assume that there exists a CPM test configuration for the program P, i.e., for each predicate p in P a set of test frames is defined each of which may have an evaluation. In the process of algorithmic debugging we assume that the given CPM test configuration is consistent. From this assumption follows that if a test frame is already evaluated then this evaluation is independent of the chosen test case. This is only a hypothesis of the test designer which may be false. Therefore we must take into account that there may be inconsistent test frames represented in the CPM test configuration. If a referred test frame is not evaluated a question about its correctness is issued to the user.

Let p be a predicate and let us suppose that $<p,x,y> \notin M$ for some x and y with the complete proof tree rooted by $<p,x,y>$ being finite. Similarly to the single-stepping method, our algorithm traverses this tree in a postorder manner. Let us suppose that we have to verify whether or not $<q,u,v>$ is in M, where $<q,u,v>$ (that is, the corresponding atom) is a node in the tree. We examine with the searching function whether the test frame for q and u is already evaluated (i.e., $\phi_q(\chi_q(u)) \neq$ *undefined*). If the test frame has already been evaluated then the result of the evaluation is immediately taken for the correctness of $<q,u,v>$ and the user is not consulted.

Since the hypothesis for consistency of the test frames is not formally proved, the result given by our algorithm must be verified. First, it is possible that the algorithm does not find the erroneous clause, due to missing the incorrectness of an atom when consulting the inconsistent test database. Second, if an erroneous clause has been identified by the IDT algorithm, it may occur that the test frame for the head of the clause instance is inconsistent, or the body contains an atom whose test frame is inconsistent. The latter case means that the clause instance found is not the deepest incorrect one in the tree. For resolving this inconsistency the original single-stepping method is invoked.

All the questions and answers are stored, and thus the number of queries the user has to answer is less than or equal to (in the worst case) the single-stepping method. (The same holds for a comparison to the divide-and-query algorithm.) With consistent CPM-testing, the number of queries may be drastically reduced. If an inconsistent test frame is found during debugging, the user has to modify the test database in such a way that the conflicting test cases will be assigned to different test frames. More exactly, the user has to define a refinement for the CPM test configuration such that the new configuration will be consistent with regard to all known test cases.

Algorithm 3.1 below shows the basic method of integrating CPM-testing with the algorithmic debugger of Shapiro [Sha83]. The algorithm traverses the proof tree for an erroneous query triple $< p, x, y >$ in postorder and consults the test database for verifying the correctness of the nodes in the tree.

Algorithm 3.1 [Algorithmic debugging + CPM-testing]

Input: $< p, x, y >$: an atom not in M;
T: a proof tree.
Output: q: erroneous clause instance;
found: *true* if q has been identified,
false otherwise.

procedure Debug($< p, x, y >, T, q, found$);
begin
let $(< p_1, x_1, y_1 >, \ldots, < p_n, x_n, y_n >)$ be the top-level trace of $< p, x, y >$;
found:= *false*; $q := empty$; i:= 1;
while (i $\leq$ n) **and** (**not** *found*) **do**
Debug($< p_i, x_i, y_i >, T, q, found$);
i:= i+1;
end while;
if not *found* **then**
if $\phi_p(\chi_p(x)) = undefined$ **then** Query($< p, x, y >$); **end if**; /* see below */
if $\phi_p(\chi_p(x)) = incorrect$ **then**
found := *true*;
$q := (< p, x, y > :- < p_1, x_1, y_1 >, \ldots, < p_n, x_n, y_n >)$;
end if;
end if;
end Debug.

Due to a possibly inconsistent CPM test configuration, procedure *Debug* of Algorithm 3.1 above may miss the (deepest) erroneous clause instance in the proof tree. Such an anomaly is taken care by the main algorithm of our integrated debugging method. This Algorithm 3.2 verifies the obtained result (that is, the identified erroneous clause instance) from the user and invokes the single-stepping algorithm if he/she rejects the result.

Algorithm 3.2 [IDT]

Input: $< p, x, y >$: an atom not in M;
T: a proof tree.
Output: r: an erroneous clause instance.

procedure IDT($< p, x, y >, T, r$);
begin
Debug($< p, x, y >, T, r, found$);
let $r = (< q, u, v > :- < q_1, u_1, v_1 >, \ldots, < q_n, u_n, v_n >)$;
if (**not** *found*) **or** (Query($< q, u, v >$) = *correct*) **or**
(Query($< q_i, u_i, v_i >$) = *incorrect* **for some** $i = 1, \ldots, n$) **then**
/* inconsistent test database */
fp($< p, x, y >, T, r$); /* see below */
/* else the identified r is an erroneous clause instance */
end if;
end IDT.

The user oracle is consulted (a) if a node with *undefined* CPM-testing is encountered during the debugging traversal (Algorithm 3.1), and (b) when the obtained result is verified against the intended meaning of the program (Algorithm 3.2). In case (a) the test database is extended with the indicated CMP-testing value, while case (b) may reveal a frame inconsistency between the test database and the user's intended interpretation. In the latter case the test specification must be refined by the user. User consultation is not necessary, however, if the test database includes an already verified identical test case. The user consultation process is defined in Algorithm 3.3 below.

Algorithm 3.3 [Oracle]
Input: $< p, x, y >$: an atom to be verified against M.
Output: *correct* if $< p, x, y > \in M$,
incorrect if $< p, x, y > \notin M$.

function Query($< p, x, y >$): $\{correct,\ incorrect\}$;
begin
  if $\exists(p, f, x, y, e) \in T(F(\Sigma_P))$ **where** $(e \neq undefined)$ **then return** e
  else
    $cpm := \phi_p(\chi_p(x))$;
    $oracle :=$ "Query: Is it ok?: $< p, x, y >$" ;   /* user consultation */
    if $cpm = undefined$ **then**
      $T(F(\Sigma_P)) := T(F(\Sigma_P)) \cup \{(p, \chi_p(x), x, y, oracle)\}$
    else
      if $(cpm \neq oracle)$ **then**   /* inconsistent test database */
        refine the CPM test configuration $T(F(\Sigma_p))$ into $T(F(\Sigma_p^{'}))$
        by introducing the test frames (and their choices)
        $f_1, f_2, \ldots, f_n$, and by allocating the test cases of $\chi_p(x)$
        consistently into $f_1, f_2, \ldots, f_n$;
        /* now $F(\Sigma_p^{'}) = F(\Sigma_p) \setminus \{\chi_p(x)\} \cup \{f_1\} \cup \{f_2\} \cup \ldots \cup \{f_n\}$ */
      end if;
    end if;
    return *oracle*;
  end if;
end Query.

The procedure $fp(< p, x, y >, T, r)$ stands for the standard single-stepping algorithm, except that user consultation is done as in Algorithm 3.3 using the test database as a repository of tested cases. A detailed description of *fp* is omitted here, and we refer instead to [Sha83].

Example 3.3 We demonstrate the IDT method with our example. Suppose that predicate `delete` has the CPM test specification presented in Example 3.2. The initial state of the CPM test configuration is given below.

$T(F(\Sigma_P))$:

(`delete`, (`zero`, _), (a, [ ]), *fails*, *correct*)
(`delete`, (`one`, `first`), (a, [a]), [ ], *correct*)
(`delete`, (`one`, `none`), _, _, *undefined*)
(`delete`, (`more`, `first`), _, _, *undefined*)
(`delete`, (`more`, `last`), _, _, *undefined*)
(`delete`, (`more`, `inside`), _, _, *undefined*)
(`delete`, (`more`, `none`), _, _, *undefined*)

For instance, the item (`delete`, (`one`, `first`), (a, [a]), [ ], *correct*) in the test database denotes that the predicate `delete` has been tested against the test frame (`one`, `first`). The input was (a, [a]) and the output [ ]; hence the item corresponds to the atom `delete`(a, [a], [ ]). This result is correct (the last argument in the test item is *correct*). The item (`delete`, (`more`, `last`), _, _, *undefined*) denotes that the predicate `delete` has not been tested against the test frame (`more`, `last`). Using this database, the IDT method can identify the buggy clause of Example 2.1 after the following questions (cf. Example 3.1).

Query: Is it ok?	`member`(2, [2])	*correct*
Query: Is it ok?	`member`(2, [1, 2])	*correct*
Query: Is it ok?	`delete`(2, [1, 2], [1])	*correct*
Query: Is it ok?	`complement`([1], [ ], [1])	*correct*
Query: Is it ok?	`complement`([1, 2], [2], [1])	*correct*
Query: Is it ok?	`sum`([ ], 0)	*correct*
Query: Is it ok?	`sum`([2], 2)	*correct*
Query: Is it ok?	`sum`([1, 2], 3)	*correct*
Query: Is it ok?	`sigma`([1, 2], [2], 5)	*correct*
Query: Is it ok?	`compute`([1, 2], [2], [1], [1])	**incorrect**

The IDT algorithm does not generate a user query for the node `delete`(2, [2], [ ]) since its correctness can be verified from the test database by the item (`delete`, (`one`, `first`), (a, [a]), [ ], *correct*) with the same input pattern (`one`, `first`). However, the user has to verify the correctness of `delete`(2, [1, 2], [1]), because the CPM-testing value of the test frame (`more`, `last`) is *undefined*. After the reply from the user, the database item for the test frame is modified into:

(`delete`, (`more`, `last`), (2, [1, 2]), [1], *correct*)

After that, the IDT method will no more generate queries for nodes belonging to the test frame (`more`, `last`) of predicate `delete`. □

4 Static Slicing

Analysis, maintenance, and modification of sizable programs is a laborous task, unless the programs are decomposed into coherent semantic units. One program representation supporting such focused views are *program dependence graphs* that have been suggested for software engineering tasks such as program understanding, program integration, program differencing, debugging, and testing [HoR92, BaH93].

In (algorithmic) debugging, *program slicing* is a standard technique applied in reducing the debugger's search space by extracting a suspect part from the program. Intuitively, a program slice with respect to a specific variable (at some program point) contains all those parts of the program that may affect the value of the variable, or that may be affected by the value of the variable. In the first case, the analysis technique is conventionally referred to as *backward slicing*, and in the second case as *forward slicing*, illustrating the data flow direction of producing the slice. The concept of slicing was originally introduced by Weiser [Wei84]. Later improvements are presented e.g. in [HBR90], in [KoL90], and in [Kam93].

Slicing has typically been applied on imperative languages, such as Pascal [KoL90, Kam93]. Such languages are amenable to analyzing the dependences between variables, thanks to explicitly expressing the data flow within the program in terms of assignments and input/output parameters. This does not hold for logic programming languages where the notions of data flow and data dependences are rather implicit and thus harder to extract. The data dependences within a logic program can, however, be approximated with techniques based on analyzing the *sharing* of logical variables: If two terms of a clause have a common variable, they are dependent and (possibly) represent a data flow.

The (static) slicing part of IDTS is based on techniques developed for *attribute grammars*, a formalism originally proposed for defining the semantics of programming languages. We do not more precisely discuss the properties or applications of attribute grammars, but refer e.g. to [Knu68], [DJL88], and [AlM91]. The fact that makes it feasible to relate logic programs and attribute grammars is that the semantics of both of them can be defined in terms of labelled trees. For logic programs these are the *proof trees* that are generated for a given goal. For attribute grammars the corresponding concept is an *attributed* (or *decorated*) *parse tree* with respect to an input string. The labels express argument values for a logic program and attribute instances for an attribute grammar. Based on this observation, these two formalisms can be related both conceptually and operationally, as shown by the seminal work of Deransart and Małuszyński [DeM85, Mał91, DeM93].

Practical logic programs, such as Prolog programs, contain "moded" terms that must be in a certain instantiation state in order for the computation to succeed. Typical examples are arguments of the built-in arithmetic predicate `is/2` and of comparisons, that must be *ground* at invocation of the predicate. Such terms are referred to as *functional expressions*, emphasizing the view that the associated predicates behave like deterministic functions. In our terminology, this means that some built-in predicates are inherently annotated. For instance, `is/2` has the annotation $(\uparrow\downarrow)$, and the binary comparison predicates have the annotation $(\downarrow\downarrow)$.

It is possible to statically verify the functional expressions of a program against the groundness requirement. This condition of *operational completeness* is fulfilled, if all the functional expressions are ground on success of program execution. Operational completeness for a class of functional logic programs is discussed in [BPM93a, BPM93b]. Such programs are more general than Prolog programs since functional expressions can appear at arbitrary positions as *functional terms*. Annotation-based implementation techniques of functional logic programs are presented e.g. in [Boy93], in [Paa93], and in [Paa94]. For simplifying the presentation, we do not discuss the debugging of general functional logic programs in this paper but assume that all the functional expressions are associated with built-in annotated predicates of Prolog.[2]

In the following we present a method for statically verifying that all the functional expressions are ground when execution reaches them. In a logic program, information can be passed in two ways: either *within* a clause (between two arguments sharing a variable), or *between* two clauses (through unification at an SLD-resolution step). Let us assume that there exists an annotation $\mu : \mathit{Argpos}(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \Box\}$. As for attribute grammars, this information can be used to extract the *direction* of information flow: from input positions to output positions within a clause, and from output positions to input positions at unification. Recall from Chapter 2 that input and output positions are defined in terms of inherited $(\downarrow)$ and synthesized $(\uparrow)$ positions only and do not consider dual $(\updownarrow)$ or unannotated $(\Box)$ positions whose data-flow direction is statically unknown. To reflect the data flow, we introduce the notions of *local dependence graph* $\rightsquigarrow_C$ for each clause C and *transition graph* $\rightsquigarrow_{C,D}$ for each pair of clauses C and D.

Definition 4.1 [Local dependence graph] For each clause C, the *local dependence graph* $\rightsquigarrow_C \subseteq \mathcal{I}(C) \times \mathcal{O}(C)$ is defined as follows:

$$\beta \rightsquigarrow_C \gamma \ \text{ iff } \beta \text{ and } \gamma \text{ have at least one common variable}$$

$\Box$

Definition 4.2 [Transition graph] Let C and D be two clauses, b_0 the head atom in D, and a_j a body atom in C such that a_j and b_0 unify. The *transition graph* $\rightsquigarrow_{C,D}$ on $\mathcal{O}(C \cup D) \times \mathcal{I}(C \cup D)$ is defined as follows:

$$\gamma \rightsquigarrow_{C,D} \beta \quad \text{iff} \quad \left\{ \begin{array}{l} \gamma = (C, j, q, k) \\ \beta = (D, 0, q, k) \\ \mu(q_k) = \downarrow \end{array} \right. \quad \text{or} \quad \left\{ \begin{array}{l} \gamma = (D, 0, q, k) \\ \beta = (C, j, q, k) \\ \mu(q_k) = \uparrow \end{array} \right.$$

$\Box$

As for attribute grammars, the transition graph specifies how clauses and their arguments are "pasted" together in a proof tree. If the connected positions are inherited, data flows from the (calling) body atom of clause C into the (called) head atom of clause D. The flow is reverse for synthesized positions, from the (called) head atom of D into the (calling) body atom of C.

The total functional data flow for a logic program can be modelled by combining the local dependence graphs with the transition graphs. This *global dependence graph* is an approximation of the data flow in any proof tree for an operationally complete program.

Definition 4.3 [Global dependence graph] The *global dependence graph* $\rightsquigarrow_G$ is defined as follows:

$$\rightsquigarrow_G = \bigcup_{C \in P} \rightsquigarrow_C \ \cup \ \bigcup_{C,D \in P} \rightsquigarrow_{C,D}$$

We let $\rightsquigarrow_G^*$ denote the transitive and reflexive closure of $\rightsquigarrow_G$. $\Box$

[2]Notice, however, that our techniques apply to functional logic programs as well.

Let γ be an output position. Groundness of γ is assured under the standard computation rule of Prolog,[3] if (1) all the minimal elements of the global dependence graph that have a path to γ are ground, and (2) all the dependence edges leading to γ flow in the same manner as the execution, depth-first, left-to-right. Property (1) is covered by the notion of simple annotation (see below). Property (2) is analogous to the definition of *L-attributed grammars* [LRS74, Boc76] that apply a preorder attribute evaluation strategy (see the definition of L-annotated position below).

The final necessary condition for operational completeness is *non-circularity*. Intuitively, if the annotation is simple and non-circular, recursive equations of the form $X = f(X)$ will never have to be solved during unification. Since L-annotated programs are always non-circular, the subject is not discussed here in more detail but we instead refer to [BPM93b, Paa93, Paa94].

Definition 4.4 [Simple position] A position $\gamma \in \mathcal{O}(P)$ is *simple* if for all output positions β in every clause C holds:

$$\beta \rightsquigarrow_G^* \gamma \implies \text{every variable in } \beta \text{ occurs at some input position in } C$$

Further, the *annotation* is simple if every position in $\mathcal{O}(P)$ is simple. Note that if γ is simple, and if $\beta \in \mathcal{O}(P)$ and $\beta \rightsquigarrow_G^* \gamma$, then β is also simple. □

For L-annotation, an auxiliary definition is given that extracts all such local dependences that may affect the value of the (functional) position γ:

Definition 4.5 Let D and C be two clauses such that $D \mapsto^* C$, and let $\gamma \in \mathcal{O}(C)$. $\rightsquigarrow_\gamma$ is defined by:

$$\beta_1 \rightsquigarrow_\gamma \beta_2 \quad \text{iff} \quad (\beta_1 \rightsquigarrow_D \beta_2 \quad \text{and} \quad \beta_2 \rightsquigarrow_G^* \gamma)$$

□

Definition 4.6 [L-annotated position] A position $\gamma \in \mathcal{O}(P)$ is *L-annotated* if:

$$(C, i, p, k) \rightsquigarrow_\gamma (C, j, q, l) \implies (i < j \quad \text{or} \quad j = 0)$$

□

As for L-attributed grammars, all the local dependences that affect the value of an L-annotated position must be from left to right within the clause, or directed to the head (in which case the position is synthesized). Now the central condition can be stated as the following proposition. A proof is given in [BPM93a]. Notice that due to restricting functional expressions to built-in Prolog predicates, they always appear at an output (inherited) position in the body of a clause.

Proposition 4.1 Let C be a clause $a_0 :- a_1, \ldots, a_n$, and let γ be a simple and L-annotated position in the atom a_j, containing a functional expression e. If a_j is the current atom and θ the current substitution, then $e\theta$ is ground and can immediately be evaluated. □

Since the immediate evaluation property of functional expressions is essential for correctness of a logic (Prolog) program, our system verifies it statically using the concepts presented above (with some optimizations, such as minimizing the local dependence graphs to select for each output variable *one* input position only). If there exists a functional expression in a position which is not simple and L-annotated, the program is considered potentially invalid and a warning is emitted. Note, however, that a conservative static analysis method can only ensure validity of the program but not absolute run–time invalidity. Hence, a "potentially invalid" program may well behave correctly when executing it.

Since it may be hard for the user to provide a proper annotation, it can be *automatically inferred* [BPM93a, BPM93b]. In the integrated debugging method, inference starts from a minimal annotation that must necessarily be preserved by the user. Initially, all the predicate positions get the default annotation (□), except for the built-in predicates that have a predefined annotation. Let C be a clause

[3] Recall that γ shall be ground on entry to the associated goal if γ is inherited, and ground on success if γ is synthesized.

a_0 :− $a_1, \ldots, a_n$, let e be a functional expression contained in the atom $a_j (1 \leq j \leq n)$, and let v be a variable in e. Let $\gamma = (C, i, q, k)$ be a non-functional position in C such that γ contains v and $i < j$. If γ is *unique* in C, then it is annotated as an input position. (Recall the simple and L-annotated conditions.) If there does not exist any γ for some variable v in e, or if the minimal annotation becomes inconsistent (that is, making some position both an inherited and a synthesized one), the program is considered potentially invalid and a warning is emitted to the user.

The obtained consistent annotation is minimal in the sense that it only assigns a mode for a position when there is no choice with respect to operational completeness and evaluation of functional expressions. After this, the user may annotate the program further in terms of its intended use by assigning a mode on those positions that are not constrained into a fixed data flow. That is, the user may define one of the modes $\downarrow, \uparrow, \updownarrow$ for each predicate position which is still unannotated ($\Box$). Typically the user annotates the top-level predicate, but in general he is free to annotate as many (or as few) positions that he thinks are relevant for his intended model of the program. The degree of annotation, however, may significantly influence effectivity of debugging: The more inherited ($\downarrow$) and synthesized ($\uparrow$) annotations, the more precise category partition testing and slicing.

After obtaining an extended initial annotation from the user, the system verifies operational validity of the program (see Proposition 4.1). The initial annotation is incrementally and consistently extended, until a simple and L-annotated one is found.[4] The inference algorithm assigns a data flow mode ($\downarrow, \uparrow$) on those unannotated positions only that may affect the evaluation of functional expressions. Consequently, the final annotation may still be partial. If no valid annotation is found, the program is considered potentially invalid. There may be several valid annotations, in which case the user can choose the proper one by backtracking over the inference algorithm.

The program dependence graph is constructed simultaneously with annotating the program. For slicing it is necessary to know the data dependences on dual and unannotated positions as well, because they also may be the source of incorrect results even though the direction of their influence is not statically known. Therefore, the program dependence graph is a superset of the global dependence graph, by including a conservative two-directional data flow for all the dual and unannotated positions.

Definition 4.7 [Program dependence graph] Let C be a clause, and let $\mathcal{U}(C)$ denote the set of dual and unannotated positions of C. Let $\beta \in \mathcal{U}(C)$, and $\gamma \in \mathcal{P}os(C)$. The graph $\leftrightarrow_C \subseteq \mathcal{P}os(C) \times \mathcal{P}os(C)$ is defined as follows:

$$\beta \leftrightarrow_C \gamma \ \text{ and } \ \gamma \leftrightarrow_C \beta \ \text{ iff } \beta \text{ and } \gamma \text{ have at least one common variable}$$

Let C and D be two clauses, b_0 the head atom in D, and a_j a body atom in C such that a_j and b_0 unify. Let $\beta = (C, j, q, k) \in \mathcal{U}(C)$, $\gamma = (D, 0, q, k) \in \mathcal{U}(D)$. The graph $\leftrightarrow_{C,D} \subseteq \mathcal{U}(C \cup D) \times \mathcal{U}(C \cup D)$ is defined as follows:

$$\beta \leftrightarrow_{C,D} \gamma \ \text{ and } \ \gamma \leftrightarrow_{C,D} \beta$$

The *program dependence graph* $\rightsquigarrow_P$ is defined as follows:

$$\rightsquigarrow_P \ = \ \rightsquigarrow_G \ \cup \ \bigcup_{C \in P} \leftrightarrow_C \ \cup \ \bigcup_{C,D \in P} \leftrightarrow_{C,D}$$

We let $\rightsquigarrow_P^*$ denote the transitive and reflexive closure of $\rightsquigarrow_P$. $\Box$

In debugging, we are interested in finding the clause of the program that is responsible for an incorrect result of the program's execution. Slicing is a method that can be applied to focusing the debugging process on relevant aspects only and, consequently, to minimizing the number of questions to the user. The starting point of algorithmic debugging is an indicated erroneus output value x of the program execution, or several of them. When given a proof tree for the incorrect computation as input, the (backward) slicer will construct a reduced tree that includes all such nodes of the original proof tree that may have been involved in producing the value x, and *excludes* all the nodes that are not relevant for

[4] That is, all the inherited and synthesized positions must be both simple and L-annotated. This is sufficient for ensuring their groundness, the characteristic property of the inherited and synthesized modes (under the assumption that the property holds for the initial query as well). Notice that the user-given partial annotation must also be verified to consistently make it groundness-preserving with the automatically inferred annotation.

x. After that, the program is algorithmically debugged over the sliced proof tree only. In the definition below, no distinction is made between a node in the proof tree and the corresponding atom in the program. This notational simplification is due to the fact that each node in the proof tree corresponds to exactly one atom in the program. Notice that this correspondence also makes it possible to analyze proof trees in terms of the program dependence graph.

Definition 4.8 [Slice] Let P be a logic program, $\leadsto_P$ its program dependence graph, and T a proof tree for the program with an indicated position γ.[5] A (*static*) *slice* over the proof tree T with respect to γ is a tree, such that

1. Every node in the slice is a node of T.
2. A node n of T is in the slice iff it contains a position β such that $\beta \leadsto_P^* \gamma$. □

Notice that, by the definition of the program dependence graph, a slice over a proof tree T is always a connected region of T (that is, a tree), but not necessarily a complete subtree of T. Notice also that while each subtree of a proof tree is an instance of a complete program clause, a subtree of a slice may correspond to just a part of a clause, with some body atoms excluded. Hence, for extracting an invalid clause from the program using algorithmic debugging, the mapping of subtrees to clauses must be preserved in the slice.

Example 4.1 Let us study our example program containing a bug in clause C_2 (see Example 2.1). We assume that the built-in predicates have the normal annotations `is/2`: (↑↓), `print/1`: (↓). Hence, the terms in the second argument of `is` and in the argument of `print` must be ground at invocation time and are considered functional expressions.

Program annotation starts with producing the minimal operationally safe annotation with the inference algorithm. The program contains four functional expressions, `C` and `S` as the argument of `print` in clause C_1, and the arithmetic expressions `S1+S2` and `S1+X` as the argument of `is` in clauses C_9 and C_{11}, respectively. Since the variables in the arithmetic expressions appear at a unique non-functional position in their respective clause, we annotate these positions as input ones: `sum`: (↓↑). At the same time, the following local dependences are introduced:

$(C_9, 1, \texttt{sum}, 2) \leadsto_{C_9} (C_9, 3, \texttt{is}, 2)$ (`S1`)
$(C_9, 2, \texttt{sum}, 2) \leadsto_{C_9} (C_9, 3, \texttt{is}, 2)$ (`S2`)
$(C_{11}, 1, \texttt{sum}, 2) \leadsto_{C_{11}} (C_{11}, 2, \texttt{is}, 2)$ (`S1`)
$(C_{11}, 0, \texttt{sum}, 1) \leadsto_{C_{11}} (C_{11}, 2, \texttt{is}, 2)$ (`X`)

The user may annotate the program further. Assume that he is interested in the top-level predicate only: `program`: (↓↓↑↑). An initial annotation has been obtained, with the rest of the positions being in the unannotated mode □.

Now the program must be verified against the operational criteria (see Proposition 4.1): It is checked that the four functional expressions in the program are always ground when invoking the associated call. For instance, let us analyze the expression `S1+X` in clause C_{11}. In order for `X` to be ground, the first argument of the head shall be ground on entry to the clause (because it contains `X`). This implies that when calling the predicate at two occurrences of `sum` in clause C_9, their input arguments `L1` and `L2` must be ground. This reasoning introduces the following transition dependences between the clauses C_9 and C_{11}:

$(C_9, 1, \texttt{sum}, 1) \leadsto_{C_9, C_{11}} (C_{11}, 0, \texttt{sum}, 1)$ (`L1`)
$(C_9, 2, \texttt{sum}, 1) \leadsto_{C_9, C_{11}} (C_{11}, 0, \texttt{sum}, 1)$ (`L2`)

In order for `L1` and `L2` to be ground when calling `sum` in clause C_9, the first two arguments of `sigma` (the only candidate input positions containing `L1` and `L2`) have to be ground when entering the clause. Therefore the annotation is extended with the inferred input positions, `sigma`: (↓↓□), and the following local dependences are introduced for clause C_9:

[5] Usually the indicated position stands for an erroneous output value (in mode synthesized, dual, or unannotated), and is thus associated with a top-level predicate of P.

$(C_9, 0, \texttt{sigma}, 1) \leadsto_{C_9} (C_9, 1, \texttt{sum}, 1)$ (L1)
$(C_9, 0, \texttt{sigma}, 2) \leadsto_{C_9} (C_9, 2, \texttt{sum}, 1)$ (L2)

Likewise, for the transition $\leadsto_{C_2,C_9}$ we infer the groundness of $\texttt{sigma}_1$ and $\texttt{sigma}_2$ in clause C_2 by selecting its occurrences containing `L1` and `L2` as input ones: `compute`:$(\downarrow\downarrow\Box\Box)$. Note that clause C_2 involves a choice with respect to `L1` and `L2` that both appear at two candidate positions, $(C_2, 0, \texttt{compute}, 1)$ and $(C_2, 1, \texttt{complement}, 1)$ for `L1`, and $(C_2, 0, \texttt{compute}, 2)$ and $(C_2, 1, \texttt{complement}, 2)$ for `L2`. In both cases we prefer the *leftmost* position, as suggested by the L-annotated property (see Definition 4.6). The synthesis results in the following local dependences for clause C_2:

$(C_2, 0, \texttt{compute}, 1) \leadsto_{C_2} (C_2, 2, \texttt{sigma}, 1)$ (L1)
$(C_2, 0, \texttt{compute}, 2) \leadsto_{C_2} (C_2, 2, \texttt{sigma}, 2)$ (L2)

Following in the similar style of backward inference, the following operationally valid (see Proposition 4.1) partial annotation is obtained:

`program`: $(\downarrow\downarrow\uparrow\uparrow)$, `compute`: $(\downarrow\downarrow\uparrow\uparrow)$, `complement`: $(\downarrow\Box\uparrow)$, `member`: $(\Box\Box)$,
`delete`: $(\Box\downarrow\uparrow)$, `sigma`: $(\downarrow\downarrow\Box)$, `sum`: $(\downarrow\uparrow)$

The induced dependences can be illustrated by showing the local dependence graph for the buggy clause C_2 (over the functionally relevant inherited and synthesized positions only):

```
compute(L1,L2,C,C) :- complement(L1,L2,C), sigma(L1,L2,S).
```

Let us see what happens if the user wants the program not only to compute the difference and sum of two lists but even to generate the sublists of *one* input list; that is, the second argument may also be used as output. This objective is indicated with the annotation `program`: $(\downarrow\updownarrow\uparrow\uparrow)$. Notice especially the dual mode on the second position.

The annotation inference starts as for the case above, by analyzing dependences over the functional expressions of the program. The essential difference is revealed in clause C_2. Selecting, as above, $(C_2, 0, \texttt{compute}, 2)$ as the input position for the output position $(C_2, 2, \texttt{sigma}, 2)$ containing `L2` would imply that `L2` in the position $(C_1, 1, \texttt{compute}, 2)$ must be ground on entry to `compute` in clause C_1. This, however, may not be the case since according to the user's annotation $\updownarrow$ for $\texttt{program}_2$, `L2` may be uninstantiated by the query. Therefore we have to select the other still unannotated occurrence of `L2` in C_2 as the input for $(C_2, 2, \texttt{sigma}, 2)$, leading to the annotation `complement`:$(\Box\uparrow\Box)$ and to the local dependence

$(C_2, 1, \texttt{complement}, 2) \leadsto_{C_2} (C_2, 2, \texttt{sigma}, 2)$ (L2)

Notice how the role of $\texttt{complement}_2$ has changed from a neutral one $(\Box)$ into a functionally ground one $(\uparrow)$.

Completing the inference, the final annotation now becomes strikingly different from the previous one:

`program`: $(\downarrow\updownarrow\uparrow\uparrow)$, `compute`: $(\downarrow\Box\uparrow\uparrow)$, `complement`: $(\downarrow\uparrow\uparrow)$, `member`: $(\uparrow\downarrow)$,
`delete`: $(\Box\downarrow\uparrow)$, `sigma`: $(\downarrow\downarrow\Box)$, `sum`: $(\downarrow\uparrow)$

Note how extensive the inferenced annotation is now when compared with the modest specification by the user. This is because the dual $\updownarrow$ mode for $\texttt{program}_2$ has made it necessary to ensure the program's operational correctness with less information about groundness, that is, irrespective of the input/output mode of $\texttt{program}_2$ in a query. Notice also that the inference algorithm has generated an unconventional annotation `member`: $(\uparrow\downarrow)$ instead of the normally applied $(\downarrow\downarrow)$ (or $(\Box\Box)$ as in the first case above), due to a possible sublist generation mode of the program. Finally, taking into account the tester's view that the two first arguments of `delete` are used as input (see Example 3.2), its annotation can be refined into $(\downarrow\downarrow\uparrow)$.

The induced functional dependences in this case can be illustrated by showing the local dependence graph for clause C_2. Notice the differences to the corresponding graph above.

```
compute(L1,L2,C,C) :- complement(L1,L2,C), sigma(L1,L2,S).
```

The program dependence graph, modelling the complete data flow within the program (taking into account even the modes $\updownarrow$ and $\Box$), is depicted below. For simplicity, only the clauses C_1 and C_2 are included in the figure. The transition edges are denoted with arrows between the clauses.

```
(C1)    program(L1,L2,C,S) :- compute(L1,L2,C,S), print(C),  print(S)

(C2)    compute(L1,L2,C,C) :- complement(L1,L2,C), sigma(L1,L2,S)
```

$\Box$

The presented slicing technique is based on program dependence graphs that are valid for each execution of the program and, as a consequence, for each proof tree. Another possibility is to produce the slice in terms of a proof tree which is transitively annotated with local dependences, without using the program dependence graph at all [PGH94]. In that case the slices will be more precise, but their computation takes more time.

5 Integrated Debugging, Testing and Slicing

In Chapter 3 we have discussed how the algorithmic debugging process can be improved by integrating category partition testing into it. An additional technique for reducing the number of user interactions is static slicing, presented in Chapter 4. If it can be assured that the visible symptom of a bug is not affected by an atom in the proof tree, it is irrelevant to consider its correctness. Such redundant atoms are removed from the proof tree by slicing it with respect to the program dependence graph. The resulting slice contains only those atoms that are necessary to diagnose against the indicated erroneous output values.

When integrating slicing into our method, the algorithmic debugger (see Section 3.3) gets as input a slice of the proof tree T instead of the complete tree. Hence, the main Algorithm 3.2 is revised as follows:

Algorithm 5.2 [IDTS]

Input: $< p, x, y >$: an atom not in M;
S: a slice over the proof tree T with respect to an erroneous output position γ in y.
Output: r: an erroneous clause instance.

procedure IDTS($< p, x, y >, S, r$);
begin
Debug($< p, x, y >, S, r, \mathit{found}$);
...
/* as in Algorithm 3.2 */
...
end IDTS.

The single-stepping algorithm *fp* is not affected by the slicing optimization because it still has to traverse the whole proof tree T. Our basic debugger, *Debug*, however, traverses only a slice S when extracting an erroneous clause from the program. Hence, Algorithm 3.1 is revised to consider only those nodes of the complete proof tree T that are also included in the slice S:

Algorithm 5.1 [Algorithmic debugging + CPM-testing + Slicing]

Input: $< p, x, y >$: an atom not in M;
S: a slice over the proof tree T.
Output: q: erroneous clause instance;
found: *true* if q has been identified,
false otherwise.

procedure Debug($< p, x, y >, S, q, found$);
begin
let $(< p_1, x_1, y_1 >, \ldots, < p_n, x_n, y_n >)$ be the top-level trace of $< p, x, y >$;
found:= *false*; $q :=$ *empty*; i:= 1;
while (i $\leq$ n) **and** (**not** *found*) **do**
if $< p_i, x_i, y_i >$ **is in** S **then** Debug($< p_i, x_i, y_i >, S, q, found$); /* changed */
i:= i+1;
end while;
...
/* as in Algorithm 3.1 */
...
end Debug.

There is still one nuance to take into account. While the slice includes all such atoms that may affect the erroneous output values, it may in some cases miss the source of the bug by throwing away atoms that are in the proof tree even though they should have *failed*. In other words, the reason for erroneous results may also be an invalid *structure* of the proof tree. Notice that this does not happen e.g. in imperative languages that do not advocate "failure" as a computational primitive. The following example illustrates the phenomenon.

Example 5.1 Consider the following (buggy) program that should invert its input bit `X` to the output bit `Y`:

```
p(X,Y) :- even(X), Y=1.
p(X,Y) :- odd(X), Y=0.
even(1).              /* Bug; should be:  even(0).  */
odd(1).
```

Now the query `?- p(1,Y)` produces the result `Y=1`, even though `p(1,1)` is not in the intended model of the program. When slicing the proof tree for this query with respect to the erroneous output value on `Y`, we obtain the following tree. The removed subtree is indicated by not connecting it with its father.

```
          p(1,1)
              \
    even(1)    1=1
```

The faulty atom `even(1)` has been removed from the proof tree, because the variables `X` and `Y` are independent in the first clause (under any annotation). Hence, a standard algorithmic debugger would not find the instance of the buggy clause, but only its ancestor. □

To cope with the situation described above, the debugging algorithm has to take into account two possible cases when it enters the root of the proof tree: (1) The clause corresponding to the top–level subtree is incorrect in which case a buggy clause has been extracted; or (2) The clause is correct, in which case the optimizing IDTS method has not succeeded in locating the bug. The latter case may occur not only in a slicing situation described in Example 5.1, but also in connection with inconsistent test frames (see Algorithm 3.2 in Section 3.3). In these extreme cases an exhaustive algorithm [Sha83] is invoked to debug even the proof tree fragment which has been removed by the slicer.

Example 5.2 Consider the buggy program presented in Example 2.1 and statically analyzed with two alternative annotations in Example 4.1. When submitting the query

```
?- program([1,2],[2],C,S).
```

the answer `C=[1]; S=[1]` is obtained. The output value of `S` is incorrect, and the integrated algorithmic debugger is consulted.

Before the actual interactive diagnosis process is started, the proof tree for the faulty result is sliced according to the transitive data dependences on the erroneous fourth argument position of `program`. From the program dependence graph (see Example 4.1) we see that there is no data flow from the subtree for `sigma` to the erroneous position; this holds for both presented annotations. (This is because the bug in clause C_2 has lost the intended data flow.) Therefore the whole subtree for `sigma` is excluded from the slice and will not be subject to the integrated IDTS debugging process. Assuming the existence of a test database (see Example 3.3), the number of questions to the user will be reduced to *five*, a significant improvement over the original 11 questions. Of course, a more extensive test database would in the optimal case make user interaction completely useless. □

The idea to include an efficient algorithmic debugger (such as IDTS) as an integral part of an incremental inductive learning system has been proposed already in the seminal work of Shapiro [Sha83]. Shapiro used Gold's identification in the limit approach for learning [Gol67]. This learning algorithm works as follows: The input for the learning session is a queue of samples. After the n:th step the developed program is consistent at least for the first n samples. In the $n+1$:th step of the algorithm the first sample (denoted by e) in the queue is extracted for which the program is inconsistent. After that, an algorithmic debugger is called to identify the clause which is responsible for the inconsistency. Then this clause is dropped from the program, and a refined version is included instead. The transformed program must be consistent at least for the same set of samples as the original one and also for the particular sample e. The learning algorithm terminates when all the samples are consistent with the program.

6 Discussion

We have presented a technique which integrates category partition testing and static program slicing with the algorithmic debugging technique presented in [Sha83]. A similar scheme is followed in the GADT system [FGK91] to diagnose Pascal programs but as far as we know, our integrated method is unique in the context of logic programming.

The IDTS system improves Shapiro's original algorithmic debugging method by reducing the number of questions to the oracle. The validity of results of a procedure call is not asked from the oracle if it can be inferred from a CPM test configuration. Another improvement is that only the relevant program execution paths that may have affected the value of an incorrect output are analyzed.

The category partition testing part of IDTS can be used if an initial test configuration has been defined. As a drawback, this initial test configuration can be considered as an extra knowledge for the debugger (and for the incremental inductive learner). However, in the learning process we usually have a preliminary assumption about the excepted behavior of a predicate to be learnt. From this point of view, a category partition specification can be seen as a higher–order description of the program, with a close resemblance to *integrity constraints* [DeB92].

For instance, let us suppose that we would like to infer the predicate `delete`. We can assume that this predicate has two input arguments and one output argument. The first argument is a single element and the other two are lists. From this "extra knowledge" the test specification given in Example 3.2 could be automatically produced. Using this test specification the IDTS system can generate an initial test configuration for the predicate `delete`. Hence, this approach removes the tedious manual task of preparing an initial test configuration. Of course, the initial configuration introduces *undefined* as the evaluation value for each item in it.

A new contribution of IDTS with respect to category partition testing is that the test database is not only consulted but it can also be developed during the debugging process. In the current version of the system, the user is asked to improve the CPM test specification if an inconsistent test frame is recognized during debugging. We are currently working on a method which assists the user in a semi-automatic way to develop a consistent test specification.

Another novel technique within IDTS is static slicing of logic programs. The approach applies well–known techniques of attribute grammars to analyze the directed data–flow within a program, thus making it possible to reorganize the search space of the debugging algorithm to concentrate on the suspect part of the program. To our knowledge, static slicing has not been previously applied on logic (Prolog) programs. The rational debugger presented in [Per86] has similar goals, but there slicing is done dynamically in connection with a sophisticated unification algorithm and no dependence graphs are constructed.

The basic single-stepping debugging algorithm and the category partition component of the IDTS system have already been implemented. The implementation of the static slicing is currently under development. The final step will be to combine these techniques into an integrated whole, and to extend the method to the whole inductive learning process.

In the last few years very intensive research activities have emerged in the field of inductive logic programming (ILP), and several systems have been developed [Mug92]. In the ILP context, e.g. [ADB93] has a similar goal than IDTS to avoid the questions to the oracle consulted in Shapiro's technique. After the complete implementation of IDTS we would like to investigate the possiblity of integrating our method with some other ILP systems. Another topic of future development is to apply slicing more extensively, e.g. in incremental testing as proposed in [BaH93].

Acknowledgements

The slicing technique is based on the joint work of the first author with Johan Boye and Jan Małuszyński at the Department of Computer and Information Science, Linköping University, Sweden, with a financial support by the Academy of Finland and by NorFA. The work of the second and third author has been supported by the ESPRIT project BRA 6020 and by the OTKA grant 501. The comments by the referees have been helpful for improving the presentation.

References

[ADB93] Adé H., De Raedt L., Bruynooghe M.: Theory Revision. In: *Proc. 3rd Int. Workshop on Inductive Logic Programming* (ILP'93), Bled, Slovenia, 1993. Technical Report IJS-DP-6707, J. Stefan Institute, 1993, 179–192.

[AlM91] Alblas H., Melichar B. (eds.): *Proc. Int. Summer School on Attribute Grammars, Applications and Systems* (SAGA), Prague, 1991. LNCS 545, Springer-Verlag, 1991.

[BaH93] Bates S., Horwitz S.: Incremental Program Testing Using Program Dependence Graphs. In: *Conf. Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South California, 1993. ACM Press, 1993, 384–396.

[Boc76] Bochmann G.V.: Semantic Evaluation from Left to Right. *Communications of the ACM* 19, 2, 1976, 55–62.

[Boy93] Boye J.: Avoiding Dynamic Delays in Functional Logic Programs. In: *Proc. 5th Int. Symposium on Programming Language Implementation and Logic Programming*, (PLILP'93), Tallinn, 1993. LNCS 714, Springer-Verlag, 1993, 12–27.

[BPM93a] Boye J., Paakki J., Małuszyński J.: Dependency-Based Groundness Analysis of Functional Logic Programs. Research Report LiTH-IDA-R-93-20, Department of Computer and Information Science, Linköping University, 1993.

[BPM93b] Boye J., Paakki J., Małuszyński J.: Synthesis of Directionality Information for Functional Logic Programs. In: *Proc. 3rd Int. Workshop on Static Analysis*, Padova, 1993. LNCS 724, Springer-Verlag, 1993, 165–177.

[DeB92] De Raedt L., Bruynooghe M.: Belief Updating from Integrity Constraints and Queries. *Artificial Intelligence* 53, 2–3, 1992, 291–307.

[DeM85] Deransart P., Małuszyński J.: Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming* 2, 1985, 119–156.

[DeM93] Deransart P., Małuszyński J.: *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[DJL88] Deransart P., Jourdan M., Lorho B.: *Attribute Grammars – Definitions, Systems and Bibliography.* LNCS 323, Springer-Verlag, 1988.

[FGK91] Fritzson P., Gyimóthy T., Kamkar M., Shahmeri N.: Generalized Algorithmic Debugging and Testing. In: *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, 1991. *ACM SIGPLAN Notices* 26, 6, 1991, 317–326.

[Gol67] Gold E.M.: Language Identification in the Limit. *Information and Control* 10, 1967, 447–474.

[HBR90] Horwitz S., Reps T., Binkley D.: Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems* 12, 1, 1990, 26–61.

[HGA93] Horváth T., Gyimóthy T., Alexin Z., Kocsis F.: Interactive Diagnosis and Testing of Logic Programs. In: *Proc. 3rd Symposium on Programming Languages and Software Tools* (M. Tombak, ed.), Kääriku, Estonia, 1993. Department of Computer Science, University of Tartu, 1993, 34–46.

[HoR92] Horwitz S., Reps T.: The Use of Program Dependence Graphs in Software Engineering. In: *Proc. 14th Int. Conference on Software Engineering*, Melbourne, 1992. IEEE Computer Society Press, 1992, 392–410.

[Kam93] Kamkar M.: Interprocedural Dynamic Slicing with Applications to Debugging and Testing. Linköping Studies in Science and Technology – Dissertations No. 297, Department of Computer and Information Science, Linköping University, 1993.

[Knu68] Knuth D.E.: Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 1968, 127–145.

[KoL90] Korel B., Laski J.: Dynamic Slicing of Computer Programs. *The Journal of Systems and Software* 13, 3, 1990, 187–195.

[LRS74] Lewis P.M., Rosenkrantz D.J., Stearns R.E.: Attributed Translations. *Journal of Computer and System Sciences* 9, 1974, 279–307.

[Mał91] Małuszyński J.: Attribute Grammars and Logic Programs: A Comparison of Concepts. In: [AlM91], 330–357.

[Mug92] Muggleton S. (ed.): *Inductive Logic Programming.* Academic Press, 1992.

[OsB88] Ostrand T.J., Balker M. J.: The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM* 31, 6, 1988, 676–686.

[Paa93] Paakki J.: Multi-Pass Evaluation of Functional Logic Programs. Research Report LiTH-IDA-R-93-02, Department of Computer and Information Science, Linköping University, 1993.

[Paa94] Paakki J.: Multi-Pass Execution of Functional Logic Programs. In: *Conf. Record of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM Press, 1994, 361–374.

[Per86] Pereira L.M.: Rational Debugging in Logic Programming. In: *Proc. 3rd Int. Conference on Logic Programming*, London, 1986. LNCS 225, Springer-Verlag, 1986, 203–210.

[PGH94] Paakki J., Gyimóthy T., Horváth T.: An Integrated Method for Algorithmic Debugging of Logic Programs. Submitted for publication.

[Sha83] Shapiro E.: *Algorithmic Program Debugging.* The MIT Press, 1983.

[SzG91] Szücs R., Gyimóthy T.: T-GEN: An Extended Version of the Category Partition Testing Method. In: *Proc. 2nd Symposium on Programming Languages and Software Tools* (K.Koskimies, K.-J.Räihä, eds.), Pirkkala, Finland, 1991. Report A-1991-5, Department of Computer Science, University of Tampere, 1991, 70–77.

[Wei84] Weiser M.: Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4, 1984, 352–357.