DEPENDENCY ANALYSIS AND LEARNING METHODS OF DECLARATIVE LANGUAGES

PhD Thesis Summary

Author: Gyöngyi Szilágyi
Thesis Supervisor: Dr. Tibor Gyimóthy

Doctoral School in Mathematics and Computer Science
PhD Programme in Informatics
University of Szeged
2003

Abstract

The present thesis summarizes the scientific results of the author of the PhD dissertation ("Dependency Analysis and Learning Methods of Declarative Lanuages").
Besides the **imperative** *programming paradigm an other main type is the* **declarative** *paradigm. Declarative Programming requires a more descriptive style than imperative ones. The programmer must know what relationship holds between the various entities.*
The dissertation concentrates on three declarative paradigms, namely **Logic Programming (LP)** *[20,17],* **Constraint Logic Programming (CLP)** *[18,12] and* **Attribute Grammars (AG)** *[1]. There is a close relationship between Attribute Grammars (AG) and Logic Programming (LP) [6], and Logic Programming (LP) can be viewed as a special case of Constraint Logic Programming (CLP) [12].*
One of the results of the dissertation is the slicing of Logic and Constraint Logic Programs. **Slicing** *[11] is a program analysis technique which facilitates understanding of data flow and debugging. 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 may be affected by the value of the variable.*
In the dissertation a **dynamic slicing algorithm for Logic Programs** *[23,10,9] is defined augmenting the data flow analysis with control flow dependences to help one locate the connected components of the program and the source of a bug. A general slice definition is provided which is valid for the success and failure branches of the SLD-tree, as well. The extension of the data flow analysis to the failure branches of the SLD-tree helps improve the existing debugging techniques, it can help one detect dead code, and is useful in program maintenance.*
Constraint Logic Programming (CLP) *[12,18] is a fusion of two declarative paradigms, namely Constraint Solving and Logic Programming. In the dissertation we lay the theoretical foundations for the* **slicing of Constraint Logic Programs** *[24,25,26] which provide a basis for defining techniques based on variable sharing.*
Inductive Constraint Logic Programming (ICLP) *is a research topic that combines the theory and results of CLP [12,18] and Machine (Inductive) Learning (ILP) [19,16]. In the dissertation* **a specialization technique is provided for learning of CLP programs** *[28]. The specialization method combines unfolding, clause removal and, as an improvement of the algorithm, slicing. We lay the theoretical foundations for specializing CLP programs, state the associated theorems, and prove that the defined unfolding transformation preserves the operational and logical semantics of CLP programs.*
Attribute Grammars (AG) *[29,15,1] are a generalization of the concept of Context-Free Grammars [13]. In the framework of compilation oriented language implementation, Attribute Grammars (AG) are the most widely applied semantic formalism. Efficient techniques for learning Attribute Grammars can help in finding good definitions of AGs, which usually require a lot of effort.*
Based on the correspondence of Attribute Grammars and Logic Programs [6] some of the learning technique developed for Logic Programs (ILP) [19,16] could be applied to AGs. We introduced an AG based description language in ILP and defined **a parallel method for learning semantic functions of Attribute Grammars** *(AGs) [27]. The parallelism can help to provide a more efficient technique than the sequential one, in both execution time and interactions needed.*

1. INTRODUCTION

In the following we provide some basic definitions needed to summarise our results. The others can be found in the Dissertation.

Definition 1 Constraint Logic Program

A **constraint domain** *is a pair* $\langle L, D \rangle$ *where* L *is a first order language over an alphabet of variables, predicate symbols (including equality), and function symbols (including constants).* D *is a set (domain). All function symbols of* L *are given a fixed interpretation on* D*. A* **constant** *is a function symbol of arity 0.*
A **term** *is a variable, a constant or a n-ary* $(n > 0)$ *function symbol followed by a bracketed n-tuple of terms (The latter is called a compound term).*

The **predicate symbols** *of L are divided into the following two disjoint sets:*

1. *constraint predicates Σ, which are given a fixed interpretation in D. These include the symbol $=$, interpreted as identity.*
2. *defined predicates Π, which may occur in program clause heads and for which the user has an intended interpretation on D.*

A **defined atom** *is a formula of the form $p(t_1, \cdots, t_n)$ where p is an n-ary defined predicate ($p \in \Pi$) and $t_1, \cdots, t_n$ are terms.*

Constraints atoms *are formulae constructed with some* constraint predicates *with a predefined interpretation. A typical example of a constraint is a linear arithmetic equation or inequality with rational coefficients where the constraint predicate used is the equality symbol interpreted over rational numbers, e.g. $X - Y = 1$. The variables of a constraint range over the domain of interpretation.*

A **clause** *is a formula of the form $h : -b_1, ..., b_n$, $n \geq 0$, where $h, b_1, ..., b_n$ are* atomic formulae. *The predicates used to construct $b_1, ..., b_n$ are either constraint predicates or defined predicates. The predicate of h is a defined predicate.*

A **goal** *is a clause without h. A* **fact** *is a clause $h \leftarrow c_1, \cdots, c_n$ where $c_1, \cdots, c_n$ are constraints.*

A **constraint logic program** *is a set of* clauses.

Logic Programs only use defined predicates.

Definition 2 Skeleton

A skeleton for a program P is a labelled ordered tree:

- *with the root labelled by a goal clause and*
- *with the nodes labelled by clause instances of the program $< c, \sigma >$; some leaves may instead be labelled "?", in which case they are called* incomplete nodes.
- *Each non-leaf node has as many children as the non-constraint atoms of its body.*
- *The head predicate of the i-th child of a node is the same as the predicate of the i-th non-constraint body atom of the clause labelling the node.*

Definition 3 The set of constraints of a skeleton

For a given skeleton S the set $C(S)$ of constraints, which will be called the set of constraints of S, *consists of :*

- *the constraints of all clauses labelling the nodes of S*
- *all equations $\vec{x} = \vec{y}$ where $\vec{x}$ are the arguments of the i-th body atom of the clause labelling a node n of S, and $\vec{y}$ are the arguments of the head atom of the clause labeling the i-th child of n. (No equation is created if the i-th child of n is an incomplete node.)*

A **derivation tree** for a program P is a skeleton for P whose set of constraints is satisfiable. If the skeleton is complete (i.e. it has no incomplete node) the derivation tree is called a **proof tree**.

In order to properly present the slicing techniques we need to refer to **program positions** and to **derivation tree positions**. A slice is defined with respect to some particular occurrence of a variable (in a program or derivation tree), and positions are used to identify these occurrences. The set of all tree positions of a derivation tree T will be denoted by $Pos(T)$.

2. DATA FLOW ANALYSIS OF LOGIC PROGRAMS

The related papers are [23,10,9].

Data flow analysis of Logic Programs (LP [20,17]) plays an important role in debugging, testing, program maintenance, and so on . **Slicing** [11]is a program analysis technique originally developed for imperative languages. It can be applied in a number of software engineering tasks, is a natural tool for debugging, is useful in incremental testing, and can help one detect dead code or find parallelism in programs. 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 (backward slice) or may be affected by the value of the variable (forward slice). Data flow in logic programs is not explicit, and for this reason the concept of a slice and the slicing techniques of imperative languages are not directly applicable. Moreover, implicit data flow makes the understanding of program behavior rather difficult. Thus program analysis tools explaining data flow to the user are of great practical importance. One of the results of our research is the extension of the the scope and optimality of previous algorithmic debugging techniques of Prolog programs [22] using slicing techniques based on a depenence graph. We provide **a dynamic slicing algorithm** augmenting the data flow analysis with control flow dependences to help one locate the connected components of a program and the source of a bug included in the program.
A general slice definition is provided which is valid for the success and failure branches of the SLD-tree, as well. The extension of data flow analysis for the failure branches of the SLD-tree helps to improve the existing debugging techniques, it can help detect dead code, and is useful in program maintenance.
A tool was developed for debugging Prolog programs which also handles the specific programming techniques.

Consider the following example.

Example 1 The Data Flow slice is not enough to find the source of a bug

The buggy program is:	*The correct program should be:*
1. p(A,X) :- q(A,X).	*1. p(A,X) :- q(A,X).*
2. q(A,X) :- $\mathbf{A > 0}$*, X is 2.*	*2. q(A,X) :-* $\mathbf{A = 0}$*, X is 2.*
3. q(A,X) :- X is 3.	*3. q(A,X) :- X is 3.*

Executing this program for the goal $p(0, X)$ the given solution is $X = 3$, while we expect $X = 2$. So a bug must be present in the program somewhere.
Creating the dynamic data flow slice for an instance of X, it does not contain the buggy predicate $A > 0$ because X does not exactly depend on the predicates of clause 2, there being only control dependences between them. This means that if $A > 0$ had been evaluated differently it could have affected the solution of X. Our new slicing approach contains the buggy predicate $A > 0$.

2.1. The Augmented SLD-tree of Logic Programs, Skeleton(n) and Proof Tree Dependence Graph

The derivation of a program P for a goal can be represented by a tree called SLD-tree. Each branch of the SLD-tree [21] is a derivation of a program for a goal. Branches corresponding to successful derivations are called *success branches*, while branches of the infinite derivations are called *infinite branches*; those corresponding to failed derivations are called *failure branches*.
An SLD-tree may have many failed branches and very few or just one success branch. Control information supplied by the user may prevent the interpreter from constructing failed branches. To control the search the concept of *cut*(!) is introduced in Prolog. *Cut* has the following effect: after success of "!" no backtracking to the literals in the left-hand part is possible. Denote this part of the SLD-tree by $cut(W)$. However, in the right-hand part execution goes on as usual.
We add these pieces of information to the SLD-tree to get the so called *Augmented SLD-tree*. Our dynamic slicing algorithm is defined in this structure.
The SLD-tree representation is unsuitable for representing the data flow information of a logic program (for a given goal). The structure $Skeleton(n)$ is used to represent this information, where n identifies a leaf node of the SLD-tree.
There is an one-to-one correspondence between the nodes belonging to one branch of the SLD-tree (T) (identified by n) and the nodes of the corresponding $Skeleton(n)$ (denoted by S). This correspondence is

expressed by the map $\phi : nodes(S) \rightarrow nodes(T)$.
The definition of Skeleton(n) is augmented with groundness information.

We would like to represent the data flow of a derivation tree. In a logic program data can be transferred in two ways: firstly from one clause to another via unification, and secondly within a clause multiple occurrence of variables result in data dependences [4,5]. The following definition reflects these conditions.

Definition 4 Proof Tree Dependence Graph (PTDG: $T_{g,n} = (Pos(S), \sim_T)$)
Let T be an SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T and S the Skeleton(n), $\beta, \delta \in Pos(S)$.

- *The nodes of PTDG are the elements of $Pos(S)$.*
- *$\beta \sim_T \delta$ iff one of the following conditions holds:*
  1. *β and δ have common variable in their variable set V (***local edge***)*
  2. *the predicate of δ was unified with the predicate of β, and β and δ are both the k-th argument position of their predicate (***transition edge***).*

It follows directly from the definition that the dependence graph is constructed only for one branch of the SLD-Tree (identified by n), for Skeleton(n). But of course we can construct a PTDG for every Skeleton(n) (n is a leaf node of T), that is for every branch of the SLD-tree T.

This graph is further extended with directionality information ($\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$).

2.2. General Data Flow Slice and Debug SLice

Thesis 1

Below a *general slice definition* is given. The definition identifies those argument positions upon which a given argument position depends.

Definition 5 Slice($T_{g,n}, \alpha$)
Let P be a logic program, T a SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T, S Skeleton(n) and $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$ the corresponding Directed Proof Tree Dependence Graph. Let $\alpha \in Pos(S)$.
A **slice** *($T_{g,n}, \alpha$) over $\vec{T}_{g,n}$ with respect to α*

- *is a subgraph of $\vec{T}_{g,n}$*
- *a node $\beta \in Pos(S)$ is in the slice iff $\beta \rightarrow_T^* \alpha$*

This is a general data flow slice definition that is valid for one derivation path of the SLD-tree which may be a failed branch.

To create the Debug slice we specify, as a first step, the **Potentially Dependent Predicates Set** *(PDPS).*

Definition 6 Potentially Dependent Predicate (PDPS)
Let P be a logic program, T the Trace-tree for the goal g. A leftmost (selected) predicate in a node of T is in the **Potentially Dependent Predicate Set (PDPS)** *if it actually did not affect the value of an argument of a predicate in the success branch of T, but could have affected it had its boolean outcome been different.*

The following theorem identify those predicates which satisfy this condition.

Theorem 1 The Potentially Dependent Predicate Set
Let P be a logic program, T the Trace-tree for the goal g. Then
PDPS= $\{$The predicates of the success branch of $T\} \cup \{$The predicates of the failed leaves of $T\}$.

Definition 7 Debug Slice
The **Debug slice** *of an Augmented SLD-tree for a goal g is the following set:*

$Debug\ slice = PDPS \cup Data\_Flow\_of\_PDPS \cup Cut(W)\_Set =$

$$
\begin{aligned}
= \quad & \{\ \textit{The predicates of the success branch of } T\} \\
& \bigcup \{\ \textit{The predicates of the failed leaves of } T\} \\
& \bigcup \phi(\cup_{n,\alpha}\{k \in nodes(S) \mid k \textit{ has at least one head argument position in} \\
& \qquad slice(T_{g,n}, \alpha),\ \alpha \textit{ is an argument position of } p,\ n \textit{ is a failed leaf of } T\}) \\
& \bigcup (\cup_{Mark}\ \{p \in nodes(T) \mid p \textit{ is a leftmost predicate on the path from the node} \\
& \qquad \textit{whose leftmost goal is } cut(Mark) \textit{ at the node identified by } Mark\}\ )
\end{aligned}
$$

□

The idea behind the definition of the Debug Slice comes from imperative languages, which is called there to "relevant slice" [7]. The relevant slice could be used to find such a bug instance, which could not be identified by examining the data flow slice alone. In this case the data flow slice is augmented with control dependences as well.
The searching strategy is controlled by **(1)** the boolean outcomes of the predicates (i.e. the Interpreter continues the search in the case of success of a predicate, backtracks at failed predicates), and **(2)** a special built in predicate *cut*(!) (i.e. after success of *cut* no backtracking to the literals in the left-hand side is possible). Our Debug slice deals with these two kinds of main control effects, and it is further extended by the data flow slice of those predicates which take part in the control flow **(3)**. The reason for this extension is that the cause of a failure of a predicate (p) could be a wrong value which reached p via the data flow.

So an informal definition for the Debug slice is the following.
Let P be a logic program, T be the Trace-tree for the goal g. The **Debug slice of T** consists of the following predicates:

1. The predicates of the Potentially Dependent Predicate Set (PDPS)
2. The predicates specified by the data flow of the predicates of PDPS
3. The predicates that belong to some $cut(W)$ of T

Certain types of bugs were found during testing by a prototype implementation which were missed by the data-flow slice but were identified using the Debug slice method, as they appeared in the failure branches of the SLD-tree. These types include cases when a cut is mis-placed, a failed predicate is mis-printed (its name or arity) or a condition ($<, >, =$) has failed, or a wrong data value has reached the failed node. So in the data-flow from the root to the failed node, a wrong constant value, a mis-printed predicate or a failed condition has appeared.

3. DATA FLOW ANALYSIS OF CONSTRAINT LOGIC PROGRAMS

The related papers are: [24,25,26].

One other significant result of the dissertation is the **slicing of Constraint Logic Programs**.
This part of the work formulates **declarative notions of slice** suitable for CLP. (The problem of finding minimal slices may be undecidable in general, since satisfiability may be undecidable.) These definitions provide a basis for defining **slicing techniques** (both dynamic and static) **based on variable sharing**. The techniques are further extended by using groundness information.
A prototype dynamic slicer of CLP programs has now been implemented.

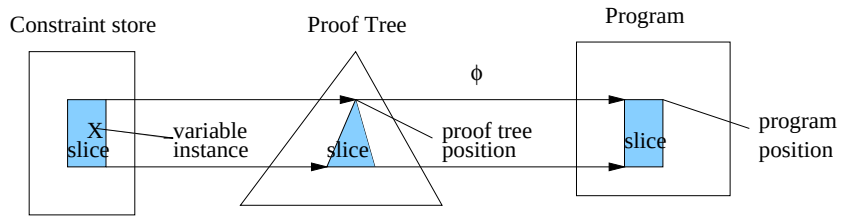

Fig. 1. A slice of a constraint set, a proof tree and a program.

Given a variable X in a CLP program we would like to find a fragment of the program that may affect the value of X (see Figure 1).
We first define a concept of slice for a set of constraints, which is then used to define slices of *derivation trees* representing states of CLP computations. Afterwards, we define slices of a program in terms of the slices of its derivation trees.
To formalize this thesis we use the following definitions and notations.

Let C be a set of constraints. A derivation tree T is a skeleton with a set of constraints $C(T)$. The variables of $C(T)$ originate from positions of T. Let $\mathcal{P}$ be a set of positions of T, i.e. $\mathcal{P} \subseteq Pos(T)$. Then $\Psi(\mathcal{P})$ identifies the variables of $C(T)$ with occurrences originating from positions in $\mathcal{P}$. We denote the set of all constraints of $C(T)$ that include these variables by $C_{\mathcal{P}}$.

Every position of a derivation tree T is a (renamed) copy of a program position or of a goal position. This provides a natural map Φ_T of the positions of T into program positions and goal positions. Corresponding to this definition of Φ_T for a program position q, the set $\Phi_T^{-1}(q)$ contains those proof tree positions such that if $r \in \Phi_T^{-1}(q)$ then $\Phi_T(r) = q$.

Definition 8 Solution of a Constraint set
The binding of a variable X to a value v is said to be a **solution** *of C with respect to X iff there exists a valuation ν such that $\nu(X) = v$ and ν satisfies C. The set of all solutions of C with respect to X will be denoted by $Sol(X, C)$.*

Thesis 2

2.a A precise declarative formulation of the slicing problem for Constraint Logic Programs, Proof trees and there Sets of Constraints.

Definition 9 Slice of a satisfiable constraint set C
A **slice of a satisfiable constraint set** C *with respect to X is a subset $S \subseteq C$ such that $Sol(X, S) = Sol(X, C)$.*

Definition 9 implicitly gives a notion of **minimal slice** with respect to X.

Definition 10 A minimal slice of C
A minimal slice of C *is a slice S of C with respect to X such that if we further reduce S to S', then $Sol(X, S')$ will be different from $Sol(X, C)$.*

Notice that the whole set C is a slice of itself, any superset of a slice is also a slice, and that the definition does not provide any hint about how to find a minimal slice. The minimal slice may not be unique. In general, the problem of finding minimal slices may be undecidable as satisfiability may be undecidable.
We now formulate the slicing problem for derivation trees.

Definition 11 A slice of a derivation tree
A **slice of a derivation tree** *T with respect to a variable position of X is any subset $\mathcal{P}$ of the positions of T such that $C_{\mathcal{P}}$ is a slice of $C(T)$ with respect to X.*

Lastly we define the notion of a CLP program slice with respect to a variable position.

Definition 12 A slice of a CLP program
A slice of a CLP program *P with respect to a program position q is any set S of positions of P such that for every derivation tree T whenever its position r is in $\Phi_T^{-1}(q)$, there exists a slice Q of T with respect to r such that $\Phi_T(Q) \subseteq S$.*

2.b Data flow based automatic slicing techniques for the construction of slices of CLP programs, Proof Trees and Constraint Sets

Generally it is undecidable whether a subset of a set of constraints is a slice. We now present a simple and sufficient condition for a "syntactical" approach to slicing constraint stores, proof trees and programs. We use variable sharing between constraints as a basis for the slicing of sets of constraints.

Definition 13 Explicit Dependence of Constraint Stores
Let C be a set of constraints, and $vars(C)$ be the set of all variables occurring in the constraints in C. Let X, Y be variables in $vars(C)$. X is said to **depend explicitly** *on Y iff both occur in a constraint c in C.*

Definition 14 Dependency Relation of Constraint Stores
A **dependency relation** *on $vars(C)$ is the transitive closure of the explicit dependency relation. The dependency relation on $vars(C)$ will be denoted by dep_C.*

Note that dep_C is an equivalence relation on $vars(C)$. We map any equivalence class $[X]_{dep_C}$ to the subset C_X of C which consists of all constraints that include variables in $[X]_{dep_C}$.

Definition 15 Direct Dependency Relation of a Derivation tree
Let T be a derivation tree, $\alpha, \beta \in Pos(T)$, and let $\sim_T$ denote the direct dependency relation on $Pos(T)$. Then $\alpha \sim_T \beta$ if and only if one of the following conditions holds:

1. *α and β are positions in an occurrence of a clause constraint (constraint edge).*
2. *α and β are positions in a node equation (transition edge).*
3. *α and β are positions in an occurrence of a term (functor edge).*
4. *α and β share a variable (local edge).*

The transitive closure $\sim_T^*$ of the direct dependency relation will be called the *dependency relation* on $Pos(T)$. Thus $\sim_T^*$ is an equivalence relation.

Recall that each position of a derivation tree T "originates" from a position of the selected program P. This is formally expressed by the mapping $\Phi : Pos(T) \rightarrow Pos(P)$.

Definition 16 Direct Dependency Relation of a Program
Let P be a CLP program, $\alpha, \beta \in Pos(T)$. Let $\sim_P$ denote the direct dependency relation on $Pos(P)$. Then $\alpha \sim_P \beta$ iff at least one of the following conditions holds:

1. *α and β are positions of the same constraint (constraint edge).*
2. *α is a position of the head atom of a clause c and β is a position of a body atom of a clause d and both atoms have the same predicate symbol (transition edge).*
3. *α and β belong to the same argument of a function (functor edge).*
4. *α and β are in the same clause and have a common variable (local edge).*

Comparing the definitions of $\sim_T$ and $\sim_P$ one can check that whenever
$\alpha \sim_T \beta$ in some tree T of P then $\Phi(\alpha) \sim_P \Phi(\beta)$ as well. Consequently, for any proof tree T $(\alpha \sim_T^* \beta) \Rightarrow \Phi(\alpha) \sim_P^* \Phi(\beta)$. The transitive closure $\sim_P^*$ is an equivalence relation on $Pos(P)$.

Directionality information is introduced into the Proof Tree Dependency Graph
$(T_{DG} = (Pos(T), \rightarrow_T))$ to make the slice more precise.

2.c The proofs that the automatic slice construction methods satisfy the conditions of the declarative slice definitions

The following theorem states that C_X satisfies the declarative slice definition as well.

Theorem 2 Slice of a Constraint Set
Let C be a satisfiable set of constraints and let $X \in vars(C)$.
Then C_X is a slice of C with respect to X.

The following result relates $\sim_T^*$ to $dep_{C(T)}$.

Theorem 3 The relation of $\sim_T^*$ to $dep_{C(T)}$
Let α be a variable position of a proof tree T and let $\Psi(\{\alpha\}) = \{X\}$.
Then $\Psi([\alpha]_{\sim^*}) = [X]_{dep_{C(T)}}$.

As a corollary we immediately obtain the following Theorem.

Theorem 4 A Slice of a Proof Tree
Let T be a proof tree and let α be a variable position of T. Then $[\alpha]_{\sim^*}$ is a slice of T with respect to α.

So a slice of a proof tree can be obtained by finding the equivalence class of the dependency relation involving a given variable position.
Theorem 4 gives suggestions for a simple slicing technique of derivation trees based on finding equivalence classes of variables in the constraints of a derivation tree.
The following result shows how $\sim_P^*$ can be used for the slicing of P.

Theorem 5 A Slice of a Program
Let P be a CLP program and let β be a position of P. Then $[\beta]_{\sim_P^*}$ is a slice of P with respect to β.

Definition 16 with Theorem 5 give a method for constructing program slices without referring to proof trees.

2.d Two approaches had been proposed to reduce the size of the slices.

We introduced directionality information (which uses groundness information) into the slicing and dealt with the so-called calling context problem.

Theorem 6 Directed Slice of a Proof Tree
$\{\beta | \beta \rightarrow_{T(G)}^* \alpha\}$ is a slice of T with respect to α.

□

3.1. Summary of Thesis 2

2.a We gave a precise, declarative formulation of the slicing problem for Constraint Logic Programs, Proof trees and their Sets of Constraints: **Definitions 9, 11, 12**.

2.b A data flow based automatic slicing technique was used for the construction of
- static slices of CLP programs: **Definition 16**
- dynamic slices of Proof Trees: **Definition 15**
- dynamic slices of Constraint Sets: **Definition 14**.

2.c We proved that the automatic slice construction methods satisfy the conditions of the declarative slice definitions: **Theorems 5, 4, 2**.

2.d Two approaches were proposed to reduce the size of the slices. We introduced directionality information into the slicing (Theorem 6) and dealt with the so-called calling context problem.

A prototype slicing tool has been implemented using static and dynamic slicing techniques. Our experiments with this tool show that the slices obtained were quite precise in some cases, and on average provided a substantial reduction in the program size.

4. LEARNING OF CONSTRAINT LOGIC PROGRAMS

The related paper is [28].

Inductive Constraint Logic Programming (ICLP) is a new research topic that combines the theory and results of Constraint Logic Programming (CLP) [12,18] and Inductive Logic Programming (ILP) [19,16].
Inductive Logic Programming (ILP) refers to a class of machine learning algorithms where the agent learns a first-order theory from examples and background knowledge. The use of first-order logic programs as the underlying representation makes ILP systems more powerful and useful than the conventional propositional machine learning systems, but they are weak in handling numerical data.
CLP languages make logic programs execute very efficiently by focussing on a particular problem domain, and they are capable of handling numerical data, as well.

The dissertation discusses the **specialization of Constraint Logic Programs by applying unfolding** and, as an improvement of the method, by combining unfolding with a slicing technique. The transformation rule for unfolding together with clause removal is a method (called SPECTRE [3]) for the specialization of Logic Programs (LP).
Firstly, we formulate the exact definitions of the basic elements of an unfolding learning algorithm such as the specialization and unfolding transformation of CLP programs giving the semantics of them. We prove that the defined unfolding transformation preserves the operational and logical semantics of CLP programs.
Another result is the exact formalization of the specialization method (CLP_SPEC) and the proof of the correctness of the algorithm based on the logical and operational semantics of CLP programs.
Yet another result is that an improved interactive version of the specialization algorithm has been defined integrating an algorithmic debugging algorithm and a slicing method with a specialization algorithm for CLP programs.
A prototype learner of LP and CLP programs implementing the above ideas is briefly described.
The following definitions have been used to present our results.

Definition 17 *D*-**Interpretation**
Let $\langle L, D\rangle$ be a constraint domain, where L is a first order language over an alphabet of variables, predicate symbols (including equality), and function symbols (including constants). D is some set (domain).
An $\Im$ D-interpretation of the alphabet A of L is the domain D and a mapping that associates
- *each constant $c \in A$ with an element $c_\Im \in D$ which is the same interpretation as c has in D*
- *each n-ary function $f \in A$ with a function $f_\Im : D^n \to D$ which is the same interpretation as f has in D*
- *each n-ary defined predicate $p \in \Pi$ with a relation $p_\Im \subseteq D \times \cdots \times D$*
- *each n-ary constraint predicate $p \in \Sigma$ with the same interpretation as it has in D*

Let $B_D = \{p(\widetilde{d}) \mid p \in \Pi, \widetilde{d} \in D^n\}$.
Then a D-interpretation can be represented as a subset of B_D.

The **top-down operational semantics** of constraint logic programs P can be seen as a transition system of states, tuples $\langle A, C, S\rangle$, where A is a multiset of atoms and constraints, and C and S are multisets of constraints [12]. The constraints C and S are referred to as the constraint store. Intuitively, A is a collection of as-yet-unseen atoms and constraints, C is a collection of constraints playing an active role (they are awake), and S is a collection of constraints playing a passive role (they are asleep).
We will take as given a computation rule that selects a transition type and an appropriate element of A for each state.
An **initial goal** G for execution is represented by the state $\langle G, \emptyset, \emptyset\rangle$. A **derivation** is a sequence of transitions. A state which can not be rewritten is called a **final state**. A derivation is **successful** if it is finite and the final state has the form $\langle \emptyset, C, S\rangle$.
A detailed description of the transitions rules can be found in the dissertation.

Here the specialization problem is formalized in the following way.

Definition 18 The specialization problem
The specialization problem of a Constraint Logic Program with respect to a set of positive examples (E^+) and a set of negative examples (E^-) can be defined as follows:
Given: *a P Constraint Logic Program and two disjoint sets of ground terms (E^+ and E^-).*
The aim is: *to find a P' Constraint Logic Program (the specialization of P with respect to (E^+ and E^-)) such that $M_{P'} \subseteq M_P$, $E^+ \subseteq M_{P'}$ and $M_{P'} \cap E^- = \emptyset$, where M_P denotes a D-model of P.*

We assume that every positive / negative example is a ground instance of a target (defined) predicate G (goal).

Thesis 3

3.a The unfolding transformation

The unfolding transformation is formulated below and two theorems are given which state that the defined unfolding transformation preserves the operational and logical semantics of CLP programs.

Definition 19 The unfolding transformation
Let P be a CLP program with the rules $R_1, \cdots, R_n$, such that
$R_j : h_j \leftarrow b_{j_1}, \cdots, b_{j_{m_j}}, c_j$ ($j = 1, \cdots, n$), where $h_j, b_{j_1}, \cdots, b_{j_{m_j}}$ are defined predicates, c_j denotes the conjunction of the atomic constraints appearing in the body of R_j.
Let $R : h \leftarrow b_1, \cdots, b_m, \cdots, b_k, c$ be a program clause in P, and $\overline{R} = \{R_1, \cdots, R_q\}$ be a set of program clauses given new variables names such that the head of each $R_i \in \overline{R}$ ($i = 1, \cdots, q$) and b_m have the same predicate symbol, and b_m is selected by some computation rule.

Then the program P' after unfolding is :

$P' = Unf(P, R, b_m) =$ argument equations $\qquad body(R_j)$

$= P \setminus \{R\} \cup (\bigcup_{Rj \in \overline{R}} h \leftarrow \overbrace{(\overline{b}_m = \overline{h}_j)}, b_1, \cdots, b_{m-1}, \overbrace{b_{j_1}, \cdots, b_{j_{m_j}}, c_j}, b_{m+1}, \cdots, b_k, c),$

where $\overline{b}_m = \overline{h}_j$ is an abbrevation for the conjunction of argument equations between the corresponding argument positions of b_m and h_j.

We note that only defined predicates can be unfolded and no constraint predicates.

Now we can state our theorem about the operational semantics preservation.

Theorem 7 The operational semantics preservation of the unfolding transformation
Let P be a CLP program with the constraint domain $\langle L, D\rangle$, and with the rules $R_1, \cdots, R_n$ such that $R_j : h_j \leftarrow b_{j_1}, \cdots, b_{j_{m_j}}, c_j$ $(j = 1, \cdots, n)$, where $h_j, b_{j_1}, \cdots, b_{j_{m_j}}$ are defined predicates and c_j denotes the conjunction of the atomic constraints appearing in the body of R_j.
Let $\Im$ be a D-interpretation and let $\exists_{-\widetilde{X}} Q$ denote the existential closure of the formula Q except for the variables $\widetilde{X}$, which remain unquantified.

For every SLD-tree ($P \cup \{G\}$ where $G = p(\widetilde{X})$), for every clause $R \in P$ and for every $b_m \in body(R)$ defined predicate :

$$SS(P) = SS(P'),$$

where $P' = Unf(P, R, b_m)$ and $SS(P) = \{p(\widetilde{X}) \leftarrow c \mid \langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, C', C''\rangle, \Im \models c \longleftrightarrow \exists_{-\widetilde{X}} C' \wedge C''\}$ collects the answer constraints to simple goals $p(\widetilde{X})$ with free variables $\widetilde{X}$.
So an unfolding transformation preserves the operational semantics

Theorem 8 The logical semantics preservation
The unfolding transformation preserves the logical semantics (D-semantics) of CLP programs.

3.b The CLP_SPEC algorithm

The CLP_SPEC algorithm specializes Constraint Logic Programs with respect to positive and negative examples by applying the unfolding transformation rule together with clause removal.

Theorem 9 The correctness of the CLP_SPEC algorithm
The output $P'^{(n)}$ of the CPL_SPEC algorithm is a specialization of P with respect to E^+ and E^- if the reason for the termination of the algorithm is not that no more unfolding steps can be applied. This also means that $P'^{(n)}$ is complete and consistent (i.e. it covers all positive examples and does not cover any negative examples).

3.c The CLP_SPEC_SLICE algorithm

The algorithm CLP_SPEC specializes clauses defining a target predicate by using different strategies for selecting the literal to apply unfolding upon. The identification of a clause to be unfolded is of crucial importance in the effectiveness of the specialization process [2]. If a negative example is covered by the current version of the initial program there is supposedly at least one clause that is responsible for this incorrect covering. In our algorithm CLP_SPEC_SLICE a debugging system combined with slicing technique is used to identify a buggy clause instance. Afterwards this clause is removed from the initial program.

Theorem 10 The correctness of the CLP_SPEC_SLICE algorithm
The output $P'^{(n)}$ of the CPL_SPEC_SLICE algorithm is a specialization of P with respect to E^+ and E^- if the DEB_SLICE algorithm is able to identify a buggy clause (otherwise the CLP_SPEC algorithm can be used to find the hypothesis) and the reason for the program termination is not that no more unfolding steps can be applied.

□

5. LEARNING SEMANTIC FUNCTIONS OF ATTRIBUTE GRAMMARS

The relevant paper is: [27].

In the framework of compilation-oriented language implementation, **Attribute Grammars (AG)** [29,15,1] are the most widely applied semantic formalism. Attribute Grammars are generalizations of the concept of Context-Free Grammars [14]. The formalism of AGs has been widely used for the specification and implementation of programming languages. Since the definition of an AG and its semantic functions may be complex it is very useful to have an efficient tool for inferring semantic rules of AGs from examples. Based on the correspondence between Attribute Grammars and Logic Programs [6] some of the learning technique developed for Logic Programs (ILP) could be applied to AGs. Introducing an AG based description language in ILP leads to the definition of an Attribute Grammar Learner. In the dissertation we present **a parallel method for learning semantic functions of Attribute Grammars** based on an ILP [19,16] approach. The method presented is suitable for S-attributed and for L-attributed grammars, as well. The parallelism can help to reduce the large number of user queries posed during an interactive learning session, so the parallel method is more efficient in both execution time and interaction needed than the sequential one.

Definition 20 S-attributed Grammars
A special class introduced by Knuth [13] is the $S-$attributed grammars in which only synthesized attributes are allowed.

Due to the restrictive form of S-attributed grammars, $L-$*ordered* grammars are used in practice.

Definition 21 L-attributed Grammars
An attribute grammar is said to be L-attributed *if and only if each inherited attribute of $X_{p,j}$ in the production $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p}$ depends only on the attrributes in the set $\bigcup_{k\in\{1,\cdots,j-1\}} Inh(X_{p,k}) \cup Syn(X_{p,0})$ for $j = 1, \ldots, n_p$.*

AGLEARN [8] is an interactive method for learning semantic functions of attribute grammars. The method uses background knowledge for learning semantic functions of S-attributed and L-attributed grammars. The given context-free grammar and background knowledge allow one to restrict the space of relations and give a smaller representation of data. The basic idea behind this method is that the learning problem of semantic functions is transformed to a propositional form and the hypothesis induced by a propositional learner can be transformed back into semantic functions. This approach was motivated by the fact that there is a close relationship between attribute grammars and logic programs [6].
AGLEARN uses the same concept as Inductive Logic Programming (ILP) but has a different representation. The background knowledge and the concepts are represented in the form of attribute grammars. One example contains a string which can be derived from the target nonterminal. We presuppose that the underlying context-free grammar is given. The task of AGLEARN is to infer the semantic functions associated with the production rule. In the learning process the grammar, background semantic functions and the examples are made use of.

We developed a parallel method for learning semantic functions of AGs (using a general purpose multi paradigm AG evaluator) based on the AGLEARN method. We handled the so-called circuity problem and gave parallel algorithms for S-attributed and L-attributed grammars as well.

Thesis 4

Our method (PAGELEARN) is parallelized according to three aspects:

1. Every semantic rule of the target rules is learnt in parallel.
2. The Ag evaluator is a parallel parser, so the attributed tree built for the actual example is handled in parallel. Moreover, this facility makes it possible to learn many semantic rules concurrently.
3. The concurrent learning of many semantic rules allows for the possibility of reducing the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation arrises. While the main problem of the sequential system was the large number of the user queries, this method improves the efficiency of the previous method from this aspect too. The reduction of user queries depends (on the training examples) on the number of circuits appearing among the waiting rules during the learning method.

6. SUMMARY OF THE AUTHORS CONTRIBUTION

I. Data Flow Analysis of Logic Programs:

The idea of applying the philosophy of a relevant slice to Logic Programs came from Tibor Gyimóthy. The relevant papers were written in the main by myself, but the development of the main algorithm is a common work of the authors. So, the the formalization of the algorithms, the necessary structures (Augmented SLD-tree, Skeleton(n)), the exteneded slice definitions and the necessary theorems were made by myself. The author assisted in the implementation of some ideas, but the main implementer was László Harmath.

II. Data Flow Analysis of Constraint Logic Programs:

The relevant papers were written in the main by myself. Most of the results of this research topic were made by myself too with supporting suggestions and ideas from Jan Małuszyński and Tibor Gyimóthy. Jan Małuszyński proposed the semantic definition of a slice of a Constraint Set and he clarified some formalisms. The main implementer was László Harmath.

III. Learning of Constraint Logic programs:

The relevant paper was written myself. The idea of examining the adaptability of the specialization technique for CLP programs came from Tibor Gyimóthy. The adaptation, the necessary theoretical background (including the definitions of the unfolding transformation, the theorems about the logical and operational semantics preservation) and also the formalization of the algorithms were made by myself. The implementation was done by myself and an informatics student.

IV. Learning Semantic Functions of Attribute Grammars:

This work was carried out by myself without the description of the PAGE system (parallel AG evaluator), but this system is not relevant from the view of the formalised parallel method since it can be substituted by an appropriate parallel AG parser.

7. RELATED PAPERS OF THE AUTHOR

[1] **Szilágyi, Gy.**, Małuszyński, J., Gyimóthy, T., 2002. Static and Dynamic Slicing of Constraint Logic Programs. Journal of Automated Software Engineering, Kluwer Academic Published, Vol. 9, No. 1, Jan 2002, pages 41-65.

[2] **Szilágyi, Gy.**, Gyimóthy, T., Małuszyński J., 2000. Slicing of Constraint Logic Programs. In Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG'2000), Munich, Germany, pages 176-187.

[3] **Szilágyi, Gy.** and Thanos, A. M., 2000. PAGELEARN: Learning Semantic Functions of Attribute Grammars in Parallel. Journal of Computing and Information Technology (C.I.T.), Vol. 8, No. 2, pages 115-131.

[4] **Szilágyi, Gy.**, Harmath, L. and Gyimóthy, T., 2001. Debug Slicing of Logic Programs. Acta Cybernetica, Vol. 15, No.2, pages 257-278 .

---

[5] Harmath, L., **Szilágyi, Gy.** and Gyimóthy, T., 2000. Debug Slicing of Logic Programs. Conference of PhD Students on Computer Sciences 2000, Hungary, pages 43-44.

[6] Harmath, L., **Szilágyi, Gy.**, Gyimóthy, T., Csirik, J., 1999. Dynamic Slicing of Logic Programs. In Poceedings of the Program Analysis and verification, Fenno- Ugric Symposium (FUSST'99), Tallin, Estonia, pages 101-113.

[7] **Szilágyi, Gy.** and Gyimóthy, T., 2003. Learning of Constraint Logic Programs by Combining Unfolding and Slicing. Submitted to AI Communications, The European Journal on Artificial Intelligence.

[8] **Szilágyi, Gy.**, Gyimóthy, T. and Małuszyński, J., 1998. Slicing of Constraint Logic Programs. Technical Report, Linköping University Electronic Press 1998/020,
www.ep.liu.se/ea/cis/1998/020.

Other Papers:

[9] Juhos, I., **Szilágyi, Gy.**, Csirik, J., Szarvas, Gy., Szeles, T., Kocsis, A., Szegedi, A., 2002. Time Series Prediction Using Artificial Intelligence Methods. In Proceedings of Conference of PhD Students in Computer Science (CS^2 2002), Szeged, Hungary.

[10] Hócza, A., **Szilágyi, Gy.** and Gyimóthy, T., 2002. LL Frame System of Learning Methods. In Proceedings of Conference of PhD Students in Computer Science (CS^2 2002), Szeged, Hungary.

References

[1] Alblas, H., 1991. Introduction to Attribute Grammars. LNCS 545, Springer Verlag.

[2] Alexin, Z., Gyimóthy, T., Boström, H., 1996. IMPUT: An Interactive Learning Tool based on Program Specialization submitted to the Intelligent Data Analysis Journal published by the Elsevier Ltd.

[3] Boström, H. and Idestam-Almquist, P., Specialization of Logic Programs by Pruning SLD-trees. Proceedings of the Fourth International Workshop on Inductive Logic Programming (ILP-94), Bad Honnef/Bon, Germany, pages 31-47.

[4] Boye, J. Paakki, J. and Małuszyński, J., 1993. Dependency-Based Groundness Analysis of Functional Logic Programs. Research Report LiTH-IDA-R93-20, Department of Computer and Information Science, Linköping University.

[5] Boye, J. Paakki, J. and Małuszyński, J., 1993. Synthesis of Directionality Information for Functional Logic Programs. In Proceedings of 3rd International Workshop on Static Analysis, LNCS 724, Springer-Verlag, pages 165-177.

[6] Deransart, P. and Małuszyński, J., 1993. A grammatical view of logic programming. The MIT Press.

[7] Gyimóthy, T. Beszédes, À. and Forgács, I., 1999. An Efficient Relevant Slicing Method for Debugging. In Proceedings of the 7th European Software Engineering Conference (ESEC'99), LNCS 1687 Springer Verlag, Toulouse, France, pages 303-322.

[8] Gyimóthy, T. and Horváth, T., 1997. Learning Semantic Functions of Attribute Grammars. Nordic Journal of Computing, Vol. 4 (1997), pages 287-302.

[9] Harmath, L., Szilágyi, Gy., Gyimóthy, T., Csirik, J., 1999. Dynamic Slicing of Logic Programs. In Proceedings of the Program Analysis and verification, Fenno- Ugric Symposium (FUSST'99), Tallin, Estonia, pages 101-113.

[10] Harmath, L., Szilágyi, Gy. and Gyimóthy, T., 2000. Debug Slicing of Logic Programs. Conference of PhD Students on Computer Sciences 2000, Hungary, pages 43-44.

[11] Horwitz, S. and Reps, T., 1992. The Use of Program Dependence Graphs in Software Engineering. In Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia, pages 392-411.

[12] Jaffar, J. and Maher, M.J., 1994. Constraint logic programming: A Survey. The Journal of Logic Programming, 19/20, pages 503-582.

[13] Knuth, Donald E., 1968. Semantics of Context-Free Languages. Mathematical Systems Theory, volume 2, number 2, pages 127 -145.

[14] Knuth, Donald E., 1968. Semantics of Context-Free Languages, correction. Mathematical Systems Theory, Volume 5, Number 1, pages 95 - 96.

[15] Knuth, Donald E., 1990. The Genesis of Attribute Grammars. In Proceedings of the International Conference on Attribute Grammars and their Applications, Springer-Verlag, LNCS, 1990, Volume 461, pages 1 - 12.

[16] Lavrac, N. and Dzeroski, S., 1994. Inductive Logic Programming: Techniques and Applications. Ellis Horwood.

[17] Lloyd, J.W., 1987. Foundations of Logic Programming, Spinger Verlag.

[18] Marriott, K. and Stuckey, P.J., 1998. Programming with Constraints. An Introduction. The MIT Press.

[19] Muggleton S., 1994. Inductive Logic Programming. The ACM Press.

[20] Nilsson, U. and Małuszyński, J., 1995. Logic, Programming and Prolog. John Wiley and Sons Ltd.

[21] Pereira, L.M. and Calejo, M., 1988. A Framework for Prolog Debugging. In Proceedings of ICLP/SLP'88, pages 481-495.

[22] Shapiro, E., 1983. Algorithmic Debugging. The MIT Press.

[23] Szilágyi, Gy., Harmath, L. and Gyimóthy, T., 2001. Debug Slicing of Logic Programs. Acta Cybernetica, Vol. 15, No.2, pages 257-278.

[24] Szilágyi, Gy., Małuszyński, J., Gyimóthy, T., 2002. Static and Dynamic Slicing of Constraint Logic Programs. Journal of Automated Software Engineering, Kluwer Academic Published, Vol. 9, No. 1, Jan 2002, pages 41-65.

[25] Szilágyi, Gy., Gyimóthy, T., Małuszyński, J., 2000. Slicing of Constraint Logic Programs. In Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG'2000), Munich, Germany, pages 176-187.

[26] Szilágyi, Gy., Gyimóthy, T. and Małuszyński, J., 1998. Slicing of Constraint Logic Programs. Technical Report, Linköping University Electronic Press 1998/020, *www.ep.liu.se/ea/cis/1998/020*.

[27] Szilágyi, Gy. and Thanos, A. M., 2000. PAGELEARN: Learning Semantic Functions of Attribute Grammars in Parallel. Journal of Computing and Information Technology (C.I.T.), Vol. 8, No. 2, pages 115-131.

[28] Szilágyi, Gy. and Gyimóthy, T., 2003. Learning of Constraint Logic Programs by Combining Unfolding and Slicing. Submitted to AI Communications, The European Journal on Artificial Intelligence.

[29] Wilhelm, R., 1979. Attributierte Grammatiken. Informatik Spektrum, Volume 2, pages 123 - 130.